Redundancy-free and load-balanced TGNN training with hierarchical pipeline parallelism

Journal:	*Transactions on Parallel and Distributed Systems*
Manuscript ID	TPDS-2023-08-0498
Manuscript Type:	Regular
Keywords:	Distributed training, dynamic GNN, redundancy-free, communication balance, pipeline parallelism

Redundancy-free and load-balanced TGNN training with hierarchical pipeline parallelism

Yaqi Xia, Zheng Zhang, Donglin Yang, Chuang Hu, Xiaobo Zhou, Hongyang Chen and Dazhao Cheng

Abstract—Recently, Temporal Graph Neural Networks (TGNNs), as an extension of Graph Neural Networks, have demonstrated remarkable effectiveness in handling dynamic graph data. Distributed TGNN training requires efficiently tackling temporal dependency, which often leads to excessive cross-device communication that generates significant redundant data. However, existing systems are unable to remove the redundancy in data reuse and transfer, and suffer from severe communication overhead in a distributed setting. This work introduces Sven, a co-designed algorithm-system library aimed at accelerating TGNN training on a multi-GPU platform. Exploiting dependency patterns of TGNN models, we develop a redundancy-free graph organization to mitigate redundant data transfer. Additionally, we investigate communication imbalance issues among devices and formulate the graph partitioning problem as minimizing the maximum communication balance cost, which is proved to be an NP-hard problem. We propose an approximation algorithm called Re-FlexBiCut to tackle this problem. Furthermore, we incorporate prefetching, adaptive micro-batch pipelining, and asynchronous pipelining to present a hierarchical pipelining mechanism that mitigates the communication overhead. Sven represents the first comprehensive optimization solution for scaling memory-based TGNN training. Through extensive experiments conducted on a 64-GPU cluster, Sven demonstrates impressive speedup, ranging from 1.9x to 3.5x, compared to state-of-the-art approaches. Additionally, Sven achieves up to 5.26x higher communication efficiency and reduces communication imbalance by up to 59.2%.

Index Terms—Distributed training, dynamic GNN, redundancy-free, communication balance, pipeline parallelism

1 Introduction

Graphs, pervasive in various domains [1], serve as efficient representations for encoding real-world objects into relational data structures. In recent years, Graph Neural Networks (GNNs) have gained increasing interest for processing graph data [2], [3], encompassing tasks such as node classification [4], link prediction [5], and graph classification [6]. While significant progress has been made in GNNs, such as GCN [2] and GraphSAGE [3], these methods primarily focus on static graphs with fixed nodes and edges. In many real-world applications, however, graphs are dynamic and constantly evolve, providing additional temporal information. Recently proposed Temporal Graph Neural Networks (TGNNs) such as TGN [7], APAN [8], and JODIE [9] address this by learning both temporal and topological relationships simultaneously, integrating the ever-changing nature into the embedding information. As a result, TGNNs have demonstrated superior performance over static GNNs[10], [11].

Handling rich information in dynamic graphs asks for massive parallel and distributed computation in processing TGNNs [10]. To capture temporal dependencies, many TGNNs, known as memory-based TGNNs, employ a module called *node memory and message*, which summarizes the historical behavior of each vertex [7], [8], [9]. However, scaling TGNN training introduces two primary performance issues. Firstly, TGNN requires the latest temporal dependencies based on the vertices of all interactions, resulting in abundant redundant dependency data. Secondly, in a large-scale GPU cluster, temporal dependencies and model parameters are distributed across devices, necessitating collective operations such as *all-to-all* or *all-gather* for dispatching and aggregating dependencies, along with an *all-reduce* for synchronizing gradients of model parameters. The communication overhead significantly increases time consumption and limits the performance of TGNN training. Measurements indicate that the time proportion of communication can exceed 70% when training the JODIE model [9]. Thus, these performance bottlenecks motivate us to reduce communication volume by eliminating redundant data and mitigating communication overhead through hierarchical pipeline parallelism.

Currently, well-known GNN systems like PyG [12], and AGL [13] offer efficient and flexible operators attributed to static GNN training. Unfortunately, these frameworks have limited or no support for dynamic graphs. Several frameworks have been designed to provide efficient training primitives for dynamic graphs. DGL [14] only offers some rudimentary interfaces to support dynamic graph training; however, these interfaces are inefficient and lack compatibility with distributed environments. TGL [10] is a framework for large-scale offline TGNN training, supporting various TGNN variants training on one-node multi-GPUs. Nevertheless, TGL suffers from significant communication overhead due to extensive host-to-device memory transfers. ESDG [15] proposes a graph difference-based algorithm to reduce communication traffic and extend TGNN training to multi-node multi-GPU systems. However, ESDG only supports training with Discrete-time Dynamic Graphs (DTDGs), also known as snapshot-based graphs. As pointed out by TGN [7], Continuous-time Dynamic Graphs (CTDGs) offer the most general formulation compared to other types. Therefore, the schemes proposed in ESDG cannot be extended to general TGNN model training.

Very recently, dependency-cached (DepCache) and dependency-communicated (DepComm) mechanisms [16]

have been developed to maintain dependencies for distributed GNN training. However, both DepCache and DepComm mechanisms are found to be inefficient for TGNN training due to the heavy cross-device communication overhead.

We present Sven, an algorithm and system co-designed framework for high-performance distributed TGNN training on multi-node multi-GPU systems. Specifically, Sven is designed for TGNN training on CTDGs. We provide insight into the potential improvement in redundant dependencies and the margin between the time consumption of communication and computation. Instead of resolving these two challenges separately, we address them from a holistic perspective and in a collaborative manner.

We analyze the TGNN model behavior of handling temporal dependencies, which dispatches dependencies at the current state for computation and aggregates the latest dependencies after computation. Those models result in a tremendous volume of redundant data at the dispatch and aggregation stages because one vertex may interconnect with other vertices multiple times. We measure the redundancy ratio on various dynamic graph datasets and effectively exploit the potential communication reduction of temporal dependencies by extracting the redundancy-free graph from the original graph.

Furthermore, we augment DepCache and DepComm mechanisms to maintain temporal dependencies for distributed TGNN training. We propose to reduce the communication volume by removing data redundancy, which mainly comes from the temporal dependency communication, including all-gather operation for DepCache and all-to-all operation for DepComm. To further mitigate the impact of communication overhead, we propose an adaptive micro-batch pipelining approach to reduce the system overhead incurred by the all-to-all operation and develop an asynchronous pipelining method to hide the all-reduce communication.

Additionally, our investigation reveals that existing graph partition approaches can result in imbalanced communication across devices, significantly impacting the performance of distributed TGNN training. Addressing this communication imbalance requires a more rigorous problem formulation and a tailored approximation algorithm designed to handle communication workloads effectively. To tackle this challenge, we introduce Re-FlexBiCut, a novel approach that approximates balanced minimum cuts by constructing source/sink graphs for each partition and enables the reassignment of vertices between partitions to enhance balance.

A preliminary version of this paper appeared in [17]. Building upon the findings from that work, this manuscript expands our approach to offer a holistic and robust solution, ensuring communication-balanced partitioning across devices. Specifically, we make the following new contributions:

- We identify the imbalances in communication load across devices as a critical challenge and formulate the partitioning objective to minimize the maximum communication cost. We prove this optimization problem is NP-hard and develop a novel approximation algorithm called Re-FlexBiCut to tackle this problem in polynomial time.
- We integrate the Re-FlexBiCut partition algorithm into Sven and conduct extensive experiments, comparing its performance with other existing methods under diverse settings. The results demonstrate the superiority of Re-FlexBiCut over baseline methods consistently, achieving communication cost reductions of up to 59.2% and speedup improvements of up to 1.41x.
- We implement a mapping mechanism that maps global vertex IDs to partition-local IDs, enabling the proposed vertex-level graph partitioning. Our robust dual-hash mapping strategy effectively handles irregular graph partitioning, ensuring seamless mapping of vertices to their respective dependency data locations across devices

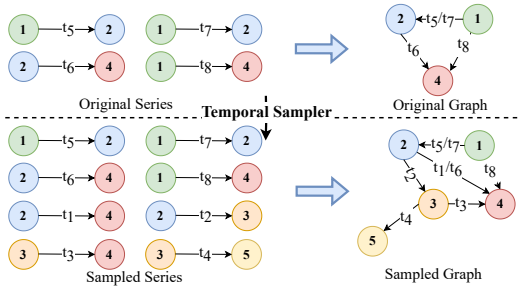


Fig. 1: The dynamic graph sampling process.

Evalution on a 16-node 64-GPU HPC system demonstrate that Sven achieves up to 59.2% communication balance improvement and 5.26x communication volume reduction. Compared to state-of-art methods, Sven obtain 1.8x-3.5x speedup, surpassing DepComm by 1.8x, DepCache by 1.9x, and sTGL by 3.5x. Furthermore, Sven exhibits excellent scalability, achieving up to 40x speedup on the system with 64 GPUs.

