FlareDTDG: Harnessing Temporal Recency for Scalable Discrete-Time Dynamic Graph Training

Wenjie Huang
Zhejiang University
wjie@zju.edu.cn

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

Jing Cao
State Key Laboratory of Blockchain and Data Security, Zhejiang University
Zhejiang Key Laboratory of Big Data Intelligent Computing, Hangzhou City University
jingcao@hzcu.edu.cn

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

Xinyu Wang
Zhejiang University
wangxinyu@zju.edu.cn

Mingli Song
Zhejiang University
brooksong@zju.edu.cn

Sai Wu
Zhejiang University
wusai@zju.edu.cn

Chun Chen
Zhejiang University
chenc@cs.zju.edu.cn

ABSTRACT

Discrete-time dynamic graphs (DTDGs), modeled as snapshot sequences, are widely used to capture temporal evolution in relational systems. Scaling DTDG training remains challenging: full-batch methods incur prohibitive memory and communication costs, while sampling or offloading often sacrifices accuracy or efficiency. A major limitation of existing frameworks is that they treat all snapshots equally, ignoring the temporal recency effect, where recent snapshots are typically far more predictive than older ones. We introduce FlareDTDG, a distributed framework that exploits temporal recency for efficient and scalable training. Its core is hybrid batching with temporal decay, which applies full-batch processing to recent snapshots, while progressively coarsely sampling older ones to form hybrid batches. We also integrate two co-designed optimizations: fast graph reconstruction via shrinking to eliminate cross-snapshot remapping, and adaptive comm-comp overlap scheduling to reduce synchronization overhead. Experiments show FlareDTDG achieves 1.4-2.5 times faster training and 10–85% lower GPU memory usage than full-batch baselines, while preserving accuracy. It also scales to graphs with 100M nodes per snapshot, where existing systems fail due to memory limits or degraded performance.

PVLDB Reference Format:
Wenjie Huang, Rui Wang, Jing Cao, Tongya Zheng, Xinyu Wang, Mingli Song, Sai Wu, and Chun Chen. FlareDTDG: Harnessing Temporal Recency for Scalable Discrete-Time Dynamic Graph Training. PVLDB, 19(7): 1614 - 1627, 2026.
doi:10.14778/3801059.3801073

*Corresponding author.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 19, No. 7 ISSN 2150-8097.
doi:10.14778/3801059.3801073

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/wjie98/FlareDTDG.

1 INTRODUCTION

Dynamic graph neural networks (DGNNs) are essential for modeling evolving relational systems [19, 35], with applications spanning social networks [4, 15, 21, 52], traffic forecasting [3, 11, 23, 42, 48], etc. One common approach is to represent temporal evolution using *discrete-time dynamic graphs* (DTDGs), where the system is depicted as a sequence of graph snapshots [23, 27, 28, 45, 51], capturing the nodes and edges at specific time steps. At large scale, DTDGs can span millions of nodes and thousands of snapshots, creating scalability challenges.

Many DTDG models and frameworks have been proposed [8, 40, 44] to capture both snapshot structure and temporal dependencies by combining traditional GNNs [6, 16, 17, 24] with temporal models [12]. Current DTDG models often rely on *full-batch processing*, loading all snapshots jointly for parameter updates. While this approach maintains temporal fidelity, it becomes impractical for large graphs due to high GPU/CPU memory demands and communication bottlenecks, forcing truncated temporal horizons or reduced model complexity that limit long-term dependency modeling. To improve scalability, recent distributed DTDG frameworks [1, 2, 10, 33, 34, 37], have introduced three primary strategies: (1) *snapshot-parallel approaches* partition graph snapshots across multiple GPUs, and extend full-batch training via optimizations like gradient checkpointing [1]. (2) *Sampling-based node-parallel approaches* partition graph by nodes rather than snapshots, distributing nodes (and their sampled edges) across GPUs to reduce memory and computation costs [2, 10, 37, 49, 50], but introduce sampling bias and extra remapping overhead. (3) *Distributed caching and offloading* store features and historical embeddings in CPU memory to lower GPU usage and inter-GPU communication, at the cost of heavy CPU memory overhead.

Despite these advances, existing DTDG frameworks remain limited on large graphs due to two key bottlenecks. *(1) Memory explosion*: Loading multiple snapshots for large-scale graphs (e.g., `ogbn-papers100M` with 53 GB of node features) can exceed 3 TB of GPU memory for ESDG [1] or 56 TB of CPU memory for DynaHB [37]. Even sampling-based DynaGraph [10] consumes hundreds of GB and sacrifices accuracy. *(2) Communication overhead*: Global synchronization (e.g., all-to-all exchanges in ESDG), incurs non-overlapped communication costs amounting for 24–54% of total training time. These inefficiencies stem from treating all snapshots equally, overlooking the *temporal recency effect*, which shows that recent snapshots hold disproportionately greater predictive value than distant historical ones in dynamic graphs [14, 25, 26, 47]. Consequently, substantial computation and memory are expended on processing stale historical data with limited utility.

To bridge these gaps, we propose **FlareDTDG**, a novel distributed DTDG training framework that strategically incorporates temporal recency effect for efficient and scalable learning. Its core innovation is *hybrid batching with temporal decay*, which prioritizes snapshots based on temporal relevance. Unlike prior methods that treat all snapshots equally, FlareDTDG maintains full-batch processing for the most recent snapshots to preserve high-fidelity information, while older snapshots are progressively coarsely sampled, reducing memory and computation. These hybrid batches are then partitioned for distributed execution. Realizing this hybrid approach at scale requires overcoming graph reconstruction and communication bottlenecks. FlareDTDG introduces two co-designed optimizations: *fast graph reconstruction via shrinking*, which eliminates cross-snapshot remapping through global node remapping and shrinking; and *adaptive comm-comp overlap scheduling*, which reduces communication overhead via runtime-aware GPU scheduling. Our contributions are summarized as follows:

- We identify key scalability bottlenecks in existing distributed DTDG training frameworks, particularly the costs of processing multiple snapshot sequences for large graphs.
- We propose a temporal recency-aware DTDG training paradigm, i.e., *hybrid batching with temporal decay*, which hierarchically prioritize snapshots according to their temporal relevance using a trapezoid-shaped sliding window.
- We design two co-designed system optimizations: *fast graph reconstruction via shrinking* and *adaptive comm-comp overlap scheduling*, that jointly improve graph reconstruction efficiency and communication-computation overlap.
- Extensive experiments show that FlareDTDG achieves 1.4 to 2.5× faster training and 10 to 85% lower GPU memory usage than full-batch baselines, while maintaining accuracy and scaling effectively across diverse dynamic graph tasks. Moreover, FlareDTDG scales to graphs with 100M nodes per snapshot, where ESDG and DynaHB run out of memory and DynaGraph suffers from inefficiency and accuracy loss.

2 BACKGROUND AND MOTIVATION

This section first introduces the definition and training paradigm of Discrete-Time Dynamic Graphs (DTDGs), then reviews existing system optimizations for large graph training, and finally identifys the key bottlenecks in existing methods that motivate our design.

2.1 Discrete-Time Dynamic Graphs (DTDGs)

DTDGs provide a principled way to represent graph-structured data evolving over discrete time intervals [19, 35]. Formally, a DTDG is defined as a sequence of graph snapshots $\mathcal{G} = \mathcal{G}_1, \mathcal{G}_2, \dots, \mathcal{G}_T$, where each $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t)$ encodes the vertex and edge sets at time step t. The goal is to learn node embeddings $\mathbf{h}_v^t$ that capture both structural properties of each snapshot and temporal dependencies across snapshots. A common approach applies a two-stage architecture [11, 23, 28, 34, 51]: a GNN [16, 17] extracts spatial representations from each snapshot, which are then integrated over time using a temporal model like RNN [12]. This procedure is typically formalized as:

$$\mathbf{h}_v^t = \mathrm{RNN}(\mathbf{h}_v^{t-1}, \mathrm{GNN}(\mathcal{G}_t)),$$

where $\mathbf{h}_v^{t-1}$ is the embedding of node v at the previous time step.

2.2 Optimizations in Large Graph Training

Distributed DTDG training encounters significant system hurdles when dealing with large graphs. Two key bottlenecks dominate: (1) the massive memory footprint of snapshots and historical embeddings, and (2) heavy communication overhead from propagating temporal information across workers. To tackle these issues, previous studies have introduced various optimization techniques.

Single-GPU Solutions. Approaches like PiPAD [40], BLAD [8], and STGraph [44] focus on fine-grained resource utilization of dynamic graph training within a single GPU, optimizing batching, scheduling, and memory management to maximize efficiency on datasets with many small graph snapshots. However, they remain bound by the compute and memory limits of a single GPU, making them unsuitable for large-scale DTDG training.

Snapshot-Parallel Approaches. These methods partition graph snapshots across multiple GPUs. For instance, ESDG [1] assigns subsets of snapshots to different GPUs and redistributes node embeddings after each step to maintain temporal consistency, turning workload imbalance into structured communication. However, this approach assumes each GPU can hold at least one full snapshot, limiting scalability when snapshots are very large.

Sampling-based Node-Parallel Approaches. These methods partition graphs by nodes rather than snapshots, reducing memory demands by distributing nodes and sampling their neighbors. Frameworks like DynaGraph [10] build localized subgraph sequences, consisting of assigned nodes and their sampling edges, on each GPU instead of storing the full graph topology. To improve efficiency, they often cache aggregated messages to avoid redundant computation, and sometimes randomly repartition nodes across workers to enhance diversity and generalization. However, this approach faces several challenges. Neighbor sampling itself can be computationally expensive, and frequent repartitioning incurs substantial synchronization and remapping overhead. Moreover, random partitioning disrupts graph locality, further amplifying communication costs, and information loss from sampling can degrade model accuracy.

Distributed Caching and Offloading. To alleviate inter-GPU communication, these methods offload multi-hop features and historical embeddings to CPU memory. DynaHB [37], for instance, stores multi-hop neighbor features on the CPU and retrieves them only when required. It further integrates a hybrid batching strategy

Table 1: Peak memory usage for training MPNN-LSTM.

Dataset	rec-amazon	soc-youtube	soc-bitcoin	ogbn-papers100m
# Nodes	2.1M	3.2M	24.6M	111.1M
# Snapshots	70	80	90	222
Feature Sizes	1.1GB	1.7GB	12.8GB	55.6GB
ESDG GPU	78.4GB	65.6GB	81.6GB	GPU OOM >3.0TB
DynaHB GPU	8.1GB	6.5GB	38.4GB	CPU OOM
DynaHB CPU cache	322.2GB	235.6GB	1.5TB	CPU OOM >56.6TB
DynaGraph GPU	25.6GB	20.4GB	38.4GB	732.8GB

Note: "GPU OOM" indicates out-of-GPU-memory, "CPU OOM" denotes out-of-host-memory, and > is a lower-bound estimate of required memory.

optimized via reinforcement learning (RL) to enhance data loading efficiency. While effective in reducing GPU memory demands and GPU-to-GPU synchronization, this approach shifts the burden to CPU memory usage and CPU-to-GPU PCIe bandwidth, and the RL-based optimization can suffer from slow convergence, limiting throughput during the early stages of training.