In the rest of this paper, Section 2 presents the background of TGNN training and motivation studies. Sections 3 and 4 describe the design and implementation of Sven, respectively. Section 5 presents the experimental results. Section 6 discusses the related work. Section 7 concludes the paper.

2 Background and Motivation

2.1 Temporal Graph Neural Network

2.1.1 *Dynamic graph and Tenporal GNN*

Recently, two representative models are proposed to describe the dynamic graphs, i.e., DTDGs and CTDGs. DTDGs consist of a series of static graphs with intercepted time intervals, while CTDGs capture continuous dynamics of temporal graph data. Since CTDGs are more general and flexible, this paper focuses on CTDGs. Temporal GNNs have focused on two aspects of dynamicity, the graph structure, and the temporal dependencies. With these features, dynamic GNNs are able to combine techniques for structural information encoding with techniques for temporal information encoding.

2.1.2 *Training process of dynamic GNN*

In general, the training process of TGNN is composed of two parts, sampler and trainer.

Sampler. Dynamic graph sampling is more complex than static graph sampling because it takes into account the timestamps of the neighbors. The temporal sampler needs to prob-

Algorithm 1: Training process of TGNN

input : original graph $G = \xi(t_s, t_e) = \{(v_i(t), v_j(t)) \| t \in (t_s, t_e)\}$, sampled graph $G' = \xi(t'_s, t'_e) = \{(v_i(t), v_j(t)) \| t \in (t'_s, t'_e)\}$, edge features $\{\boldsymbol{e} | e \in G'\}$, node features $\{\boldsymbol{v} | v \in G'\}$, node memory $\boldsymbol{S} = \{\boldsymbol{s}_i(t^-) | v_i \in G'\}$, model parameters $\boldsymbol{W}$

output: updated parameters $\boldsymbol{W}$

1 **for** $v_i(t) \in G' = \xi(t'_s, t'_e), t \leftarrow t'_s$ **to** t'_e **do**
2 $\boldsymbol{s}_i(t) = mem(\boldsymbol{m}_i(t^-), \boldsymbol{s}_i(t^-))$
3 $\boldsymbol{m}_i(t) = msg(\boldsymbol{s}_i(t), \boldsymbol{s}_j(t), t, \boldsymbol{e}_{i,j}(t))$
4 $\boldsymbol{z}_i(t) = emb(i, t) = emb(\boldsymbol{s}_i(t), \boldsymbol{s}_j(t), t, \boldsymbol{e}_{i,j}(t), \boldsymbol{v}_i, \boldsymbol{v}_j)$
5 **end**
6 **for** $v_i(t) \in G = \xi(t_s, t_e) = \{(v_i(t), v_j(t))\}, t \leftarrow t_s$ **to** t_e **do**
7 **for** $t_1, \ldots, t_b \leq t$ **do**
8 $\boldsymbol{s}_i(t) \leftarrow agg(\boldsymbol{s}_i(t_1), \ldots, \boldsymbol{s}_i(t_b))$
9 $\boldsymbol{m}_i(t) \leftarrow agg(\boldsymbol{m}_i(t_1), \ldots, \boldsymbol{s}_i(t_b))$
10 **end**
11 **end**
12 compute the loss according to specific task

abilistically identify the candidate edges from all past neighbors. As illustrated in Figure 1, the sampled series/sampled graph contains both the original series/graph and interactions that occurred in the past with its neighbors.

TABLE 1: Notations.

Notation	Description
v_i	vertex i
$\boldsymbol{v}_i, \boldsymbol{e}_{ij}$	node feature of vertex i and edge feature of edge ij
$\boldsymbol{s}_i(t), \boldsymbol{m}_i(t)$	node memory and message of vertex i at time t
t_v^-	time when $\boldsymbol{s}_v$ is updated
$\xi(t_s, t_e)$	all time series between time t_s and time t_e

Trainer. There is an essential mechanism between static graph training and dynamic graph training, i.e., **N**ode memory and **M**essage (N&M). N&M summarizes the history of the vertices in the past, which provides enough information to generate the dynamic node embedding at the current state. Algorithm 1 describes the training process of one iteration for TGNN training. Table 1 summarizes important notations. In the mini-batch training, given an original graph G, sampled graph G', its corresponding features, and N&M, the key objective of the model is to encode each vertex into an embedding and utilize it for the downstream tasks. First, a *mem* function is applied to refresh the behavior of vertices by summarizing the message and previous node memory (Line 2), which is denoted as the memory-update module. Afterward, the model launches a *msg* function to combine the information of the latest interacting neighbors of the vertices to update its message, including timestamps, node memory, and edge features (Line 3). After that, the temporal GNN module utilizes *emb* function to project feature information and memory information that is involved in the interaction events into the embedding space (Line 4). Then, TGNN updates the N&M with the result from the computation with an aggregation function *agg* (Lines 8 and 9). TGN first proposes two efficient aggregation functions: *most recent message*, which keeps only the most recent message for a given node, and *mean message*, which averages all messages for a given node. TGL demonstrates that there is no significant difference between these two methods, thus we follow TGL's approach and adopt the most recent message policy in this paper because of higher efficiency.

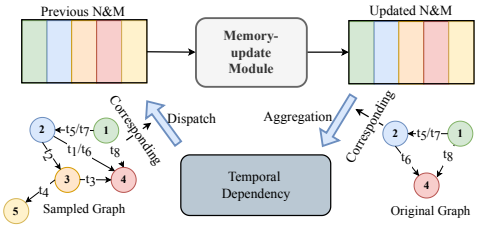


Fig. 2: Workflow of the dispatch and aggregation.

2.2 Redundancies in Temporal Dependencies

During the training process, TGNN performs two essential steps: dispatch and aggregation. As illustrated in Figure 2, during the dispatch phase, TGNN relies on the latest N&Ms in the sampled graph to calculate the updated N&Ms. It's worth noting that for one mini-batch, the N&Ms of the same vertex can be reused multiple times. In the aggregation phase, TGNN combines all events related to the same vertex from the original graph. The updated message's outcome, as indicated in line 3 of Algorithm 1, is influenced not only by the most recent update for the source vertex but also by the state of the target vertex when handling multiple events of the same vertex.

In Figure 1, there are three interactions involving vertex 1 in the sampled series $((1-2, t_5), (1-2, t7), (1-4, t_8))$. However, TGNN requires three identical N&Ms for vertex 1, which can be reused. During the aggregation phase, among all interactions involving vertex 1, only the latest one (i.e., $(1-4, t_8)$) is retained for updating the previous N&M. However, other interactions related to vertex 1 are not necessarily discarded, as they may be needed by other adjacent vertices. For example, while vertex 1 discards $(1-2, t_7)$, it is still required by vertex 2. Following the above principle, N&M data continues to evolve during training. Therefore, there exist vast redundant data in the process of dispatching and aggregating the N&M dependencies. Specially, we define the redundancy rate of the former (i.e. dispatch) and latter (i.e. aggregation) as η_1 and η_2 respectively.

$$\eta_1 = \frac{N_1^r}{N_1}, \quad \eta_2 = \frac{N_2^r}{N_2} \tag{1}$$

in which N_1 and N_2 are the numbers of sampled and original nodes respectively, and N_1^r and N_2^r are the number of duplicated nodes of sampled and original series respectively.

Figure 3 illustrates the detailed redundancy rate under various batch sizes and datasets. We can observe that the proportion of redundant nodes is quite large, i.e. more than 80% for most cases. Especially, the ratio η_1 is close to 100%, when the batch size for LASTFM is increased to 2000. With the increase in batch size, the redundancy rate also increases. The observed data redundancies result in poor training system performance. Therefore, we are motivated to design and develop a redundancy-free strategy tailored for N&M dispatch and aggregation.

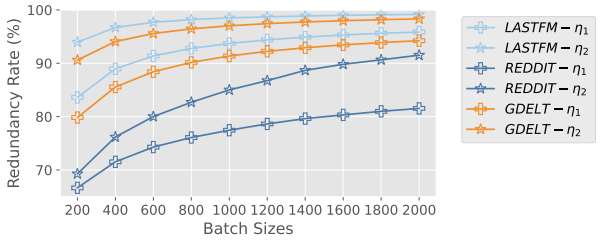
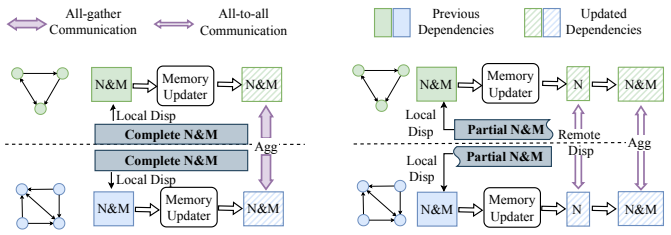


Fig. 3: Redundancy rate of the dispatch and aggregation phase.



(a) DepCache. (b) DepComm.

Fig. 4: The workflows for the disp (dispatch) and agg (aggregation) stages for DepCache and DepComm. (a) The memory update phase with two trainers of DepCache. A gray block represents that each trainer caches a complete N&M. (b) The memory update phase with two trainers of DepComm. A gray block represents that each trainer maintains a disjoint N&M subset.