2.3 Limitations of Existing Methods

Despite the above recent progress, current DTDG training frameworks still face fundamental limitations that restrict scalability and efficiency for training large-scale dynamic graphs.

#L1: Memory Explosion in Large Graphs. Memory overhead remains a critical limitation for DTDG frameworks, especially on large graphs, as illustrated in Table 1, which reports the peak memory usage for training MPNN-LSTM [27] across multiple datasets. For the largest dataset `ogbn-papers100M` [13], some values are lower-bound estimates (>) because actual requirements exceeded available hardware, making full training infeasible. This bottleneck arises in two ways. Firstly, individual snapshots can be massive as they need to store dense node features in addition to the graph structure information. For example, loading the 128-dimensional float32 node features for a single snapshot of `ogbn-papers100M` (with $N = 111.1M$ nodes) requires approximately $111.1 \times 128 \times 4/1024 = 55.6$ GB, surpassing the capacity of most GPUs and making single-step processing without partitioning or sampling unfeasible. Secondly, capturing temporal dependencies often involves loading multiple snapshots simultaneously and maintaining hidden states and gradients across T time steps, leading to memory usage growing proportionally with the number of snapshots. Formally, the memory demand scales as $O(T \cdot N \cdot (d_{feat} + d_{hidden})$, where d_{feat} and d_{hidden} represent the dimension of node features and hidden size, respectively. For example, ESDG [1] ran out of GPU memory on `ogbn-papers100M`, with estimated memory needs exceeding 3.0 TB, far beyond the combined $48 \times 16 = 768$ GB of our 16 A40 GPUs.

Efforts to offload multi-hop neighbor features to CPU memory, such as in DynaHB [37], simply relocate the bottleneck. In the case of `ogbn-papers100M`, CPU memory needs surpassed 56.6 TB due to its vertex-cached partitioning approach, which duplicates cross-boundary features to reduce communication, leading to CPU OOM errors. Sampling-based methods like DynaGraph [10] attempt to mitigate GPU memory pressure by processing only a subset of sampled snapshots, but still required 732.8 GB of GPU memory for this dataset, nearly maxing out 16 A40 GPUs. This strategy also introduces new limitations: neighbor sampling is costly, repartitioning adds remapping and communication overhead, and sampling loses information, lowering accuracy. Together, these results highlight the severity of the memory bottleneck, which overwhelms both GPU and CPU resources and fundamentally limits the scalability of existing DTDG training frameworks on truly large graphs.

#L2: Communication Scalability Limits. A second critical bottleneck stems from the communication cost of synchronizing temporal dependencies across distributed workers. Global exchanges, such as ESDG's all-to-all embedding redistribution [1], generate heavy network traffic, leading to non-overlapped communication accounting for 24–54% of training time as shown in Figure 1. This cost grows with the number of snapshots and is sensitive to graph topology. Methods like DynaGraph and DynaHB reduce communication via sampling or caching, but introduce extra computation and memory overhead (as discussed in #L1). As a result, existing frameworks face a fundamental trade-off between high communication costs and heavy memory/computation demands, limiting scalability for large graphs or long temporal horizons.

3 DESIGN OF FLAREDTDG

3.1 Observation and Main Idea

Observation. Discrete time dynamic graphs (DTDGs) exhibit strong *temporal recency effect* [25, 26, 31, 47]: Recent graph snapshots hold greater predictive power for the near-future state of the graph, while the influence of historical data diminishes quickly as we move back in time [9, 20, 38]. Yet, most existing DTDG training methods [1, 2, 10, 33, 34, 37] treat all snapshots equally, incurring substantial memory and computational overhead on historical information processing with limited effectiveness. This *temporal recency effect motivates a shift toward prioritizing recent snapshots while progressively reducing the resolution of older ones.*

Main Idea. Based on this observation, we propose a *temporal recency-aware computation paradigm* that concentrates full-graph computation on recent snapshots while applying progressively coarser sampling to older ones. As snapshots recede in time, fewer nodes and edges are processed, reducing memory and computation costs. To preserve long-term context without storing full-resolution older snapshots, we reuse historical RNN hidden states, lowering

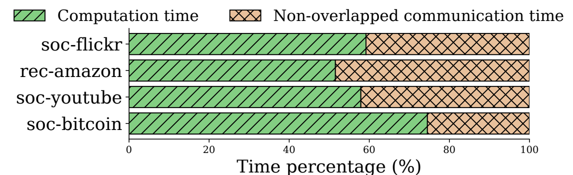

Figure 1: Non-overlapped communication time percentage for training MPNN-LSTM model on ESDG framework.

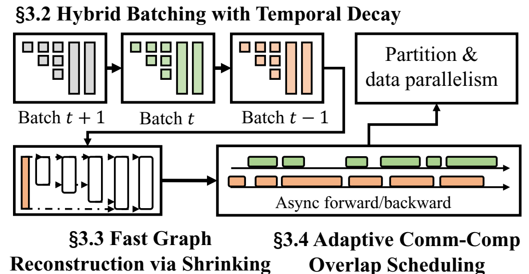

Figure 2: Overview of the FlareDTDG.

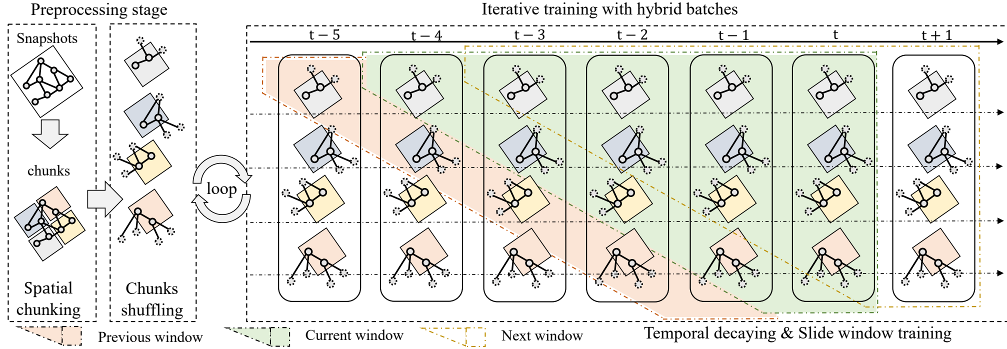

Figure 3: Illustration of hybrid batching with temporal decay. The training batch at each time step forms a trapezoid-shaped sliding window, composed of a rectangular base of recent full snapshots (spatial blocks) and a triangular tail of decayed subgraphs (temporal blocks). The window moves sequentially over time, inheriting the hidden states of historical blocks to maintain temporal continuity. The trapezoid's shape reflects the decaying importance of older historical data.

sampling variance and maintaining learning continuity. This design directs computational effort toward the most predictive snapshots while efficiently downscaling the influence of older data, striking an effective balance between scalability and accuracy, making it well-suited for large, real-world dynamic graphs.

Overview. To realize this idea and enable efficient DTDG training on large graphs, we design FlareDTDG, a distributed framework built on a *temporal recency-aware computation paradigm* that hierarchically prioritizes snapshots. Figure 2 provides an overview of FlareDTDG, which incorporates three key techniques: (1) *Hybrid batching with temporal decay*, balancing training efficiency and accuracy; (2) *Fast graph reconstruction via shrinking*, reducing cross-snapshot costs; and (3) *Adaptive communication-computation overlap scheduling*, mitigating synchronization delays. The following subsections describe each technique in detail.

3.2 Hybrid Batching with Temporal Decay

Hybrid Batching Strategy. We first propose a *hybrid batching with temporal decay* mechanism to form the cornerstone of FlareDTDG. As illustrated in Figure 3, at each training step, we use the sliding window form a training batch consisting of: (1) a small number of recent full snapshots (*spatial blocks*), and (2) sampled subgraphs from older snapshots (*temporal blocks*), thus efficiently capturing the *temporal recency effect* of DTDGs while reducing memory and computation overhead. Formally, let $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t)$ represent the graph snapshot at time t. The batch $\mathcal{B}_t$ at time t is defined as:

$$\mathcal{B}_t = \mathcal{B}_t^{\text{spatial}} \cup \mathcal{B}_t^{\text{temporal}},$$

where $\mathcal{B}_t^{\text{spatial}} = \mathcal{G}_{t-k+1}, \ldots, \mathcal{G}_t$ contains k most recent snapshots, and $\mathcal{B}_t^{\text{temporal}} = \hat{\mathcal{G}}_{t-k-m+1}, \ldots, \hat{\mathcal{G}}_{t-k}$ contains m sampled subgraphs. Here, each $\hat{\mathcal{G}}_i \subseteq \mathcal{G}_i$ is a temporal block.

As shown in Figure 3, spatial blocks provide high-resolution information of recent snapshots, while temporal blocks capture long-term dependencies of sampled subgraphs at a reduced cost. This hybrid batching strategy naturally shapes the active data within the sliding window into a trapezoidal structure. Note that this "trapezoid" serves as a visual metaphor for the shifting information density rather than a rigid geometric constraint, with a wide rectangular base for recent full snapshots for immediate precision and a narrowing triangle towards the past for sampled subgraphs to maintain long-term dependencies. This trapezoidal shape intuitively embodies our resource allocation philosophy of concentrating computation on high-value immediate history while maintaining a compressed context of long-term dependencies.

Spatial Blocks with Temporal Continuity. Spatial blocks are selected via a sliding window that advances one time step at a time. At training step t, the window captures the k most recent snapshots to form the spatial block $\mathcal{B}_t^{\text{spatial}} = \{\mathcal{G}_{t-k+1}, \ldots, \mathcal{G}_t\}$. This selection approach ensures that each spatial block contains a continuous sequence of the latest graph structures, offering a detailed view of recent dynamics crucial for accurate predictions. Our method enforces strict temporal continuity by inheriting hidden states $\bar{\mathbf{h}}_v^{t-1}$ from the previous step, allowing the RNN to effectively model long-term dependencies: $\mathbf{h}_v^t = \text{RNN}(\bar{\mathbf{h}}_v^{t-1}, \text{GNN}(\mathcal{B}_t))$. To introduce randomness and prevent overfitting, the sequence of snapshots $\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_T\}$ is split at a random point t into two segments: $\{\mathcal{G}_1, \ldots, \mathcal{G}_t\}$ and $\{\mathcal{G}_{t+1}, \ldots, \mathcal{G}_T\}$ at the start of each epoch. Training then proceeds first on $\mathcal{G}t + 1, \ldots, \mathcal{G}_T$ and subsequently on $\mathcal{G}_1, \ldots, \mathcal{G}_t$. This randomized segmentation prevents the model from overfitting to a fixed snapshot traversal order.