2.3 Cross-Device Communication Overhead

With the development of TGNN models, there is a trend that aims to distribute the training process across devices to accelerate the training process. State-of-the-art distributed GNN frameworks, such as Deep Graph Library (DGL) [14], only support static GNNs in distributed settings. The key challenge to distributing dynamic GNNs lies in efficiently dealing with N&M dependencies which keep evolving during training iterations.

Recently, NeutronStar [16] proposes an adaptive approach to partition and distribute vertex dependencies across multiple devices, which implements hybrid dependency management mechanisms including DepCache and DepComm. Here, we introduce the idea of DepCache and DepComm in TGNNs training. To support distributed training, both of them partition N&M dependencies among trainers but DepCache requires duplicating N&N locally. As illustrated in Figure 4a, the key idea of DepCache is that each trainer maintains a complete N&M so that dependencies are readily prepared locally in the dispatch phase (called local dispatch). After that, DepCache performs the forward/backward propagation within each process following the data-parallel scheme. However, in the aggregation phase, a trainer needs to collect updated N&Ms for all trainers to update the local N&M. Essentially, all-gather operation is required at this stage. In contrast to DepCache, DepComm distributes independent N&M subsets across all trainers and collects the required N&M remotely as needed, which is illustrated in Figure 4b. DepComm firstly dispatches the N&M dependencies locally (i.e. local dispatch) and then dispatches updated node memory across trainers (i.e. remote dispatch) for subsequent embedding calculations. Since DepComm partitions the N&M dependencies into disjoint subsets among trainers, the remote dispatch requires all-to-all operation. Similarly, the aggregation phase requires another all-to-all operation after the forward/backward propagation. Finally, both DepCache and DepComm synchronize parameter gradients via all-reduce collective operation.

However, we find both DepCache and DepComm are inefficient due to cross-device communication overhead. We conduct experiments on various TGNNs to better understand the performance bottleneck of the two state-of-the-art mechanisms. In particular, we summarize the distributed training process as three phases, all-gather/all-to-all, all-reduce, and computation. Both DepCache and DepComm are evaluated on a 4-node 16-GPU cluster. We select three representative TGNN models: TGN [7], APAN [8], and JODIE [9] with dataset

TABLE 2: Breakdown of DepCache and DepComm. A2A/AG Comm. represents all-to-all and all-gather communication for N&M dependencies. AR Comm. represents all-reduce communication for model parameter synchronization. Comp. represents model computation.

		Time (ms) / percentage(%)		
Method	model	A2A/AG Comm.	AR Comm.	Comp.
DepCache	TGN	129.5/31.1	144.5/34.6	143.1/34.3
	APAN	302.9/25.7	193.1/16.4	683.2/57.9
	JODIE	218.9/40.0	182.3/33.3	146.5/26.7
DepComm	TGN	102.5/26.5	137.2/35.4	147.8/38.1
	APAN	247.9/19.4	324.0/25.3	707.1/55.3
	JODIE	82.8/19.3	197.0/45.9	149.2/34.8

LASTFM [9]. The physical cluster and model configurations can be found in Section 5.1. Table 2 summarizes the time consumption of different phases for the two mechanisms.

From the result, it can be seen that the communication overhead of N&M dependencies is a dominant factor for DepCache performance. This is due to the transfer volume of all-gather being proportional to the number of trainers. As Table 2 shows, cross-device communication of N&M dependencies occupies a relatively considerable proportion, which ranges from 25.7% to 40.0%. Furthermore, since each trainer hosts the entire N&M dataset, the device has to consume more memory to store it. Subsequently, the risk of out-of-memory (OOM) increases dramatically under memory pressure circumstances. In contrast to DepCache, DepComm adopts model parallelism to tackle N&M dependencies, which is more memory-efficient compared to DepCache. However, the required all-to-all operation still causes significant overhead in the end-to-end training, which takes up to 26.5% of training time for TGN. In addition, both DepCache and DepComm suffer from all-reduce overhead, which accounts for up to 45.9% of the training time. Considering all these system overheads, it is necessary to reduce the impact from communication on training.

In summary, we are motivated by the experimental results and analysis that it is necessary to design a more efficient approach tailored for TGNNs training workload to minimize the

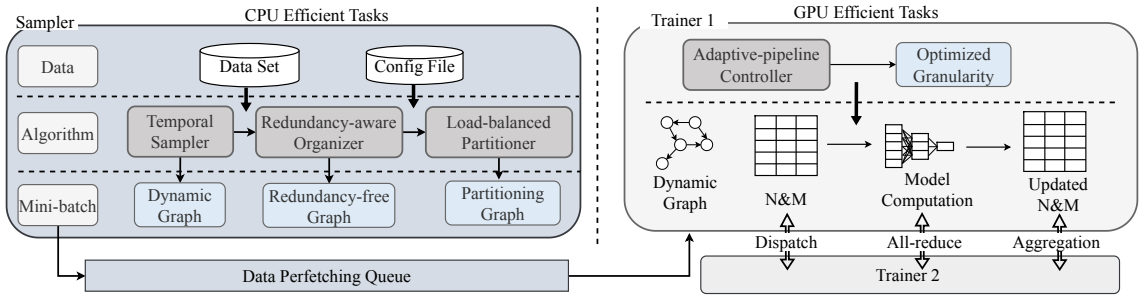


Fig. 5: The system architecture of Sven, featuring algorithm-system co-design.

transfer volume and mitigate the cross-device communication time consumption from a holistic perspective for improving training performance.

3 Sven System Design

We design and develop Sven to facilitate the end-to-end TGNN training performance in a collaborative manner. Sven innovations are three-fold, minimizing data transfer volume through redundancy-free graph organization and intra-batch N&M reuse, achieving balanced communication via the Re-FlexBiCut graph partitioning algorithm, and mitigating system overhead across different operations through multi-level pipelining. Figure 5 presents the system architecture of Sven. Sven consists of algorithm-level optimization for efficient N&M dependencies handling and system-level components for coordinating the training process. At the algorithm level, it features a redundancy-free and load-balancing sampler that reduces and balances communication volume across partitions. At the system level, it features an efficient trainer that utilizes a hierarchical pipelining mechanism with adaptive tuning.

3.1 Redundancy-free & Load-balance Sampler

3.1.1 redundancy-free data organization

As illustrated in Section 2.2, there are plenty of overlaps among intra-batch dependencies at the dispatch and aggregation stages. By extracting the overlap topology among input graphs from the original series and sampled series, the redundant data transmission can be reduced. Due to the requirements of dispatch and aggregation of TGNN, the de-duplication algorithm should be tailored to the data dependency of the input graph, as we discussed in Section 2.2. Figure 6 illustrates the data organization for the dispatch and aggregation phases.

We propose a redundancy-free algorithm to organize the extracted vertices IDs, which is illustrated in Algorithm 2. First, we flatten the interaction series of a graph into a vector according to the order of timestamps, as shown in the left column of Figure 1. Note that the vector V_o of the sampled graph in Figure 6a is converted at the vertex level while the vector of the original graph in Figure 6b is converted at the interaction level. The conversion is straightforward, so we utilize the former to represent them uniformly. Then, we traverse the vertices in vector V_o. For each vertex v_i, the algorithm first checks whether it has appeared. If not, we append it to the vertices set V_u. We locate the position

Algorithm 2: Redundancy-free graph organization.

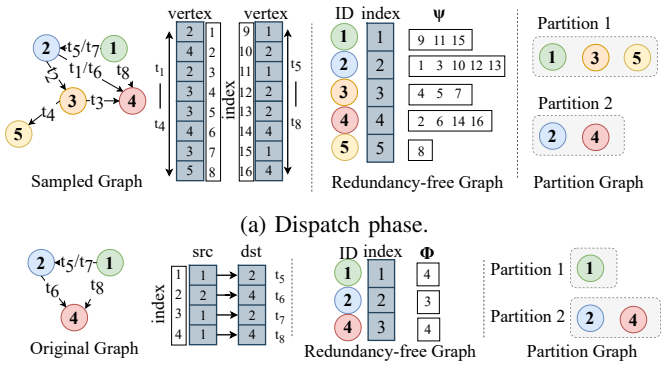
input : Input graph $G = \xi(t_s, t_e) = \{(v_i(t), v_j(t)) \| t \in (t_s, t_e)\}$, Output vertices $V_u = \{\emptyset\}$
output: Output vertices $V_u = \{v_{1'}, \ldots, v_{N'}\}$, location index for aggregation Φ, location index for dispatch Ψ

```
Flattening input graph G into vector V_o, where
  V_o = {(v_i(t_s), v_j(t_s) ..., (v_m(t_e), v_n(t_e)}
for v_i ← v_i(t_s) to v_n(t_e) do
    v_i ∈ ξ_j, where j is the index of ξ_j in G
    if v_i is not ∈ V then
        V_u = V_u + {v_i}
        Φ = Φ + {j}
    else
        replace the original index of v_i in Φ with {j}
    end
    find the index l of v_i in V_u, Ψ = Ψ + {l}
end
```

of the interaction ξ corresponding to v_i in the $\xi(t_s, t_e)$ and record its index j in Φ. If v_i is already contained in V_u, we use j to replace the index that appeared before. In this way, we filter out the redundant interaction in the aggregation phase, which is shown in the middle column of Figure 6b. Furthermore, we locate the index l of v_i in V_o and append l in Ψ. When the traversal is finished, V_u stores all the vertices without redundancy. The input graph from the redundancy-free graph organization is required to be restored after the dispatch phase. Therefore, as shown in the middle column of Figure 6a, Ψ is used to convert the input graph back to the state after the dispatch phase. In the other word, the redundancy-free strategy only removes redundant data during the communication process, while the input to the TGNN remained unchanged, ensuring that the output is equivalent. Given the number of vertices of V_o and V_u as N and N' respectively, the complexity of Algorithm 2 is $\mathcal{O}(NN')$.