Temporal Block Sampling with Spatial Continuity. Guided by the temporal recency effect, we sample temporal blocks to create low-resolution yet structurally representative subgraphs of older snapshots. This approach reduces memory usage, computation, and communication overhead, while limiting interference from outdated information. To preserve structural integrity, temporal blocks $\hat{\mathcal{G}}_t$ are sampled at the chunk level rather than as isolated nodes, following three steps: (1) *Spatial chunking* partitions the graph into fixed chunks using graph partitioning algorithms like METIS [18] to preserve local graph structures. (2) *Time-series shuffling* randomly shuffles chunks once per epoch to create a global priority list for decay, ensuring randomness and preventing overfitting. (3) *Chunk selection* occurs within the sliding window at each time step: new chunks are taken from the top of the shuffled list, while older chunks at the tail are progressively dropped to form each temporal block. This progressive selection strategy creates a triangular decay pattern where older blocks contain fewer chunks, yet all chunks in older blocks within the window are subsets of those in more recent blocks, preserving spatial continuity over time. Formally, each chunk C_j from snapshot $\mathcal{G}_t$ is a small subgraph defined as

$C_j = (\mathcal{V}_j^{\text{chunk}}, \mathcal{E}_j^{\text{chunk}})$, where $\mathcal{V}_j^{\text{chunk}} \subseteq \mathcal{V}_t$ and $\mathcal{E}_j^{\text{chunk}} \subseteq \mathcal{E}_t$ are the nodes and edges within the chunk. A sampled temporal block $\hat{\mathcal{G}}_t$ is the union of selected chunks: $\hat{\mathcal{G}}_t = \bigcup_j C_{t,j}$, where $C_{t,j}$ is the j-th chunk from $\mathcal{G}_t$. Edges connecting different chunks are also retained to preserve structural integrity.

Temporal Decay with Sliding Window. Each training batch forms a trapezoid-shaped sliding window, which is updated dynamically as the window slides forward one time step at a time across the sequence of snapshots. When moving to time t, three key actions occur: (1) the newest full snapshot $\mathcal{G}t$ enters the window as the most recent spatial block; (2) the previously oldest snapshot in spatial block, $\mathcal{G}t-k+1$, transits into the role of the newest temporal block, $\hat{\mathcal{G}}_{t-k}$, and is subjected to sampling for the first time; and (3) the oldest chunks in temporal block, $\hat{\mathcal{G}}_{t-k-m}$, is discarded from the window. This continuous process ensures training always incorporates recent high-fidelity data while maintaining a decaying, sampled view of the immediate past. Older temporal blocks form a nested, decaying sequence where $\hat{\mathcal{G}}_{t-1} \subseteq \hat{\mathcal{G}}_t$, reflecting diminishing importance. This parameterization is achieved via an exponential decay mechanism. Let $\mathcal{N}_t$ be the number of chunks in in block $\hat{\mathcal{G}}t$; the next older block contains $\mathcal{N}t-1 = \lfloor \beta \cdot \mathcal{N}_t \rfloor$ chunks, where $\beta \in (0,1)$ is the decay factor. We define β via a retention ratio α for the oldest temporal block relative to the last spatial block: $\beta = \sqrt[m]{\alpha}$, where m is the number of temporal blocks. This formulation allows intuitive control of the decay rate and the resolution of distant historical information.

Benefit Discussion. This hybrid batching balances efficiency and accuracy by prioritizing recent snapshots and decaying older ones, reducing memory and compute overhead without losing structural context. Besides, subgraph-based blocks and strict temporal continuity preserve long-term dependencies, enabling learning from both short-term dynamics and distant history. This approach is especially effective for massive DTDGs where snapshot size and sequence length typically limit traditional training methods.

3.3 Fast Graph Reconstruction via Shrinking

We introduce a *fast graph reconstruction via shrinking* strategy to eliminate the costly node remapping required for each step when training with decayed temporal blocks, as shown in Figure 4.

Challenge of Frequent Node Remapping. Hybrid batching with temporal decay produces temporal blocks with non-contiguous node IDs, since older snapshots are sampled more coarsely and retain only subsets of chunks. This non-contiguous indexing is incompatible with GNN libraries like DGL [41] and PyG [7], which require compact, contiguous node indexing for efficient tensor operations. A naive solution is to remap node IDs at every training step, but this repeated operation incurs substantial computational and memory overhead to input preparation, as well as frequent output state alignment, amplifying training costs.

Global Node Remapping and Shrinking. To eliminate repetitive re-indexing overhead, we decouple expensive global remapping from lightweight per-step updates. Before each epoch, every chunk C_i is assigned a *survival time* s_i via random permutation. Chunks are then globally sorted by s_i, and node IDs across all snapshots are remapped once such that longer-surviving nodes are assigned lower IDs as show in Figure 4. During training, subgraph reconstruction

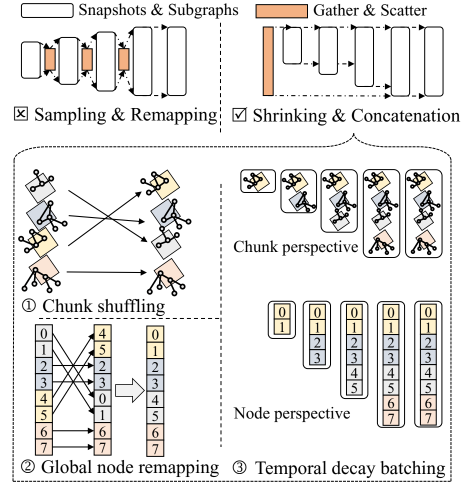

Figure 4: Illustration of fast graph reconstruction via shrinking. ① Assign survival times to chunks and sort them globally; ② Remap node IDs within chunks according to survival times; and ③ Apply temporal decay to drop high-ID chunks while preserving continuity. The resulting temporal blocks maintain consistent indexing without extra remapping.

reduces to a simple shrinking operation: nodes in discarded chunks which have the largest IDs are pruned, leaving a compact node set with a contiguous low-ID range. This ensures structural consistency and index-free state alignment for efficient tensor operations without repeated remapping costs.

Spatial Chunking and chunk-level temporal decay. To enable partition-parallel distributed training and manageable granularity for temporal decay, we use spatial chunking. To ensure node identities remain consistent across the timeline, we do not partition snapshots individually. Instead, we aggregate all snapshots into a single union graph and execute METIS [18] during the offline preprocessing stage to assign a persistent chunk ID to each node. This persistent assignment guarantees that a specific node always resides in the same spatial chunk at every time step, which is essential for the seamless transmission of RNN hidden states. Although this creates a cross-chunk edge boundary effect when neighbors are decayed, it is a necessary trade-off for scalability, avoiding costly per-snapshot partitioning and scattered node management. As shown in Table 12, this preprocessing step accounts for a negligible fraction ($\leq 1.02\%$) of the total training time, while the resulting contiguous memory layout significantly improves the throughput of the dominant computation phases.

Building upon the persistent chunk IDs, FlareDTDG implements the temporal decay mechanism via an epoch-level shuffling and memory reorganization process. At the start of each training epoch, we randomly shuffle the chunk sequence to determine their survival priority within the sliding window. We then perform a single global remapping to reorganize the memory layout according to this shuffled order. This design allows historical snapshots to be reconstructed through a simple tensor shrinking operation (i.e., selecting a contiguous range of the lowest node IDs) at each training step. By moving the complexity of re-indexing to an epoch-level preparation

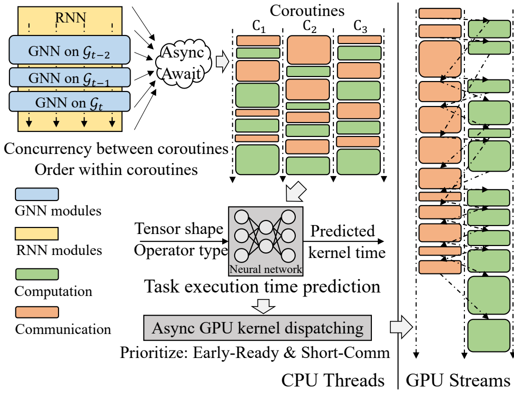

Figure 5: Illustration of adaptive comm–comp overlap scheduling. The training process is transformed into asynchronous coroutines using Python's async/await primitives. A lightweight neural network predicts task execution times, enabling a greedy scheduling strategy (Early-Ready & Short-Comm) to maximize communication-computation overlap. Independent streams are used for concurrency.

phase, we ensure high-speed, contiguous tensor operations on the GPU throughout the training process.

Benefit Discussion. By performing global node remapping once per epoch and using incremental shrinking at each step, this approach significantly reduces computational and memory costs. It streamlines the integration of spatial and temporal blocks, preserves temporal continuity, and enables scalable training on large graphs.

3.4 Adaptive Comm–Comp Overlap Scheduling

This subsection proposes an adaptive communication–computation overlap scheduling technique, by using a neural network to predict the execution time of each task, enabling the system to overlap communication with computation effectively and maximize GPU utilization, as illustrated in Figure 5.

Challenge of communication bottlenecks. Distributed training for large-scale DTDGs faces significant workload imbalances and communication bottlenecks due to the varying sizes of spatial and temporal blocks. Spatial blocks often involve heavy cross-GPU communication, while temporal blocks are largely computation-bound and can be processed locally. Without careful coordination, GPUs may remain idle while waiting for data transfers, limiting overall training efficiency.

Coroutine-Based Execution Model. To manage this heterogeneous workload, we treat the processing of each snapshot as an independent coroutine [22], implemented with Python's `async/await`. Communication-intensive spatial blocks are dispatched across GPUs, while temporal blocks are executed locally without communication. By interleaving communication and computation from different snapshots, this coroutine-based model allows asynchronous execution with minimal code modifications, effectively reducing idle time. This strategy is particularly effective for DTDGs, where snapshots are relatively independent and can be processed concurrently.

Execution Time Prediction. To reduce GPU idle periods, we should proactively schedule different communication and computation tasks. Instead of waiting for completion signals, we could predicts task execution times via a lightweight Multi-Layer Perceptron (MLP). For a task g with features $\mathbf{x}_g$ (e.g., tensor size, task type), the MLP predicts its execution time $\hat{t}_g = f_\theta(\mathbf{x}_g)$. The MLP is trained to minimize MSE between predicted and actual execution times, enabling proactive and informed scheduling decisions.