3.1.2 Graph partitioning

In a distributed environment, N&M data is partitioned across trainers. However, the N&M dependencies are inherently tied to the vertices and continuously evolve during training. Therefore, to effectively capture the locality and communication patterns of the evolving N&M dependencies, we perform graph partitioning at the vertex-level, illustrated as Figure 8a. Based on the requirements of different trainers for N&M, a com-

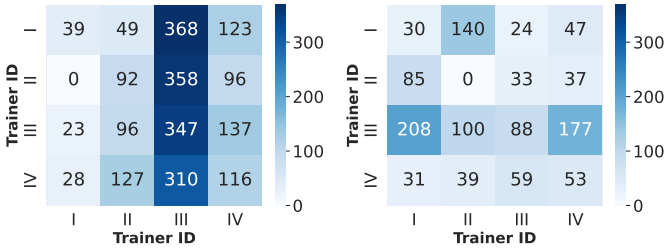


(a) Dispatch phase.

(b) Aggregation phase.

Fig. 6: Redundancy-free and load-balanced data organization.

munication topology for the all-to-all operation is established, shown in Figure 8b. Since all-to-all communication requires synchronization across trainers, the overall communication overhead is determined by the slowest communication link.



(a) Rang-based partition. (b) Interval-based partition.

Fig. 7: Heatmap of the residual matrix with 4 trainers.

Graph partitioning problem formulation

1)Problem formulation: Existing framework[14], [17] offer feasible methods, such as range-based and interval-based partitioning. However, we have observed that these coarse-grained approaches may not serve as one-size-fits-all solutions. To analyze quantitatively, we construct a communication topology matrix C, where C_{ij} represents the communication overhead between trainer i to j. The communication overhead is normalized as a multiple of the N&M data for a single vertex. Specifically, we define the residual matrix R as follows,

$$R = J_{M\times M} \cdot C_{max} - C \tag{2}$$

where $J_{M\times M}$ is a unit matrix. A higher value in matrix R indicates a less balanced communication. In Figure 8, we present heatmaps of the residual matrices for these two methods. The result demonstrates that neither of them can effectively address the issue of imbalanced communication.

We consider a pre-sampling step to obtain the topological relationship between trainers and vertices, and treat them as a graph, as illustrated in Figure 8c. Assuming the graph is $G = (V, E)$, with a set of trainers $T = \{t_1, \ldots, t_M\}$, and a set of vertices $V' = \{v_1, \ldots, v_N\}$, and a set of edges $E = \{e_1, \ldots, e_P\}$, where T and V' satisfy $T \cup V' = V$. We partition the set V into subsets $W_1, \ldots, W_M$ while ensuring that for $\forall i, j \in 1, \ldots, M$:

$$Wj = \{\{t_j\} \cup V'_j | V'_j \subseteq V'\} \tag{3}$$

Algorithm 3: Re-FlexBiCut algorithm

input : Input graph $G = (V, E)$
output: Disjoint subgraphs $W_1, ..., W_M$ where W_i contains vertices assigned to trainer t_i

Initialize sets $W_i = \{t_i\}$ containing just the trainer
Set $U = V \setminus (W_1 \cup W_2 \cup \ldots \cup W_M)$ as the set of unlabeled vertices
while $U \neq \emptyset$ **do**
  for *each trainer* $t_i \in T$ **do**
    Construct a graph G' with t_i as source, other trainers $\{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_M\}$ as sink
    Find approximate BiCut of G'
    $M_i = M_i \cup (M_i \cap U)$
    Calculate the balance cost $\zeta(W)$
    Resign intersection vertices $Y_j = W_i \cap W_j$ to W_j, calculate new balance cost $\zeta'(W)$
    if $\zeta(W) \geq \zeta'(W)$ **then**
      $M_i = M_i \setminus Y_j$ and $M_j = M_j \cup Y_j$
    end
    $U = U \setminus M_i$
  end
end

$$W_i \cap W_j = \emptyset \ \& \ W_1 \cup \ldots W_M = V \tag{4}$$

We introduce the *balance cost* between various subgraphs, denoted as $\zeta(W)$, where $\zeta(W) = \max\{W_j | j = 1, \ldots, M\}$. $\zeta(W_j)$ is defined as:

$$\zeta(W_j) = \sum_{v \in V'} \omega(t_j, v) \times \delta(t_j, v) \tag{5}$$

where $\omega(t_j, v)$ is the edge weight and $\delta(t_j, v)$ is an indicator function that indicates whether vertex v is in the partition W_j. Mathematically, the indicator function can be defined as:

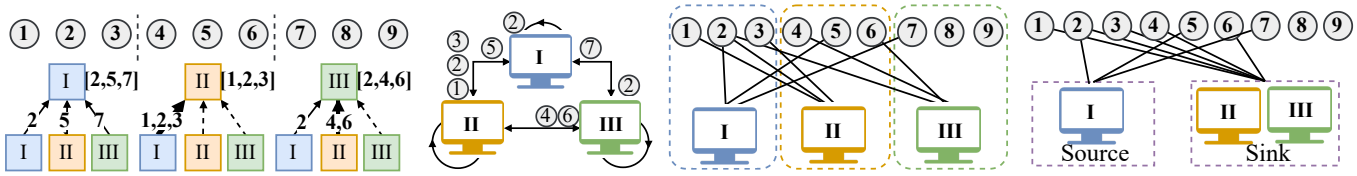
$$\delta(t_j, v) = \begin{cases} 1 & v \in W_j \\ 0 & v \notin W_j \end{cases} \tag{6}$$

The Communication-Balanced Graph Partition (CBG-P) Problem: Given a set of trainers T, a set of vertices V', and their connectivity E, we need to determine each trainer t_i and vertex v_j allocated to subset W_k, subject to constraints3and4, in order to to minimize the balance cost $\zeta(W)$.

2) The recursive bisectioning with flexible vertex reassignment algorithm for CBG-P problem: For the CBG-P problem, we have the following theorem:

Theorem 1. *CBG-P problem is NP-hard for any* $M \geq 4$.

Given the NP-hard nature of the CBG-P problem (with trainers $M \geq 4$), obtaining the globally optimal solution is computationally infeasible. However, to address this challenge, we develop the Recursive Bisectioning with Flexible Vertex Reassignment (Re-FlexBiCut) algorithm, which aims to achieve a locally optimal solution. At its core, Re-FlexBiCut recursively bifurcates the CBG-P problem into a series of bisection cuts[18], regarding pairings of trainers as binary splittings. Bisection cut problems can be approximately solved in polynomial time using existing algorithms like BiCut [19]. More formally, Re-FlexBiCut operates by iterating over trainers and constructing an source&sink graph with t_j as the source and other trainers as the sink, as illustrated in Figure 8d.



(a) Above is an illustration of vertex-level graph partitioning, and below is the communication topology for each trainer. (b) The communication topology for an all-to-all operation. (c) Redefine the partition problem as a graph cut problem. Each subgraph must include one trainer. (d) Source&sink graph with one trainer as the source and the aggregation of other trainers as the sink.

Fig. 8: A graph partition illustration with three trainers and nine vertices.

The overall algorithm for Re-FlexBiCut is outlined in Algorithm 3. Re-FlexBiCut operates by alternating between two key stages - computing BiCut to partition unlabeled vertices across trainers and revisiting prior assignments to less balance cost flexibly. In the partition stage, Re-FlexBiCut first calls BiCut to find a cut surrounding M_i that approximately maximizes the number of unassigned vertices (Line 6), which run in $\mathcal{O}(N^2 \log^2 N)$ time. Then it adds M_i's side vertices to subgraph M_i (Line 7). In the flexible reassignment stage, Re-FlexBiCut addresses suboptimal decisions made by the greedy bisection cuts by reducing the balance cost across assignments (Line 9-12), which takes $\mathcal{O}(P)$ time. Since Re-FlexBiCut takes $\mathcal{O}(\log N)$ iterations for each trainer, its computational complexity is $\mathcal{O}(M \log N(N^2 \log^2 N + P))$

Theorem 2. *The approximation ratio for the Re-FlexBiCut algorithm is* $\mathcal{O}(\log^2 N)$.