Greedy Scheduling Strategy. Based on the predictions, we use a greedy scheduling strategy called *Early-Ready & Short-Comm*. Each communication task and its dependent computation tasks are grouped into a set $\mathcal{G}$. The priority of this set is a tuple $p(\mathcal{G}) = (\hat{t}_{\text{ready},\mathcal{G}}, \hat{t}_{\text{comm},\mathcal{G}})$, where $\hat{t}_{\text{ready}}$ is the predicted ready time and $\hat{t}_{\text{comm}}$ is the predicted communication time. The scheduler prioritizes task sets that are ready earlier and have shorter communication times. By executing communication and computation in separate GPU streams, this strategy effectively hides communication latency behind computation, improving overall training throughput.

Benefit Discussion. The adaptive asynchronous overlap technique offers several advantages: (1) it minimizes communication delays by dynamically scheduling tasks based on execution time predictions; (2) it maximizes GPU utilization by overlapping communication and computation; and (3) it requires minimal modifications to existing model code, making it easy to integrate into existing frameworks.

3.5 Implementation

We implement FlareDTDG using PyTorch [29] and DGL [41]. The framework consists of two core modules: data loader and the model execution runtime. The data loader integrates spatiotemporal batching and structured indexing, performing all necessary preprocessing at the start of each epoch and loading data asynchronously. The runtime incorporates the adaptive scheduler, which automatically partitions the model, predicts task times, and dispatches tasks asynchronously. To minimize code intrusion, we leverage `async/await` primitives, as shown in the GCN layer example below.

```
1 async def async_forward(self, g, x, route):
2   x = await route.async_forward(x)  # overlapping comm
3   return self.msg_pass(g, x) @ self.weight + self.bias
```

The overall training process is summarized in Algorithm 1. Before each epoch, global node indices are remapped using the fast graph reconstruction via shrinking strategy (§3.3, lines 3–4). At each step, a hybrid batch is constructed with the temporal decay mechanism (§3.2, lines 6–7). The forward pass is executed as an asynchronous coroutine, with the scheduler overlapping communication and computation (§3.4, line 8). This design unifies all proposed techniques into a cohesive and efficient training pipeline.

Scope of Applicability. FlareDTDG supports diverse GNN tasks by balancing the full-snapshot window (K) and decay rate (α). For tasks dominated by short-term dynamics, such as traffic forecasting [11, 23, 48] or recommendation [36, 43, 46], the framework can be configured with a moderate K and lower α to prioritize high-fidelity recent interactions where most predictive signals reside. For domains where long-term structural evolution is more critical, a moderate K combined with a higher-resolution historical sequence helps the model capture distant dependencies that are discarded by "recent-only" baselines. This parameterization enables FlareDTDG to preserve essential long-range dependencies that adapt to the specific memory-accuracy trade-offs of different workloads.

Algorithm 1 FlareDTDG Training Loop

1: **Input:** Snapshots $\{\mathcal{G}_1, \ldots, \mathcal{G}_T\}$, model parameters θ, window size $W = k + m$
2: **for** each epoch **do**
3: Precompute global indices and chunk shuffling (Sec 3.3)
4: Randomly split sequence $\{\mathcal{G}_t\}$ for training
5: **for** each training step t from W to T **do**
6: $\mathcal{B}_t \leftarrow$ BuildHybridBatch($\{\mathcal{G}_{t-W+1}, \ldots, \mathcal{G}_t\}$) (Sec 3.2)
7: Inherit historical hidden states $\bar{\mathbf{h}}^{t-1}$
8: $\mathbf{h}^t \leftarrow$ AsyncForward($\mathcal{B}_t, \bar{\mathbf{h}}^{t-1}$) (Sec 3.4)
9: Calculate loss $\mathcal{L}$ based on $\mathbf{h}^t$
10: Backpropagate and update θ
11: **end for**
12: **end for**

Regarding tasks with periodic or cyclic patterns, while "recent-only" methods rely strictly on RNN hidden states to carry over information, FlareDTDG can be configured to retain specific historical chunks to directly reinforce the capture of long-term cycles. However, we acknowledge that capturing high-resolution global periodicity under extreme hardware constraints remains a common challenge for all current distributed DTDG systems. Our framework provides a flexible foundation for such scenarios, and we plan to explore more adaptive importance sampling schedules as future work to further enhance periodic pattern detection.

We explicitly employ monotonic decay as the default sampling strategy to preserve the continuity of RNN states. While our pluggable scheduler technically supports non-monotonic configurations (e.g., retaining high resolution at specific lags like $t-7$), such strategies risk disrupting the sequential state updates inherent to DTDG training ($h_t = \text{RNN}(h_{t-1}, \ldots)$). Specifically, if a non-monotonic policy retains a node at $t-7$ but drops it at an intermediate step $t-3$ to save memory, the hidden state chain breaks at $t-3$, preventing the informative signal from the past from reaching the present. By contrast, our monotonic decay strategy enforces a probabilistic nested property where $V_{t-k} \subseteq V_{t-k+1}$. This design guarantees that any node sampled in the distant past possesses a continuous pathway to transmit its temporal features to the current snapshot, thereby maximizing the utilization of historical signals while avoiding the information bottlenecks caused by intermediate state loss.

Configuration Guidelines. Based on the sensitivity analysis in Section 4.6, the choice of K and α should be guided by the interplay between task dynamics and available hardware resources (GPU/CPU memory and training time). Rather than assuming larger is always better, we observe that the optimal configuration depends on the information density of the snapshots. For high-volatility tasks like *rec-amazon*, where user interests drift rapidly, a moderate full-snapshot window ($K \approx 2$–3) is often superior to a larger one, as distant history can introduce stale noise that degrades prediction accuracy. In contrast, for tasks with high structural stability like *soc-flickr*, which rely on accumulated community formation, maintaining a deeper high-fidelity context ($K \geq 4$) is necessary to avoid fragmenting the dense subgraphs required for modeling. In practice, we recommend users first determine the maximum K and α feasible within their hardware budget and then refine these values based on the specific characteristics of the target task.

Table 2: Dataset characteristics for our experiments.

Dataset	# Nodes	# Edges	Task	# GPUs
soc-flickr	2.2M	33.1M	Reg.	4
rec-amazon	2.1M	5.8M	Reg.	4
soc-youtube	3.2M	12.2M	Reg.	4
soc-bitcoin	24.6M	122.9M	Reg.	16
ogbn-papers100M	111.1M	1.6B	Cls.	16

Note: "Reg" denotes regression and "Cls." indicates classification.

Since the optimization landscape is typically continuous and convex-like across all tested datasets in Figure 8, FlareDTDG also supports a simplified greedy auto-tuning strategy to lower the barrier for deployment. Starting from a lightweight initial configuration (e.g., $K = 2, \alpha = 0.1$), the system periodically evaluates neighboring settings ($K \pm 1, \alpha \pm 0.1$) on the validation set during training. By traversing the steepest loss reduction direction in the parameter space, FlareDTDG can adaptively converge to a near-optimal configuration that maximizes accuracy while respecting the constraints of limited training resources.

4 EXPERIMENTS

4.1 Experimental Setup

Hardware Configuration. Our experimental testbed comprises four nodes, each with two Intel Xeon 6342R CPUs, 1 TB RAM, and four 48 GB NVIDIA A40 GPUs. We utilize PCIe 4.0 for intra-node and a 200 Gbps InfiniBand network for inter-node communication.
Tested Datasets. We evaluate FlareDTDG on five benchmark datasets [13, 32], with their statistics summarized in Table 2. These datasets cover both regression and classification tasks, following the preprocessing and model configurations of prior work [1, 10, 37]. For regression tasks, node features are represented by in/out degrees, and the objective is to predict degrees in the next snapshot. For classification tasks, node features are embeddings derived from paper titles and abstracts, and the objective is to predict the categories of new nodes. Experiments on the three small-scale datasets are conducted on a single node (4 GPUs), while those on the two large-scale datasets leverage all four nodes (16 GPUs).
Tested Models. We evaluate FlareDTDG on three representative DTDG models: *T-GCN* [51] integrates GCN operations into a GRU for joint spatial-temporal modeling; *MPNN-LSTM* [27] combines an MPNN for spatial aggregation with a stacked LSTM for temporal dependencies; and *EvolveGCN* [28] uses an LSTM to dynamically evolve GCN parameters, allowing filters to adapt to changing graph structures without relying on static weights.
Baseline Frameworks. We compare our FlareDTDG against several state-of-the-art DTDG training frameworks. *ESDG* [1]: A full-batch training framework for distributed systems. It leverages snapshot parallelism to balance communication costs and adopts gradient checkpointing to reduce memory consumption, enabling the training of longer snapshot sequences. *DynaGraph* [10]: A sliding-window framework with subgraph sampling. It accelerates training through cached intermediate message aggregations and applies periodic random re-partitioning to evenly redistribute training data across devices each epoch. *DynaHB* [37]: A communication-efficient framework that caches multi-hop neighbor features to avoid direct inter-GPU exchanges. It introduces a hybrid batching strategy

Table 3: Training accuracy comparison.

Dataset	Model	ESDG	DynaGraph	DynaHB	Ours
		Regression Tasks (MSE ↓)			
soc-flickr	T-GCN	0.1093	0.1101	0.6399	**0.0876**
	MPNN-LSTM	0.0937	0.0860	0.2578	**0.0543**
	EvolveGCN	0.3536	0.3477	0.8151	**0.3081**
rec-amazon	T-GCN	0.4855	0.5235	0.5073	**0.4465**
	MPNN-LSTM	0.6272	0.3082	0.3512	**0.1245**
	EvolveGCN	0.8427	0.4735	0.3224	**0.2984**
soc-youtube	T-GCN	0.1914	0.1355	0.2298	**0.1281**
	MPNN-LSTM	0.0931	0.1560	0.1394	**0.0812**
	EvolveGCN	0.3758	0.3664	0.5701	**0.3107**
soc-bitcoin	T-GCN	0.2533	0.3764	0.2354	**0.2011**
	MPNN-LSTM	0.4121	0.3623	**0.3015**	0.3038
	EvolveGCN	0.3523	0.3464	0.3441	**0.2543**
		Classification Task (Accuracy % ↑)			
ogbn-papers 100M	T-GCN	GPU OOM	53.1	CPU OOM	**57.5**
	MPNN-LSTM	GPU OOM	49.5	CPU OOM	**54.3**
	EvolveGCN	GPU OOM	55.4	CPU OOM	**58.8**

Note: For regression tasks (MSE), lower is better. For the classification task (Accuracy %), higher is better. "GPU OOM" denotes out of GPU memory, while "CPU OOM" denotes out of host memory.

and employs reinforcement learning–based scheduling to reduce sampling and data movement overhead.

Training Configuration. We standardize the training sliding window size to 8 snapshots for all methods. For baseline frameworks like ESDG that are designed for full-batch training, we set the window to encompass the entire dataset to align with its intended use of gradient checkpointing for memory savings. We use the Adam optimizer with a learning rate of 0.001. All other framework-specific hyperparameters are set to the default values reported in their respective original publications.

4.2 Overall Performance

In this section, we comprehensively evaluate FlareDTDG against three state-of-the-art training frameworks: ESDG, DynaGraph, and DynaHB. The evaluation spans three representative DTDG models and five datasets of varying scales and tasks.

Training Accuracy. We first compare the final accuracy of models trained under different frameworks. We report Mean Square Error (MSE) for regression (lower is better) and accuracy for classification (higher is better). This evaluation measures how well each framework preserves model quality while optimizing for performance.

As shown in Table 3, FlareDTDG achieves the best accuracy across nearly all configurations, with only one exception: MPNN-LSTM on `soc-bitcoin`, where FlareDTDG still yields near-best performance, trailing the top competitor by only 0.0023 MSE. On the other regression cases, FlareDTDG consistently produces the lowest MSE, reducing error by ranging from 7.4% to 59.6% compared with the best-performing baselines. For `ogbn-papers100M`, where ESDG and DynaHB encounter out-of-memory (OOM) errors, FlareDTDG successfully trains and outperforms DynaGraph, improving classification accuracy by 4.4%, 4.8%, and 3.4% when training T-GCN, MPNN-LSTM, and EvolveGCN models.

This superior accuracy stems from our *spatiotemporal hybrid batching strategy*. By retaining full recent snapshots, we provide the model with high-fidelity, crucial information, while the decayed sampling of older data maintains long-term context. This approach avoids the information loss of DynaGraph's uniform sampling and the gradient staleness in DynaHB's asynchronous synchronization, which decouples computation from the latest graph structure.

Table 4: Convergence time comparison (seconds).

Dataset	Model	ESDG	DynaGraph	DynaHB	FlareDTDG
soc-flickr	T-GCN	3,112.6	4,561.2	2,880.0	**1,819.3**
	MPNN-LSTM	3,075.4	5,629.2	4,106.6	**2,362.3**
	EvolveGCN	5,830.2	6,994.6	4,393.3	**2,847.4**
rec-amazon	T-GCN	4,419.1	3,208.0	3,473.6	**1,701.1**
	MPNN-LSTM	4,519.1	4,016.8	3,664.8	**1,685.3**
	EvolveGCN	5,008.6	4,883.8	3,794.0	**1,948.1**
soc-youtube	T-GCN	7,559.4	3,324.4	3,172.2	**1,321.7**
	MPNN-LSTM	7,599.1	8,772.7	3,420.1	**1,716.7**
	EvolveGCN	7,839.7	9,827.9	4,820.5	**2,543.0**
soc-bitcoin	T-GCN	7,537.0	6,361.1	5,672.7	**3,325.8**
	MPNN-LSTM	6,963.7	22,884.4	10,792.7	**4,346.1**
	EvolveGCN	8,390.4	17,950.1	9,808.9	**4,086.3**
ogbn-papers 100M	T-GCN	GPU OOM	2,892.1	CPU OOM	**2,068.2**
	MPNN-LSTM	GPU OOM	4,923.2	CPU OOM	**3,890.6**
	EvolveGCN	GPU OOM	3,930.4	CPU OOM	**2,709.3**

Training Efficiency. As detailed in Table 4, FlareDTDG consistently achieves the fastest convergence across all tests, outperforming the best baselines by 1.4× to 2.5×. For instance, FlareDTDG trains MPNN-LSTM on `soc-bitcoin` 2.5× faster than the next fastest framework DynaHB. These efficiency gains stem from our hybrid batching which provides higher-quality updates for faster convergence. In contrast, the convergence speed of baselines is often hampered by information loss or gradient staleness.

Memory Consumption. We measure peak GPU memory usage per device to assess framework scalability under memory constraints. As shown in Table 5, FlareDTDG demonstrates exceptional memory efficiency by achieving the lowest GPU memory consumption across all datasets. FlareDTDG reduces GPU usage by 21.0%–58.3% compared to DynaHB, without DynaHB's heavy reliance on host-memory caching. This efficiency is particularly crucial for the massive `ogbn-papers100M` dataset: FlareDTDG successfully trains models on our testbed using 40 GB of GPU memory per device, whereas ESDG fails due to GPU memory limits and DynaHB exhausts host memory with its caching mechanism. Although DynaGraph can also train on `ogbn-papers100M`, it is outperformed by FlareDTDG in both training accuracy and efficiency.

This low memory footprint stems from our recency-aware temporal decay. By discarding chunks from older snapshots, we avoid storing the entire graph history which is a major bottleneck for methods like ESDG. In addition, our fast graph reconstruction via shrinking eliminates the need for runtime node remapping and state duplication, which are common sources of memory overhead in naive sampling approaches. These design choices make FlareDTDG uniquely scalable to graphs with long temporal sequences.

Convergence Curve. We further evaluate convergence behavior by plotting test MSE against wall-clock time in Figure 6, which captures the joint effect of system throughput and gradient quality. FlareDTDG consistently demonstrates superior dynamics, achieving lower error significantly faster across all tested scenarios. For instance, on `soc-flickr` and `rec-amazon` with T-GCN, FlareDTDG converges within 60s, while competitors remain in high-error stages.

This rapid convergence results from our high-throughput and high-quality updates, enabling more effective steps per unit of time.

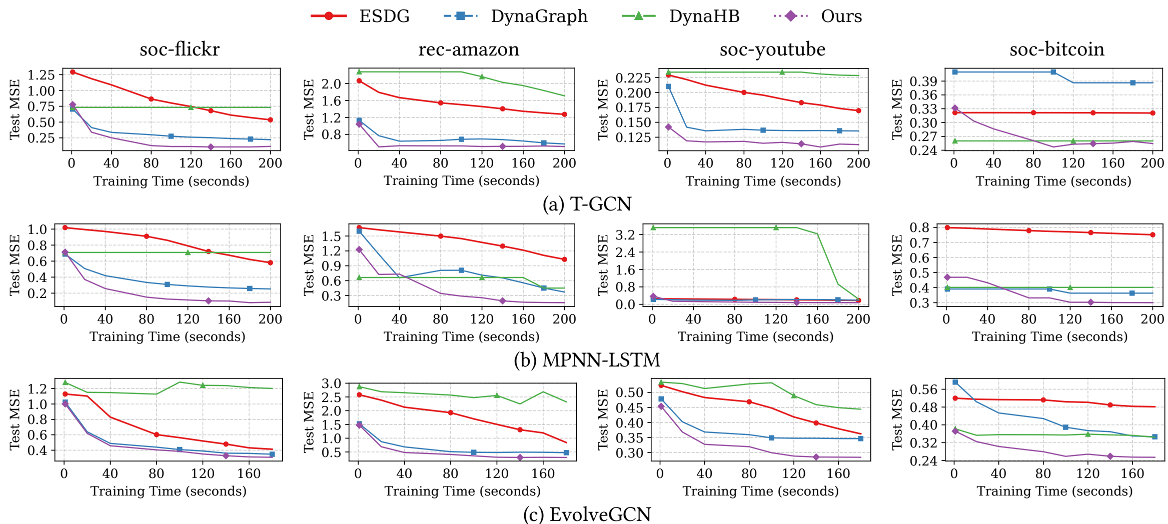

Figure 6: Convergence curves for training different DTDG models on various datasets.

Table 5: Per GPU Peak GPU memory usage comparison (GB).

Dataset	Model	ESDG	DynaGraph	DynaHB	FlareDTDG
soc-flickr	T-GCN	3.4	4.6	3.1	**2.2**
	MPNN-LSTM	3.5	4.9	3.5	**2.4**
	EvolveGCN	3.4	4.8	3.0	**2.1**
rec-amazon	T-GCN	4.7	5.8	1.8	**1.6**
	MPNN-LSTM	4.9	6.4	2.0	**1.8**
	EvolveGCN	4.5	6.0	2.5	**1.6**
soc-youtube	T-GCN	4.3	5.5	1.8	**1.3**
	MPNN-LSTM	4.1	5.0	1.6	**1.2**
	EvolveGCN	4.7	5.5	2.3	**1.2**
soc-bitcoin	T-GCN	4.1	6.3	4.0	**1.0**
	MPNN-LSTM	5.1	2.4	2.4	**1.1**
	EvolveGCN	4.5	6.7	5.3	**1.0**
ogbn-papers 100M	T-GCN	GPU OOM	41.5	CPU OOM	**38.6**
	MPNN-LSTM	GPU OOM	45.8	CPU OOM	**43.1**
	EvolveGCN	GPU OOM	38.3	CPU OOM	**37.9**

This superior efficiency stands in stark contrast to the baselines. ESDG's progress is hampered by its slow, blocking full-snapshot computations. More notably, DynaHB consistently exhibits a distinct platform period at the start of training, a phenomenon particularly visible on the `rec-amazon` and `soc-youtube` dataset. This is due to its RL scheduler's slow start, which requires a warm-up phase to explore the policy space. Our static, recency-aware strategy avoids such overhead entirely, delivering high performance from the very first step.

4.3 Ablation Studies

To quantify the contribution of each core component in FlareDTDG, we evaluate several framework variants using T-GCN, MPNN-LSTM, and EvolveGCN across three representative datasets.

Impact of Hybrid Batching with Temporal Decay. First, we analyze the effectiveness of our core strategy: Hybrid Batching with Temporal Decay. We compare our method against two variants: (1) *Full-History*, which uses all historical snapshots as full graphs (equivalent to a full-batch approach without decay), and (2) *Recent-Only*, which trains on the most recent 2 full snapshots,with the results shown in Table 6. Firstly, to verify whether the decayed historical snapshots in our hybrid batches contribute meaningfully to the model, we compare "Ours" against the "Recent-Only" baseline. We can see that, strictly truncating history results in a severe performance penalty across all tested configurations. For instance, on `rec-amazon` with MPNN-LSTM, the MSE spikes from 0.1245 (Ours) to 1.2034 (Recent-Only), representing a catastrophic loss of predictive capability. Similarly, on `soc-flickr`, the "Recent-Only" approach yields consistently higher errors across all models. This significant performance degradation confirms that the decayed temporal blocks, despite their reduced resolution, serve as a vital informational safety net by capturing long-term dependencies that are inaccessible when looking only at the immediate past.