Re-FlexBiCut is an approximation algorithm for the CBG-P problem. It can achieve locally optimal results with an approximation ratio of $\mathcal{O}(\log^2 N)$ for the CBG-P problem. We place the proof of Theorem 1 and 2 in Appendix A and B respectively.

3.2 Communication volume analysis

We analyze the incurred communication volume when accessing N&M dependencies for DepCache, DepComm, and Sven. We assume that the communication achieves ideal load-balancing. For clarity, we define the following terms: The number of sampled and original vertices is denoted as N_1 and N_2, respectively, and their redundancy ratios as η_1 and η_2, respectively. The unit memory size of data is represented by C_0. The number of trainers is denoted as M. Additionally, we use d_{msg}, d_{mem}, and d_{edge} to represent the dimensions of the message, node memory, and edge feature, respectively.

Communication volume for DepCache.

$$C_{DE} = N_2 * (1 - \eta_2) * (d_{msg} + d_{mem}) * \frac{2(M-1)^2}{M} * C_0 \quad (7)$$

Communication volume for DepComm.

$$\begin{aligned} C_{DM} = {} & N_1 * d_{mem} * \frac{2(M-1)}{M} C_0 \\ & + N_2 * (1 - \eta_2) * (d_{msg} + d_{mem}) * \frac{2(M-1)}{M} * C_0 \end{aligned} \quad (8)$$

Communication volume for Sven.

$$\begin{aligned} C_{Sven} = {} & N_1 * (1 - \eta_1) * (d_{msg} + d_{mem}) * \frac{2(M-1)}{M} * C_0 \\ & + N_2 * (1 - \eta_2) * (d_{msg} + d_{mem}) * \frac{2(M-1)}{M} * C_0 \end{aligned} \quad (9)$$

We leave all formulation derivations in Appendix C. Based on these equations, we draw the following conclusions:

1.Increasing the number of workers results in a more prominent deficiency of DepCache due to its communication volume increasing proportionally with the number of trainers, as inferred from Equation 7. 2.Equation 8 shows that DepComm is more sensitive to the number of vertices in the sampled graph, as it cannot efficiently handle N&M dependencies during the dispatch phase. 3.Sven tackles N&M dependencies using the model-parallel scheme, and its communication volume remains independent of the number of trainers, as demonstrated in Equations 9. Moreover, the proposed redundancy-free strategy benefits both the dispatch and aggregation phases.

Furthermore, we define the number of vertices of a redundancy-free graph at the dispatch phase as T times that at the aggregation phase. If the number of trainers is satisfied with the following condition,

$$M > T + 2 \quad (10)$$

we can conclude that condition $C_{Sven} < C_{DepCache}$ holds. Typically, T is less than two according to our measurement.

In addition, we define $P = d_{mem}/d_{edge}$, when the following condition holds, that is,

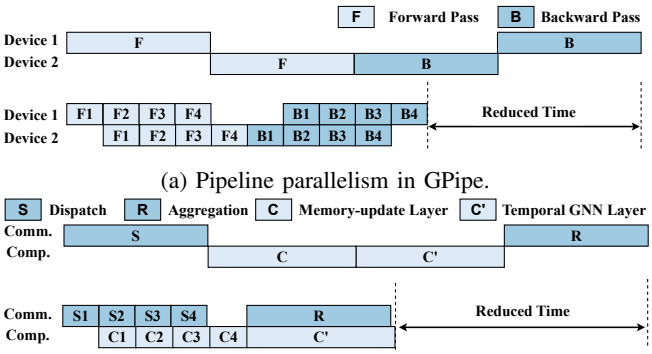
$$\eta_1 > \frac{2}{3} - P \quad (11)$$

we can derive $C_{Sven} < C_{DepComm}$. We can observe that the redundancy ratio is over $\frac{2}{3}$ in Figure 3. Thus, Sven performs better than DepComm in the cases studied in this paper.

3.3 Hierarchical Pipeline Parallelism

3.3.1 *Prefetching*

Sven incorporates a prefetch mechanism to conceal the overhead of generating batch data. Note that in heterogeneous GPU-CPU clusters, GPUs are responsible for the compute-intensive training tasks and CPUs are in charge of processing input batches. It can be noticed that the input batch data only has downstream consumers and are independent of each other. Sven develops a queue-based prefetch component. Concerning the sampler, Sven prepares mini-batch graph topology and applies the redundancy-free and vertex-level graph partitioning algorithms, after which the process graph structures are cached into the queue. Afterward, the trainer fetches the mini-batch from the cache queue directly without waiting for the processing stage. With the prefetch mechanism, the sampling stage can be performed in parallel with other stages, resulting in the reduction of end-to-end training time.



(a) Pipeline parallelism in GPipe.

(b) The proposed micro-batch pipeline parallelism

Fig. 9: Pipeline parallelism in GPipe and Sven.

3.3.2 Adaptive micro-batch pipelining

As analyzed in section 2.3, the performance of training TGNN is limited by the all-to-all operation for N&M dependencies. To mitigate the impact of the cross-device communication, we introduce micro-batch mechanism, which is first proposed in GPipe [20], shown in Figure 9a. In the naive training approach, depicted at the top of Figure 9b, only one mini-batch is active for computation or communication. We split a mini-batch into smaller micro-batches and pipeline their execution one after the other, shown in the bottom of Figure 9b. The micro-batch pipelineing allows GPUs to process communication of dispatch and computation of memory-update layer concurrently while preserving the sequential dependency of the network. Afterward, since there is no dependency between the aggregation and temporal GNN layers, they can be executed simultaneously. Through the two-level pipeline parallelism, the majority of bubble time is eliminated.

Adaptive granularity configuration. The effectiveness of pipeline parallelism is significantly influenced by the granularity, which refers to the number of micro-batch partitions, denoted as n. A coarse-grained granularity may fail to take the benefit of pipelining, while an overly fine-grained granularity may lead to GPU underutilization. To address this, we adaptively tune n as follows:

$$n = \lfloor (A * bs + B)/S \rfloor \tag{12}$$

where η and bs represent the redundancy ratio and the batch size, respectively. A and B are constants determined by the characteristics of the datasets, and S is a constant determined by the transmission speed of the system hardware. We place the derivation of Equation 12 in Appendix D.

We train the model for several iterations and collect the required data in Equation 12. Since the training process may endure thousands of iterations, the time consumption of profiling can be negligible. Note that splitting mini-batch (its size equals bs) into n micro-batches (its size equals m) does not affect the training convergence, because 1) the total number of batches per parameter update is unchanged, i.e. $m * n$=bs, and 2) differently from CNN models where BatchNorm operations are influenced by the batch size, GNN adopts LayerNorm instead, which is irrelevant to the batch size. As a result, though we split the mini-batch into several micro batches, the accumulated gradients per mini-batch are unchanged, achieving the same accuracy.

3.3.3 Asynchronous pipelining

Finally, we adopt asynchronous pipelining to alleviate the overhead caused by all-reduce operations. Inspired by PipeSGD [21], we deploy pipelined training with a width of two iterations taking advantage of both synchronous and asynchronous training. Sven introduce staleness to allow the backward propagation and optimization phases to be executed simultaneously with the all-reduce operation. To ensure the convergence of the model, Sven bounds the staleness to one step. Therefore, the weight update is as follows,

$$\omega_{t+1} = \omega_t - \alpha \cdot \nabla f(\omega_{t-1}) \tag{13}$$

where ω_t contains the model weight parameters after t iterations, which is one for Sven, ∇f is the gradient function, α is the learning rate and ω_{t-1} is the weight used in iteration t. As pointed out by p^3 [22], a potential concern with applying unbounded stale gradient techniques is the negative impact on the convergence and accuracy of the network. However, the bounded delay eliminates the above issues and ensures identical model accuracy as that of data parallelism while significantly reducing the training time. We demonstrate the accuracy of Sven's asynchronous pipelining in Section 5.7.

4 Implementation

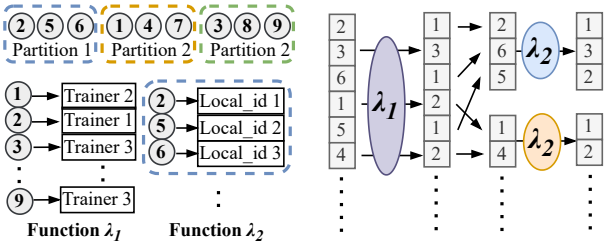
Sven is an end-to-end distributed TGNN training library implemented on top of PyTorch [23] 1.12.0 and DGL [14] 0.9.0 with CUDA 11.7 and NCCL [24] 2.10.3. A few key components and functionalities are implemented as follows.

4.1 The Partitioner

To achieve the mapping from global vertex IDs to the storage location index of their N&M data, we employ a dual-hash mapping strategy. Specifically, we use 'local_id' to represent the index of the N&M data within the corresponding trainer. We establish two hash maps, denoted as λ_1 and λ_2, based on our partition algorithm, as illustrated in Figure 10a. The former is maintained by all trainers, while the latter is maintained by each respective trainer. Initially, we map 'global_id' to 'trainer_id' using the hash function λ_1, followed by the accumulation of 'global_id' values that share the same 'trainer_id'. Lastly, within the corresponding trainer, we use hash function λ_2 to map 'global_id' to 'local_id', as depicted in Figure 10b.