We further evaluate the effectiveness of our decay strategy by comparing it with the "Full-History" baseline and the "Recent-Only" baseline. As detailed in Table 6, the "Full-History" approach utilizes all historical data at full resolution, typically achieving the lowest error but at the cost of prohibitive memory usage. In comparison, our method may incur a slight accuracy trade-off (e.g., a marginal MSE increase on `rec-amazon`) due to the reduction in resolution. However, critically, our method avoids the catastrophic information loss observed in the "Recent-Only" baseline, where MSE spikes significantly (e.g., from 0.0543 to 0.6867 on `soc-flickr`). This comparison demonstrates that FlareDTDG successfully strikes a pragmatic balance: it retains essential long-term dependencies through decayed sampling to prevent the model from degenerating to a short-sighted predictor, while reducing GPU memory usage by up to 70% (Table 7) and accelerating training by over 60% (Table 4). Effectively, we accept a minor deviation from the ideal full-history accuracy to enable scalability on massive graphs where the "Full-History" baseline would otherwise encounter OOM errors.

Regarding resource consumption, Table 7 shows our method reduces GPU memory usage by 67%–71% across all cases, making training on larger graphs feasible. Similarly, Table 8 shows our strategy shortens convergence time by 63%–72%, demonstrating that decaying older snapshots reduces both computational load and memory overhead. Overall, the proposed strategy provides a highly effective trade-off by drastically lowering costs while largely maintaining accuracy.

Impact of Fast Graph Reconstruction via Shrinking. We evaluate the impact of fast graph reconstruction by comparing a naive remap-per-step variant (w/o Shrinking) with our global remapping strategy (w/ Shrinking, Ours). Our approach eliminates the remapping bottleneck by performing a single global permutation of chunks per epoch, ensuring node ID continuity throughout training.

As shown in Table 9, shrinking reduces convergence time by 5.3%–25.3%, directly increasing throughput. The most significant gain (25.3%) occurs with MPNN-LSTM on `soc-flickr`. This confirms that high-frequency remapping in the naive approach is a bottleneck effectively eliminated by our design. Regarding resource usage (Table 10), our method reduces peak GPU memory by 2.5%–9.5%, saving up to 9.5% for MPNN-LSTM on `soc-flickr` by avoiding the repeated memory allocation required for per-step remapping.

Impact of Adaptive Comm-Comp Overlap Scheduling. We quantify the benefits of our Adaptive Comm-Comp Overlap Scheduling. We measure the total communication time per epoch with and without this feature enabled. Table 11 shows that our scheduler consistently reduces communication overhead, with substantial gains ranging from 6.9% (MPNN-LSTM on `soc-youtube`) to 20.4% (T-GCN on `soc-flickr`). Overlap effectiveness depends heavily on model architecture. T-GCN consistently derives the most benefit, suggesting its computational graph offers more opportunities to interleave tasks. Conversely, MPNN-LSTM's modest gains likely stem from longer, less divisible computation phases that limit overlap potential. While model architecture influences efficacy, the consistent improvements across all cases confirm that our lightweight predictor and greedy strategy effectively hide communication latency and boost end-to-end throughput.

4.4 Breakdown of Training Overhead

To analyze the system bottlenecks and quantify the cost of our strategies, we profile the **end-to-end training time** (until convergence) for the MPNN-LSTM model. Table 12 details the breakdown across three datasets. We categorize the execution into four primary components: (1) **Preprocessing**: "Chunking" represents the one-time offline spatial partitioning via METIS; (2) **Sampling**: "Shuffle" denotes the online chunk shuffling and selection; (3) **Reconstruction**: "Remap" accounts for the Global Node Remapping and Shrinking performed at each snapshot; (4) **Execution**: "Fwd.", "Bwd.", "Comm.", and "Load" represent the valid computation, communication, and data transfer.

Table 6: Accuracy ablation (MSE ↓) for hybrid batching. "Improv." denotes the MSE reduction of "Ours" relative to the "Recent-Only" baseline.

Dataset	Model	Full-History	Recent-Only	Ours	Improv.
soc-flickr	T-GCN	**0.0562**	0.1395	0.0876	-37.2%
	MPNN-LSTM	0.0597	0.6867	**0.0543**	-92.1%
	EvolveGCN	**0.2932**	0.7289	0.3081	-57.7%
rec-amazon	T-GCN	0.4792	0.5527	**0.4465**	-19.2%
	MPNN-LSTM	**0.0943**	1.2034	0.1245	-89.7%
	EvolveGCN	0.3183	0.6082	**0.2984**	-50.9%
soc-youtube	T-GCN	0.1306	0.5399	**0.1281**	-76.3%
	MPNN-LSTM	**0.0434**	0.0946	0.0812	-14.2%
	EvolveGCN	**0.2799**	0.7163	0.3107	-56.6%

Table 7: Ablation study of hybrid batching with temporal decay on per GPU memory usage (GB).

Dataset	Model	Full-History	Ours	Reduction
soc-flickr	T-GCN	7,016.60	2,264.96	−67.7%
	MPNN-LSTM	7,336.17	2,441.61	−66.7%
	EvolveGCN	6,980.35	2,054.24	−70.6%
rec-amazon	T-GCN	5,536.75	1,653.05	−70.2%
	MPNN-LSTM	5,798.32	1,810.17	−68.8%
	EvolveGCN	5,389.58	1,583.32	−70.6%
soc-youtube	T-GCN	4,142.99	1,274.00	−69.3%
	MPNN-LSTM	4,334.19	1,392.69	−67.9%
	EvolveGCN	4,012.72	1,184.03	−70.5%

Table 8: Ablation study of hybrid batching with temporal decay on convergence time (seconds).

Dataset	Model	Full-History	Ours	Reduction
soc-flickr	T-GCN	5,887.7	1,819.3	−69.1%
	MPNN-LSTM	7,475.6	2,362.3	−68.4%
	EvolveGCN	8,253.3	2,847.4	−65.5%
rec-amazon	T-GCN	5,332.4	1,701.1	−68.1%
	MPNN-LSTM	5,153.8	1,685.3	−67.3%
	EvolveGCN	5,279.4	1,948.1	−63.1%
soc-youtube	T-GCN	4,703.6	1,321.7	−71.9%
	MPNN-LSTM	6,002.4	1,716.7	−71.4%
	EvolveGCN	8,591.2	2,543.0	−70.4%

Table 9: Convergence time (seconds) of ablation study on fast graph reconstruction via shrinking.

Dataset	Model	w/o Shrinking	w/ Shrinking (Ours)	Reduction
soc-flickr	T-GCN	1,992.7	1,819.3	−8.7%
	MPNN-LSTM	3,162.4	2,362.3	−25.3%
	EvolveGCN	3,185.0	2,847.4	−10.6%
rec-amazon	T-GCN	1,796.3	1,701.1	−5.3%
	MPNN-LSTM	2,006.3	1,685.3	−16.0%
	EvolveGCN	2,065.8	1,948.1	−5.7%
soc-youtube	T-GCN	1,447.6	1,321.7	−8.7%
	MPNN-LSTM	2,239.8	1,716.7	−23.4%
	EvolveGCN	2,850.9	2,543.0	−10.8%

Table 10: GPU Memory (MB) of ablation study on fast graph reconstruction via shrinking.

Dataset	Model	w/o Shrinking	w/ Shrinking (Ours)	Reduction
soc-flickr	T-GCN	2,325.04	2,264.96	−2.6%
	MPNN-LSTM	2,698.03	2,441.61	−9.5%
	EvolveGCN	2,218.59	2,054.24	−7.4%
rec-amazon	T-GCN	1,694.51	1,653.05	−2.4%
	MPNN-LSTM	1,965.20	1,810.17	−7.9%
	EvolveGCN	1,638.32	1,583.32	−3.4%
soc-youtube	T-GCN	1,311.91	1,274.00	−2.9%
	MPNN-LSTM	1,527.21	1,392.69	−8.8%
	EvolveGCN	1,214.75	1,184.03	−2.5%

Preprocessing and Sampling Costs. A key observation is that the offline overhead is effectively amortized over the training lifecycle. Since *Chunking* is performed only once prior to training, it constitutes a negligible fraction ($\leq 1.02\%$) of the total runtime. Similarly, the *Shuffle* operation incurs virtually zero overhead ($< 0.5\%$) as it involves only index manipulation without accessing raw data.

Remapping Cost vs. Execution Efficiency. The primary system overhead stems from the *Remap* phase (11.68%–17.51%). While visible, this overhead is the result of a deliberate design choice: by

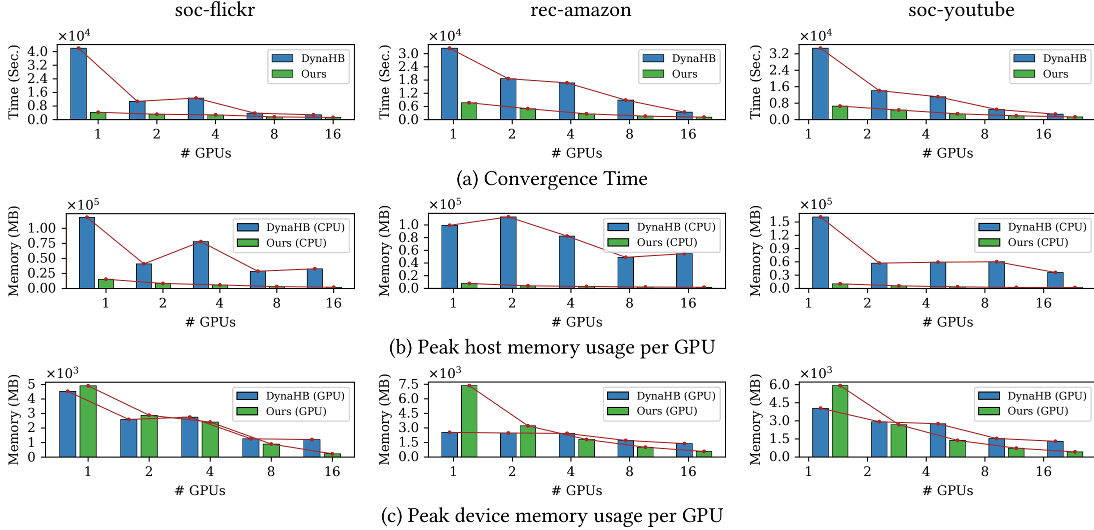

Figure 7: Performance analysis of the MPNN-LSTM model in distributed training under different numbers of GPUs.

enforcing memory contiguity via remapping, we eliminate random memory access during the computation phases. Consequently, valid computation (*Fwd.* and *Bwd.*) dominates the total runtime (averaging over 70%), confirming that FlareDTDG effectively translates system overhead into high training throughput.

4.5 Scalability Analysis

We evaluate the scalability of FlareDTDG by measuring training throughput, GPU memory, and CPU memory consumption across a cluster range of 1 to 16 GPUs. We compare against the strongest baseline, DynaHB, to highlight the system trade-offs.