4.2 The Trainer

We implement the hierarchical pipeline parallelism mechanism based on the communication package provided by PyTorch 'DistributedDataParallel' [25]. NCCL [24] all-to-all collective operator is adopted to dispatch and aggregate N&M dependencies among GPUs. NCCL all-reduce collective operator is used to synchronize gradients before being used for weight updating. An individual CUDA stream is created to split the execution of the mini-batch into several micro-batch linear sequences that belong to a specific device and it is independent of other streams. As for the asynchronous pipelining mechanism, we override the vanilla all-reduce in 'DistributedDataParallel' [25] and register it with the 'communication hook' interface to achieve bounded staleness.



(a) Dual-hash Map (b) Vertex IDs mapping

Fig. 10: Implementation of dual-hash mapping.

5 Evaluation

5.1 Experimental Setup

5.1.1 Physical cluster.

Our experiments are conducted on an HPC cluster with 16 nodes. Each node is equipped with 4 GPUs as well as 2 CPUs and has 128GiB RAM (shared by the 4 GPUs). Each GPU is NVIDIA Tesla V100 with 16GiB HBM and each CPU has 20 cores of 2.4GHz Intel Xeon E5-2640 v4. These nodes are connected by 56Gbps HDR Infiniband.

TABLE 3: Dataset characteristics. The timestamp represents the maximum edge timestamp while Dims represent the dimensions of edge features.

Dataset	Nodes	Edges	Timestamp	Dims
REDDIT	9K	157K	$2.7e^6$	172
LASTFM	2K	1.3M	$1.3e^8$	127
GDELT	17K	191M	$1.8e^5$	182

5.1.2 Datasets and TGNN models.

Table 3 lists the major parameters of dynamic graph datasets that we used in the experiments. We utilize three datasets provided by TGL [10] in the evaluation. REDDIT [7] as well as LASTFM [9] are medium-scale and bipartite dynamic graphs. GDELT [26] is a large-scale dataset containing 0.2 billion edges. Furthermore, we select three representative TGNN models including APAN [8], JODIE [9], and TGN [7] for performance evaluation. The message size of the three models is set to 10 for high GPU utilization. The batch size is set to 1,200 by default. For all experiments, we adopt Adam [27] as the optimizer.

5.1.3 Methodology.

We compare the performance of Sven with TGL [10], DepCache, and DepComm implementation. TGL is a distributed training framework for TGNNs aimed at single-node multi-GPU systems. It adopts the share-memory approach to store N&M in host memory [14], which relies on the GLOO backend for communication. However, cross-machine communication using GLOO backend is much slower than that of NCCL, which results in significant performance loss when scaling out the training process. Therefore, all measurements of TGL are conducted on the single-machine setup. To enable training TGNN with a large batch size, we adopt the 'random chunk scheduling' policy introduced by TGL for all methods.

5.2 Sven Performance Analysis

Firstly, we study the overall performance comparison with Sven and other state-of-the-art approaches and the scaling behavior of Sven.

5.2.1 Performance comparison.

Figure 11a presents the results of various model and dataset combinations. The y-axis in the histograms represents the average execution seconds of one iteration.

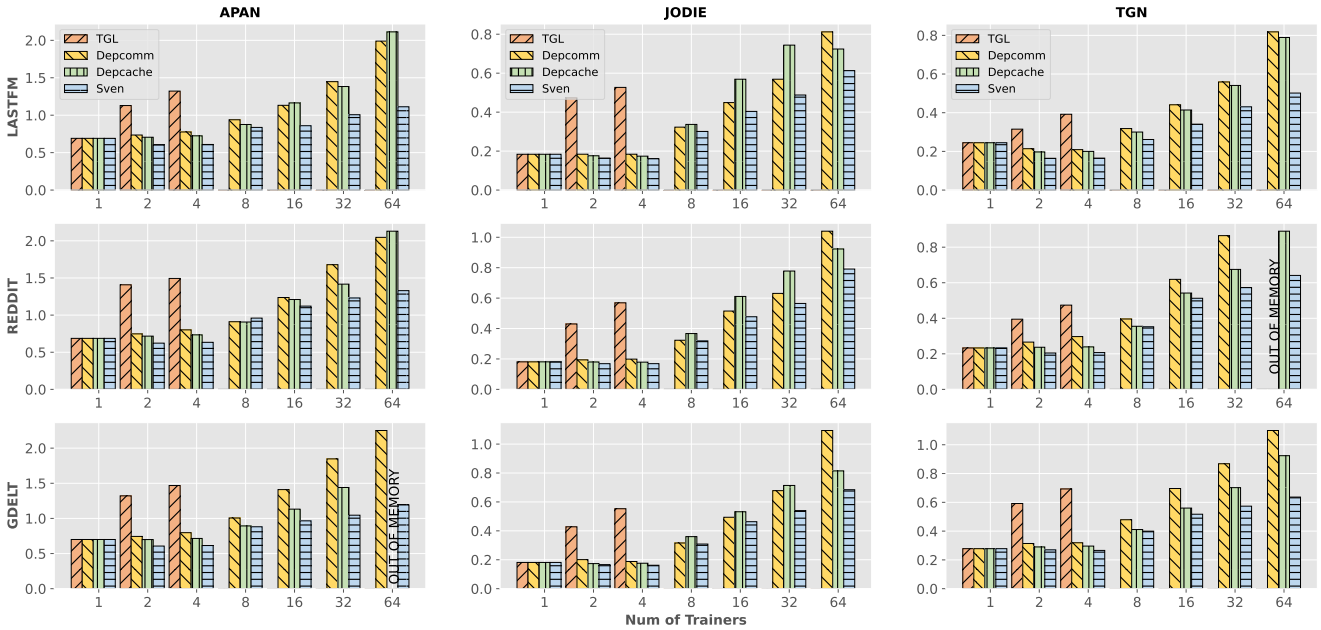
Compared to TGL. In a single-node setup, TGL performs the worst compared to other methods. Sven achieves up to 3.42x speedup against TGL. Since TGL caches N&M on the host memory, moving burdensome data dependencies from the CPU to the GPU through the PCIe bus introduces additional I/O overhead. In contrast, Sven, DepComm, and DepCache place N&M on the GPU to eliminate the overhead.

Compared to DepComm. Sven achieves up to 1.89x speedup against DepComm. The advantages of Sven are more prominent for those models with higher N&M dimensions, e.g. APAN, and datasets with higher redundancy ratios, e.g. LASTFM and GDELT. As revealed by Equation 11, if the dimension of N&M scales up and the repetition rate of the input graph increases, the discrepancy between the two sides of the inequality grows more pronounced, i.e., the disparity of communication cost between Sven and DepComm becomes prominent. The evaluation results are consistent with Equation 11, demonstrating the effectiveness of the proposed redundancy-free strategy. Moreover, DepComm encounters the OOM (Out-of-memory) issue when training the TGN model with REDDIT datasets, which is caused by the data redundancy with the memory-update phase that cannot be handled by the redundancy-free strategy.

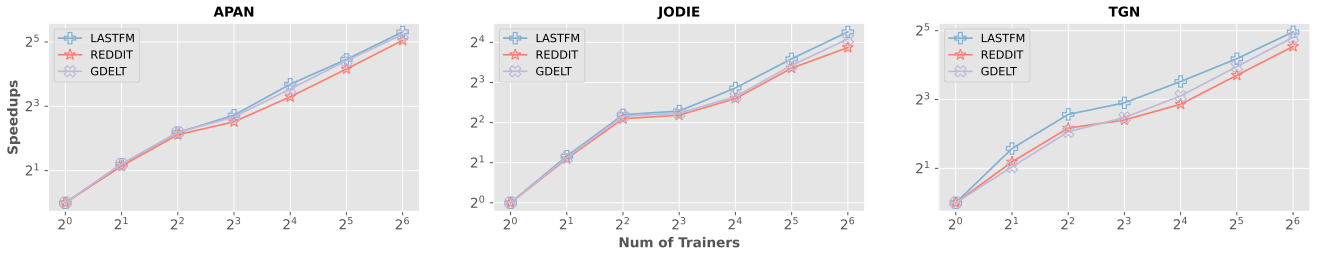
Compared to DepCache. Sven can improve up to 1.91x speedup against DepCache. From the experimental results, with APAN and TGN models, the superiority of Sven over DepCache increases when scaling up the cluster. As demonstrated in Equation 7, DepCache is very sensitive to the size of the cluster, meaning that the communication volume of DepCache is proportional to the number of trainers. However, it should be noted that Sven only outperforms DepCache slightly on the JODIE model. This is because the JODIE model does not have a temporal sampler process, which greatly reduces the benefits of the redundancy-free approach. Furthermore, DepCache maintains a whole copy of N&M on each trainer, which makes it encounter the OOM problem when training the APAN model with the GDELT dataset.

5.2.2 Scaling study

Figure 11b summarizes the speedup curves for all datasets. Taking $M = 1$ as the reference point, the plot presents the speedup achieved as we increase the number of processors. For model APAN, the speedup is up to 40x, 33x, and 38x for datasets LASTFM, REDDIT, and GDELT respectively at $M = 64$, as against the ideal value of 64x. The best speed gain is up to 31x for model TGN at $M = 64$. The scalability of the model JODIE is relatively weak, as its maximum speedup is less than 20x. Compared with models APAN and TGN,



(a) Overall performance comparisons (experimental results are shown for each data-model pair).



(b) Scalability of Sven (speedup on all datasets for different processors with $M = 1$ as the reference).

Fig. 11: Comparisons of Sven with three state-of-the-art approaches in terms of training time and scaling speedup.

JODIE does not require the process of sampling, which leads to a small number of sampled nodes. This greatly reduces its communication overhead for N&M. However, the time of model synchronization becomes the main bottleneck. In addition, we find that for various models, the speedup trends of different datasets are identical. LASTFM has the highest speedup, followed by GDELT, and REDDIT has the least. This is consistent with the order of redundancy rates for each dataset, illustrating the effectiveness of Sven's redundancy-free strategy. Finally, compared to the scenario of $M = 4$, we observe a drop in speedup at $M = 8$. When $M > 4$, the cross-machine communication over the network is introduced, resulting in reduced speedup gains.

5.3 Communication efficiency analysis

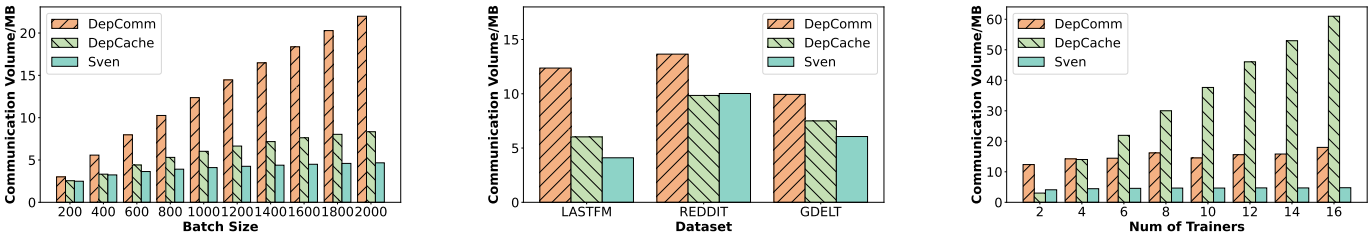
As we discussed in section 3.2, the redundancy of dynamic graphs and the number of trainers affect the communication volume of various methods. To study the communication efficiency among DepComm, DepCache, and Sven, we vary the batch size, datasets, and the number of workers. We set the default batch size, datasets, and the number of trainers to 1000, LASTFM, and 4, respectively.

5.3.1 *Impact of batch size*

We investigate how various batch sizes affect communication volume. As presented in Figure 12a, batch size has the most considerable impact on DepComm because DepComm cannot handle the burdensome N&M dependencies at the dispatch phase. It can also be derived from Equation 8 that the transfer increases linearly with batch size. DepCache is not sensitive to batch size compared to DepComm, because the redundancy-free strategy can filter out redundant data at the aggregation phase. Note that Sven outperforms all schemes. Sven achieve communication efficiency from 1.25x to 5.26x compared to DepComm, and 1.16x to 2.01x compared to DepCache. The results demonstrate that Sven gains the maximum benefit from the redundant-aware strategy and it is more tolerant of the variation in batch size.

5.3.2 *Impact of datasets*

Figure 3 and Table 3 show that the redundancy ratio and dimension of edge features vary across different datasets. Thus, we evaluate the effect of the two factors on communication volume. Figure 12b shows that Sven achieves an average communication efficiency of 2.09x and up to 3.30x compared to DepComm, and 1.23x and up to 1.47x of that compared to DepCache. Note that the results show the most prominent gap



(a) Impact of batch size. (b) Impact of datasets. (c) Impact of trainers.

Fig. 12: The communication volume of DepComm, DepCache, and Sven.

between DepComm and Sven with dataset LASTFM due to its highest redundancy ratio and smallest dimension of the edge feature. Similarly, we can observe the narrowest margin for REDDIT because of the fewest redundant vertices of REDDIT.

5.3.3 *Impact of trainers*

To evaluate the impact of the number of trainer processes on the communication volume, we vary the number of trainers for the LASTFM dataset, ranging from 2 to 16. Figure 12c shows Sven achieves an average of 2.29x and 6.11x communication efficiency against DepComm and DepCache, respectively. From Equation 8 to Equation 9, we know that the performance of both DepComm and Sven is independent of the number of trainers because both adopt model parallelism to handle the N&M dependencies. However, the communication volume of DepCache is proportional to the number of trainers. Compared to the results in Section 5.2.1, we notice that the reduction in communication volume does not result in a corresponding improvement in end-to-end performance, which is due to the fact that the time complexity of the all-gather communication is less than that of all-to-all communication [28].

In a nutshell, the analysis demonstrates that Sven removes redundant N&M dependencies and achieves the best scalability.

5.4 Communication balance analysis

We conducted a comparative analysis of our Re-FlexBiCut method against the range-based and interval-based methods concerning communication balance. The evaluation was carried out by varying the batch size and the number of trainers while using the APAN model and REDDIT dataset. The default batch size and number of trainers were set to 1200 and 8, respectively. Each model was run for 10 epochs, and we illustrate the upper quartile and the box plot of balance cost, computed based on Equation 5.

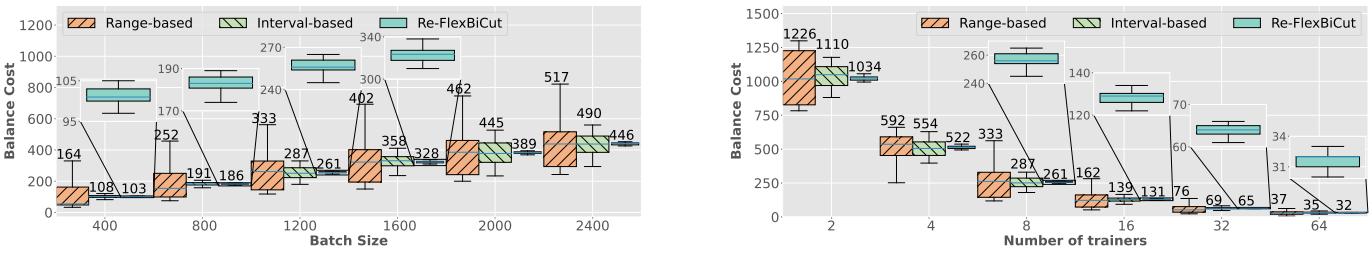
5.4.1 *Batch size sensitivity analysis*

In general, there is a clear upward trend in balance cost as the batch size increases from 400 to 2400 for all methods. Larger batch sizes lead to more remote requests for handling N&M dependencies, resulting in increased discrepancies among the trainers. As the batch size increases, the range-based and interval-based method demonstrates a steeper rise in balance cost, indicating a growing imbalance in their performance. Remarkably, our Re-FlexBiCut method maintains the most compact interquartile ranges across all batch size settings, reducing the maximum balance cost by 59.2% and 48.5% compared to the range-based and interval-based methods, respectively. These results demonstrate the superior robustness of Re-FlexBiCut in achieving communication balance.

5.4.2 *Number of trainers sensitivity analysis*

Overall, there is an ascending trend in balance cost as the number of trainers increases from 2 to 64 for all three methods. With a larger cluster, each trainer is assigned fewer vertices, resulting in decreased communication requests for each trainer. Critically, the range-based and interval-based method exhibits wider spreads, especially for small numbers of trainers. In contrast, Re-FlexBiCut consistently maintains the most compact interquartile ranges across all settings, achieving a maximum 27.6% and 10.1% reduction in communication cost, respectively.

Overall, the results demonstrate that Re-FlexBiCut outperforms the range-based and interval-based methods in terms of balance cost, showcasing its effectiveness and stability in addressing the communication balanced vertices partition problem.



(a) Batch size sensitivity (b) Number of trainers sensitivity

Fig. 13: The communication balance cost of range-based, interval-based and Re-FlexBiCut.

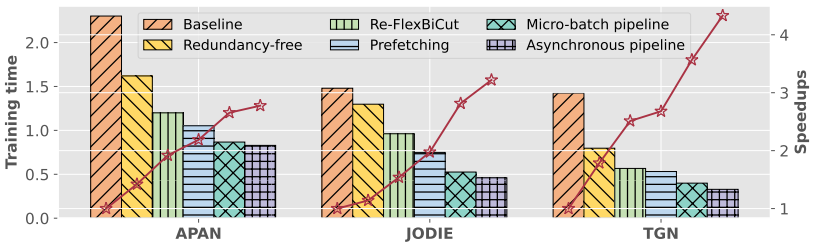


Fig. 14: The overall performance breakdown of Sven.