Convergence Time Scalability. Figure 7 (a) illustrates the total convergence time. FlareDTDG consistently achieves the fastest training speed, outperforming DynaHB by 1.4× to 5.2× across all configurations. Regarding scaling efficiency, we observe a typical sub-linear speedup (e.g., ≈2.6× speedup when scaling from 1 to 4 GPUs). This phenomenon is explained by the breakdown of communication overhead: as the graph is partitioned into more fragments, the ratio of boundary nodes increases, leading to finer-grained but more frequent synchronization events. While DynaHB exhibits a slightly flatter scaling curve due to its communication-avoiding design, its absolute performance is strictly dominated by FlareDTDG due to our efficient hybrid batching and overlap scheduling.

Table 11: Ablation study of asynchronous overlap on non-overlapped communication time (sec/epoch).

Dataset	Model	w/o Async	w/ Async (Ours)	Reduction
soc-flickr	T-GCN	0.093	0.074	−20.4%
	MPNN-LSTM	0.042	0.036	−14.3%
	EvolveGCN	0.102	0.084	−17.6%
rec-amazon	T-GCN	0.223	0.199	−10.8%
	MPNN-LSTM	0.261	0.241	−7.7%
	EvolveGCN	0.235	0.209	−11.1%
soc-youtube	T-GCN	0.298	0.251	−15.8%
	MPNN-LSTM	0.422	0.393	−6.9%
	EvolveGCN	0.305	0.257	−10.0%

Table 12: Time breakdown of training time for MPNN-LSTM.

Dataset	Chunking	Shuffle	Remap	Fwd.	Bwd.	Comm.	Load
soc-flickr	1.02%	0.42%	17.51%	27.06%	46.00%	1.27%	6.72%
rec-amazon	0.46%	0.28%	13.73%	21.91%	50.47%	9.37%	3.78%
soc-youtube	0.61%	0.35%	11.68%	23.00%	46.64%	14.05%	3.67%

Memory Scalability and Bottlenecks. Figure 7 (c) demonstrates the per-GPU memory consumption. FlareDTDG exhibits excellent **near-linear memory scaling**. As the cluster size doubles, the memory footprint per device is nearly halved (e.g., dropping from 7.5GB on 1 GPU to 2.3GB on 4 GPUs for soc-flickr). This memory reduction confirms our framework's ability to effectively distribute the storage of graph topology and historical states. In contrast, DynaHB's GPU memory usage remains relatively constant, as it relies on sampling batches that do not shrink with the cluster size.

CPU Memory Efficiency. Host memory consumption is a critical but often overlooked limitation. As shown in Figure 7 (b), DynaHB's vertex-caching strategy replicates node features across partitions to avoid communication, triggering a CPU memory explosion (>100GB) even for small datasets. In contrast, FlareDTDG maintains a minimal, constant CPU footprint regardless of cluster size. This result highlights FlareDTDG's design advantage where it accepts managed communication costs to ensure storage scalability, providing the only feasible solution for training massive dynamic graphs on memory-constrained commodity clusters.

4.6 Sensitivity Analysis

To understand the impact of FlareDTDG's key hyperparameters, we conduct a sensitivity analysis using the MPNN-LSTM model as a representative case. We vary the two main parameters of our Hybrid Batching strategy: (1) the *Number of Full Snapshots* (K), which determines the amount of recent, high-fidelity information, and (2) the *Temporal Resolution Ratio* (α), which controls the sampling rate of older, historical data as defined in Section 3.2. We measure the resulting model accuracy (MSE), peak GPU memory, and training convergence time. The results are visualized in Figure 8.

Impact on System Performance. The heatmaps clearly indicate that system performance is predominantly influenced by the number of full snapshots. As seen across all datasets in Figure 8, increasing the number of full snapshots (moving right along the x-axis) leads to a near-linear increase in GPU memory consumption and a corresponding increase in convergence time. This trend is expected,

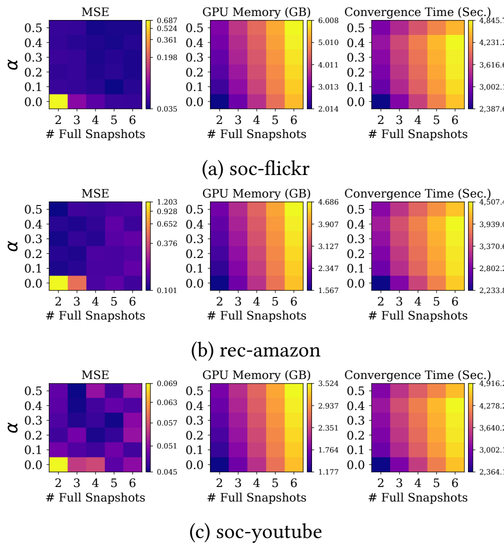

Figure 8: Parameter sensitivity analysis for the MPNN-LSTM model on diffenent datasets. Each column shows the impact of varying the number of full snapshots and the temporal resolution ratio (α) on MSE (left), GPU Memory (middle), and Convergence Time (right).

as each full snapshot adds a significant data loading and computational load. In contrast, the temporal resolution ratio, α, has a much subtler impact on these metrics, as the decayed temporal blocks represent a smaller fraction of the total batch data.

Impact on Model Accuracy. The accuracy (MSE) landscape reveals distinct patterns that correlate with the temporal dynamics of each dataset, validating our configuration guidelines. Across all datasets, a consistent high-error zone appears at the bottom-left corner (K is small and $\alpha \approx 0$), confirming that strictly truncating history (Recent-Only) starves the model of necessary long-term context and leads to poor convergence. The location of the optimal "sweet spot" varies significantly by data type: for structural datasets like **soc-flickr**, the region shifts toward higher K and α to preserve dense, accumulated community structures; conversely, for transactional data like **rec-amazon**, the best accuracy is achieved with fewer snapshots ($K \approx 2$–3) as older interactions rapidly lose predictive value due to interest drift. Hybrid datasets like **soc-youtube** find a middle ground ($K \approx 3$–$5, \alpha \approx 0.3$), balancing immediate activity bursts with long-term user profiles.

Crucially, the error landscape is generally continuous and convex-like rather than chaotic. The transition from high-error regions to the optimal configuration is smooth. This observation empirically supports the feasibility of the greedy auto-tuning strategy proposed in Section 3.5, as a simple local search can effectively traverse towards the global optimum without getting trapped in local minima. In summary, K serves as the primary lever for system performance, while α allows for fine-grained accuracy-efficiency balancing.

5 RELATED WORKS

Single-GPU DTDG Systems. Several systems optimize DTDG training on a single device by focusing on resource utilization and data movement. PiPAD [40] introduces an efficient data layout with asynchronous loading to hide I/O latency. BLAD [8] overlaps memory-intensive message passing with compute-intensive transformations to improve GPU utilization. STGraph [44] provides a flexible framework with specialized operators to accelerate spatio-temporal GNNs. While efficient, these systems are fundamentally constrained by single-GPU memory and compute capacity, making them unsuitable for large-scale graphs.

Distributed DTDG Systems. To scale beyond a single GPU, distributed frameworks partition the workload using different strategies. ESDG [1] parallelizes training across the time dimension by assigning snapshots to different GPUs but requires each worker to store at least one full snapshot. To avoid this storage requirement, node-parallel approaches partition the graph itself. DynaGraph [10] partitions nodes and employs neighbor sampling, but incurs high overhead from frequent repartitioning and sampling. Other methods reduce GPU memory pressure by offloading data. DynaHB [37] caches mutli-hop neighbor features in CPU memory, trading inter-GPU communication for CPU memory bandwidth limitations. Systems like ADGNN [5], PipeGCN [39], and Sancus [30] provide effective strategies for processing a single, extremely large snapshot using techniques like adaptive data placement and pipeline parallelism. While their methods could be applied to individual snapshots within a DTDG, they are not inherently designed to manage the temporal sequence and its associated challenges, such as inter-snapshot dependencies and exploiting temporal locality.

Scalable DTDG Architectures. Beyond system-level optimizations, some works like DyGFormer [49] redesign model architectures for scalability, replacing conventional structures with Transformers that operate on sampled subgraph sequences. However, these sampling-based approaches can be computationally expensive and often risk losing critical global structural information. Our work instead focuses on system-level optimizations for the prevalent GNN+RNN paradigm, achieving high scalability by exploiting the inherent temporal properties of dynamic graphs without altering the underlying model architecture.

6 CONCLUSION

This paper presents FlareDTDG, a distributed framework exploiting temporal recency for efficient DTDG training. By integrating hybrid batching, fast reconstruction, and adaptive overlap scheduling, FlareDTDG reduces overhead while preserving long-term dependencies. Experiments show up to 2.5× speedup and 85% lower GPU memory usage, scaling to massive graphs with 100M nodes per snapshot. Overall, FlareDTDG offers a practical, scalable solution for large-scale distributed dynamic graph learning.

ACKNOWLEDGMENTS

This work was supported by the Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study (Grant No. SN-ZJU-SIAS-001), the Hangzhou Joint Fund of the Zhejiang Provincial Natural Science Foundation of China (Grant No. LHZSD24-F020001), the Young Scientists Fund of the National Natural Science Foundation of China (Grant No. 62502438), the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China, and the Open Research Fund of the State Key Laboratory of Blockchain and Data Security, Zhejiang University.

REFERENCES

- [1] Venkatesan T. Chakaravarthy, Shivmaran S. Pandian, Saurabh Raje, Yogish Sabharwal, Toyotaro Suzumura, and Shashanka Ubaru. 2021. Efficient scaling of dynamic graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 77, 15 pages. https://doi.org/10.1145/3458817.3480858
- [2] Fahao Chen, Peng Li, and Celimuge Wu. 2023. DGC: Training Dynamic Graphs with Spatio-Temporal Non-Uniformity using Graph Partitioning by Chunks. *Proc. ACM Manag. Data* 1, 4, Article 237 (Dec. 2023), 25 pages. https://doi.org/10.1145/3626724
- [3] Kaixuan Chen, Wei Luo, Shunyu Liu, Yaoquan Wei, Yihe Zhou, Yunpeng Qing, Quan Zhang, Yong Wang, Jie Song, and Mingli Song. 2025. Powerformer: A section-adaptive transformer for power flow adjustment. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 1*. 2204–2215.
- [4] Kaixuan Chen, Jie Song, Shunyu Liu, Na Yu, Zunlei Feng, Gengshi Han, and Mingli Song. 2023. Distribution knowledge embedding for graph pooling. *IEEE Transactions on Knowledge and Data Engineering* 35, 8 (2023), 7898–7908.
- [5] Yushun Dong, Kaize Ding, Brian Jalaian, Shuiwang Ji, and Jundong Li. 2021. AdaGNN: Graph Neural Networks with Adaptive Frequency Response Filter. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 392–401.
- [6] Zhengzhao Feng, Rui Wang, Longjiao Zhang, Tongya Zheng, Ziqi Huang, and Mingli Song. 2025. Efficient Dynamic Graph Learning with Refined Batch Parallel Training. In *The 34th International Joint Conference on Artificial Intelligence*.
- [7] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.
- [8] Kaihua Fu, Quan Chen, Yuzhuo Yang, Jiuchen Shi, Chao Li, and Minyi Guo. 2023. BLAD: Adaptive Load Balanced Scheduling and Operator Overlap Pipeline For Accelerating The Dynamic GNN Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 37, 13 pages. https://doi.org/10.1145/3581784.3607040
- [9] Joao Gama, Indre Zliobaite, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A survey on concept drift adaptation. *ACM Comput. Surv.* 46, 4, Article 44 (March 2014), 37 pages. https://doi.org/10.1145/2523813
- [10] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. 2022. DynaGraph: dynamic graph neural networks at scale. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Philadelphia, Pennsylvania) *(GRADES-NDA '22)*. Association for Computing Machinery, New York, NY, USA, Article 6, 10 pages. https://doi.org/10.1145/3534540.3534691
- [11] Shengnan Guo, Youfang Lin, Ning Feng, Chao Song, and Huaiyu Wan. 2019. Attention based spatial-temporal graph convolutional networks for traffic flow forecasting. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence* (Honolulu, Hawaii, USA) *(AAAI'19/IAAI'19/EAAI'19)*. AAAI Press, Article 114, 8 pages. https://doi.org/10.1609/aaai.v33i01.3301922
- [12] Sepp Hochreiter and Jurgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
- [13] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687* (2020).
- [14] Wenjie Huang, Tongya Zheng, Rui Wang, Tongtian Zhu, Bingde Hu, Shuibing He, Mingli Song, Xinyu Wang, Sai Wu, and Chun Chen. 2025. Efficient Distributed Graph Neural Network Training with Source Chunking and Moving Aggregation. *IEEE Transactions on Knowledge and Data Engineering* (2025).
- [15] Ziqi Huang, Tongya Zheng, Rui Wang, Longjiao Zhang, Wenjie Huang, and Xinyu Wang. 2025. Quick Sense Temporal Graph Transformer with Effective Representation Augmentation. In *2025 International Joint Conference on Neural Networks (IJCNN)*.
- [16] Wei Jin, Yao Ma, Yiqi Wang, Xiaorui Liu, Jiliang Tang, Yukuo Cen, Jiezhong Qiu, Jie Tang, Chuan Shi, Yanfang Ye, Jiawei Zhang, and Philip S. Yu. 2021. Graph Representation Learning: Foundations, Methods, Applications and Systems. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (Virtual Event, Singapore) *(KDD '21)*. Association for Computing Machinery, New York, NY, USA, 4044–4045. https://doi.org/10.1145/3447548.3470824
- [17] Wei Jin, Yao Ma, Yiqi Wang, Xiaorui Liu, Jiliang Tang, Yukuo Cen, Jiezhong Qiu, Jie Tang, Chuan Shi, Yanfang Ye, Jiawei Zhang, and Philip S. Yu. 2021. Graph Representation Learning: Foundations, Methods, Applications and Systems. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (Virtual Event, Singapore) *(KDD '21)*. Association for Computing Machinery, New York, NY, USA, 4044–4045. https://doi.org/10.1145/3447548.3470824
- [18] George Karypis and Vipin Kumar. 1998. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices.*
- [19] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. 2020. Representation learning for dynamic graphs: a survey. *J. Mach. Learn. Res.* 21, 1, Article 70 (Jan. 2020), 73 pages.
- [20] Yehuda Koren. 2010. Collaborative filtering with temporal dynamics. *Commun. ACM* 53, 4 (April 2010), 89–97. https://doi.org/10.1145/1721654.1721677
- [21] 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* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 1269–1278. https://doi.org/10.1145/3292500.3330895
- [22] Brian T. Lewis. 2003. *Coroutine*. John Wiley and Sons Ltd., GBR, 465–466.
- [23] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. 2018. Diffusion Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting. In *International Conference on Learning Representations (ICLR '18)*.
- [24] Hanwen Liu, Longjiao Zhang, Rui Wang, Tongya Zheng, Sai Wu, Chang Yao, and Mingli Song. 2025. SALoM: Structure Aware Temporal Graph Networks with Long-Short Memory Updater. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- [25] Yao Ma, Ziyi Guo, Zhaocun Ren, Jiliang Tang, and Dawei Yin. 2020. Streaming Graph Neural Networks. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval* (Virtual Event, China) *(SIGIR '20)*. Association for Computing Machinery, New York, NY, USA, 719–728. https://doi.org/10.1145/3397271.3401092
- [26] 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* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 969–976. https://doi.org/10.1145/3184558.3191526
- [27] George Panagopoulos, Giannis Nikolentzos, and Michalis Vazirgiannis. 2021. Transfer Graph Neural Networks for Pandemic Forecasting. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*.
- [28] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2020. EvolveGCN: Evolving Graph Convolutional Networks for Dynamic Graphs. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 5363–5370. https://doi.org/10.1609/AAAI.V34I04.5984
- [29] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).
- [30] 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. *Proc. VLDB Endow.* 15, 9 (May 2022), 1937–1950. https://doi.org/10.14778/3538598.3538614
- [31] Farimah Poursafaei, Shenyang Huang, Kellin Pelrine, and Reihaneh Rabbany. 2022. Towards Better Evaluation for Dynamic Link Prediction. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 32928–32941. https://proceedings.neurips.cc/paper_files/paper/2022/file/d49042a5d49818711c401d34172f9900-Paper-Datasets_and_Benchmarks.pdf
- [32] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. http://networkrepository.com
- [33] Benedek Rozemberczki, Paul Scherer, Yixuan He, George Panagopoulos, Alexander Riedel, Maria Astefanoaei, Oliver Kiss, Ferenc Beres, Guzman Lopez, Nicolas Collignon, and Rik Sarkar. 2021. PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*. 4564–4573.
- [34] 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* (Houston, TX, USA) *(WSDM '20)*. Association for Computing Machinery, New York, NY, USA, 519–527. https://doi.org/10.1145/3336191.3371845
- [35] 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. https://doi.org/10.1109/ACCESS.2021.3082932
- [36] Weiping Song, Zhiping Xiao, Yifan Wang, Laurent Charlin, Ming Zhang, and Jian Tang. 2019. Session-Based Social Recommendation via Dynamic Graph Attention Networks. In *Proceedings of the Twelfth ACM International Conference on Web*

Search and Data Mining (Melbourne VIC, Australia) *(WSDM '19)*. Association for Computing Machinery, New York, NY, USA, 555–563. https://doi.org/10.1145/3289600.3290989

[37] Zhen Song, Yu Gu, Qing Sun, Tianyi Li, Yanfeng Zhang, Yushuai Li, Christian S. Jensen, and Ge Yu. 2024. DynaHB: A Communication-Avoiding Asynchronous Distributed Framework with Hybrid Batches for Dynamic GNN Training. *Proc. VLDB Endow.* 17, 11 (July 2024), 3388–3401. https://doi.org/10.14778/3681954.3682008

[38] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. Dyrep: Learning Representations over Dynamic Graphs. *International Conference on Learning Representations (ICLR)* (2019). https://openreview.net/references/pdf?id=Hy8cqH0rE

[39] 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)*.

[40] Chunyang Wang, Desen Sun, and Yuebin Bai. 2023. PiPAD: Pipelined and Parallel Dynamic GNN Training on GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 405–418. https://doi.org/10.1145/3572848.3577487

[41] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[42] Yu Wang, Junshu Dai, Yuchen Ying, Hanyang Yuan, Zunlei Feng, Tongya Zheng, and Mingli Song. 2026. Adaptive Location Hierarchy Learning for Long-Tailed Mobility Prediction. In *Proceedings of the ACM on Web Conference 2026*.

[43] Shu Wu, Yuyuan Tang, Yanqiao Zhu, Liang Wang, Xing Xie, and Tieniu Tan. 2019. Session-based recommendation with graph neural networks. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence* (Honolulu, Hawaii, USA) *(AAAI'19/IAAI'19/EAAI'19)*. AAAI Press, Article 43, 8 pages. https://doi.org/10.1609/aaai.v33i01.3301346

[44] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: vertex-centric programming for graph neural networks. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 359–375. https://doi.org/10.1145/3447786.3456247

[45] Zonghan Wu, Shirui Pan, Guodong Long, Jing Jiang, Xiaojun Chang, and Chengqi Zhang. 2020. Connecting the Dots: Multivariate Time Series Forecasting with Graph Neural Networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 753–763. https://doi.org/10.1145/3394486.3403118

[46] Chengfeng Xu, Pengpeng Zhao, Yanchi Liu, Victor S. Sheng, Jiajie Xu, Fuzhen Zhuang, Junhua Fang, and Xiaofang Zhou. 2019. Graph contextualized self-attention network for session-based recommendation. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence* (Macao, China) *(IJCAI'19)*. AAAI Press, 3940–3946.

[47] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive representation learning on temporal graphs. In *International Conference on Learning Representations (ICLR)*.

[48] Bing Yu, Haoteng Yin, and Zhanxing Zhu. 2018. Spatio-temporal graph convolutional networks: a deep learning framework for traffic forecasting. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (Stockholm, Sweden) *(IJCAI'18)*. AAAI Press, 3634–3640.

[49] Le Yu, Leilei Sun, Bowen Du, and Weifeng Lv. 2023. Towards Better Dynamic Graph Learning: New Architecture and Unified Library. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 67686–67700. https://proceedings.neurips.cc/paper_files/paper/2023/file/d611019afba70d547bd595e8a4158f55-Paper-Conference.pdf

[50] Longjiao Zhang, Rui Wang, Tongya Zheng, Ziqi Huang, Wenjie Huang, Xinyu Wang, Can Wang, Mingli Song, Sai Wu, and Shuibing He. 2025. Effective and Efficient Distributed Temporal Graph Learning through Hotspot Memory Sharing. *Proceedings of the VLDB Endowment* 18, 9 (2025), 3093–3105.

[51] Ling Zhao, Yujiao Song, Chao Zhang, Yu Liu, Pu Wang, Tao Lin, Min Deng, and Haifeng Li. 2020. T-GCN: A Temporal Graph Convolutional Network for Traffic Prediction. *IEEE Transactions on Intelligent Transportation Systems* 21, 9 (2020), 3848–3858. https://doi.org/10.1109/TITS.2019.2935152

[52] Tongya Zheng, Xinchao Wang, Zunlei Feng, Jie Song, Yunzhi Hao, Mingli Song, Xingen Wang, Xinyu Wang, and Chun Chen. 2023. Temporal aggregation and propagation graph neural networks for dynamic representation. *IEEE Transactions on Knowledge and Data Engineering* 35, 10 (2023), 10151–10165.