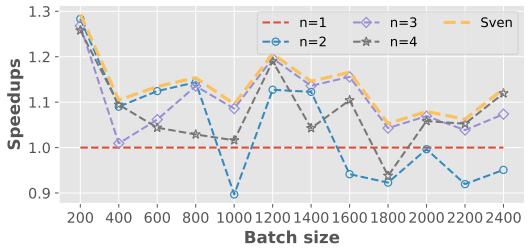


Fig. 15: Effectiveness of adaptive granularity.

5.5 Sven performance breakdown

We discuss the performance breakdown of Sven's key components. We regard the naive data parallelism as the reference baseline, and continuously assemble individual components to examine performance improvement. Figure 14 shows the performance of TGNN models APAN, JODIE, and TGN using dataset LASTFM. The left y-axis represents the training time for one iteration and the right one represents the speedups against the naive implementation.

We integrate the redundancy-free strategy to examine its performance benefit, which mainly achieves communication efficiency improvement. Compared to the naive baseline, the speedup is up to 1.42x, 1.14x, and 1.78x for training models APAN, JODIE, and TGN. As we analyzed in Section 5.2.1, the communication volume of model JODIE is insignificant compared to that of the other two TGNN models. The removal of the sampling module results in less traffic reduction after extracting redundant parts. Therefore, the redundancy-free strategy yields the lowest benefit for JODIE. Furthermore, we study the effect of different vertex partitioning strategies. We adopt range-based partitioning as the default method. For models APAN, JODIE, and TGN, our Re-FlexBiCut partitioning strategy obtains 1.35x, 1.35x, and 1.41x speedup respectively, because it is more balanced and able to reduce the amount of N&M dependencies communication. We verify the effectiveness of the prefetching mechanism. Prefetching brings 12.3%, 22.1%, and 6.2% improvement on models APAN, JODIE, and TGN respectively. The results demonstrate that Sven can schedule CPU and GPU operations simultaneously and effectively hide the data preparation time. The next component is the adaptive micro-batch pipelining, in which we overlap the dispatch all-to-all communication with memory update layer computation and overlap aggregation all-to-all time with temporal GNN layer time. For models APAN, JODIE, and TGN, we get a performance boost of 1.21x, 1.43x, and 1.33x respectively after integrating the micro-batch component. Lastly, we examine the efficiency of the proposed asynchronous pipelining. For models APAN, JODIE, and TGN, the speed is increased by 1.05x, 1.14x, and 1.21x respectively.

Overall, the maximum speedup is 4.33x, and the minimum speedup is over 2.5x against the naive baseline. The result demonstrates the effectiveness of Sven's components.

5.6 Effectiveness of granularity configuration

We examine the effectiveness of the adaptive pipeline granularity configuration of Sven, which is based on a hypothesis that n is monotonically increasing as communication volume increases. Intuitively, when the batch size continues to increase, the all-to-all communication time will increase accordingly, which requires a fine-granularity configuration. We compare the performance with various batch sizes on the APAN-LASTFM model dataset pair. We regard the implementation without micro-batch pipelining as the baseline, in which n equals 1. Figure 15 shows that when the batch size is smaller than 800, $n = 2$ is the best option. When the batch size is between 800 and 2,000, $n = 3$ maintains the most impressive performance. When batch size is larger than 2,000, $n = 4$ becomes the optimal option. Sven performs the best in all situations.

TABLE 4: Average precision for link prediction task.

	APAN	JODIE	TGN
TGL	80.19%	77.69%	88.42%
Sven	80.93%	78.12%	88.37%

5.7 Accuracy validation

We validate the prediction accuracy and study the impact of asynchronous pipelining. We evaluate the accuracy of TGL and Sven on dataset LASTFM within 4 GPUs (one physical node) for a fair comparison. Table 4 presents the accuracy comparison between TGL and Sven in the link prediction task. Following TGL, we adopt average precision to measure the accuracy on positive and negative test edges. We set the maximum training epoch to 100 and evaluate the accuracy of the model with the best performance on the validation set. We can observe that Sven achieves similar or higher precision than TGL among APAN, JODIE, and TGN. The experimental results demonstrate that Sven ensures the same accuracy as TGL does with asynchronous pipelining.

6 Related Work

Systems for static GNNs. Several GNN systems are developed recently to support large-scale GNN training and tackle remote dependency communication. Distributed GNN systems such as AliGprah [29], DistDGL [30], DistDGL_v2 [31], and AGL [13] adopt dependency-cached approaches. They prepare dependencies data locally and work together with

data parallel techniques. Frameworks such as DistGNN [32] and DGCL [33] exploit dependency-communicated schemes to transfer dependency data between processors.

Systems for temporal GNNs. Temporal GNNs can capture temporal information and topological structure, and thus outperform static GNNs in many applications [10]. Over the past few years, several research efforts have been made to achieve efficient end-to-end TGNNs training. Many of them concentrate on the DTDGs, which represent a dynamic graph as a sequence of snapshots sampled at regular intervals. PiPAD [34] and ESDG [15] both recognize the topological similarity between snapshot graphs and extract overlapping parts to avoid unnecessary data transmission. However, the data in CTDGs is not discrete, so these methods are not applicable to more general scenarios. Besides, PiPAD utilizes pipelined and parallel mechanisms to execute computation and communication asynchronously, which is orthogonal to our hierarchical pipeline parallelism. DynaGraph [35] also discovers the reuse opportunity in DTDGs and proposes cached message-passing to reduce communication overhead, which is similar to our redundancy-free strategy. However, our method is capable of handling the situation where the topology of the graph keeps evolving. PyTorch Geometric Temporal [36] and TGL [10] are two general frameworks built on top of PyG [12] and DGL [14] to support TGNNs training. However, they provide limited support for extending dynamic graph training. TGL only supports single-machine multi-GPU scenarios and is unable to tackle the temporal dependency issue on distributed training systems. NeutronStar [16] analyzes two state-of-the-art designs for resolving the issue of vertex dependencies, namely DepCache, and DepComm. The idea behind DepCache is to cache dependencies locally and prepare the required dependencies before training. The key idea of DepComm is that dependencies are distributed among trainers and remote communication of dependencies is performed as needed. Instead of following DepCache to cache the complete N&M, Sven distributes the data dependencies across machines in a model-parallel scheme, which enables training large graphs. Compared with DepComm, which introduces two dispatch phases, including local and remote dispatch, Sven only requires one dispatch. This design can increase the data dependencies reusing possibility. Furthermore, we propose a general redundant data elimination strategy to reduce redundancy. To the best of our knowledge, Sven is the first scaling study specifically for temporal GNN training on CTDGs.

7 Conclusion

This paper presents Sven, a redundancy-free and communication-balance high-performance library for distributed temporal GNN training, which optimizes end-to-end performance with hierarchical pipeline parallelism. With efficient redundancy-free data organization and load-balancing partitioning strategies, Sven addresses the key challenges of excessive data transferring. Sven holistically integrates data prefetching, adaptive micro-batch pipelining parallelism, and asynchronous pipelining to tackle the communication overhead. Experimental results show that Sven achieves up to 1.9x-3.5x speedup, 5.26x communication efficiency improvement, and 59.2% balance cost reduction.

References

[1] P. Battaglia, R. Pascanu, M. Lai, D. J. Rezende, and K. kavukcuoglu, "Interaction networks for learning about objects, relations and physics," in *NeurIPS 2016*, 2016, pp. 4509–4517.

[2] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[3] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NeurIPS 2017*, 2017, pp. 1025–1035.

[4] F. Monti, D. Boscaini, J. Masci, E. Rodola, J. Svoboda, and M. M. Bronstein, "Geometric deep learning on graphs and manifolds using mixture model cnns," in *IEEE CVPR 2017*, 2017, pp. 5115–5124.

[5] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," in *NeurIPS 2018*, 2018, pp. 5171–5181.

[6] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *ICML 2017*. PMLR, 2017, pp. 1263–1272.

[7] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein, "Temporal graph networks for deep learning on dynamic graphs," *arXiv preprint arXiv:2006.10637*, 2020.

[8] X. Wang, D. Lyu, M. Li, Y. Xia, Q. Yang, X. Wang, X. Wang, P. Cui, Y. Yang, B. Sun *et al.*, "Apan: Asynchronous propagation attention network for real-time temporal graph embedding," in *ACM SIGMOD 2021*, 2021, pp. 2628–2638.

[9] S. Kumar, X. Zhang, and J. Leskovec, "Predicting dynamic embedding trajectory in temporal interaction networks," in *ACM SIGKDD 2019*, 2019, pp. 1269–1278.

[10] H. Zhou, D. Zheng, I. Nisa, V. Ioannidis, X. Song, and G. Karypis, "Tgl: A general framework for temporal gnn training on billion-scale graphs," in *VLDB 2022*, 2022.

[11] G. Rossetti and R. Cazabet, "Community discovery in dynamic networks: a survey," in *ACM computing surveys (CSUR) 2018*, 2018, pp. 1–37.

[12] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[13] D. Zhang, X. Huang, Z. Liu, J. Zhou, Z. Hu, X. Song, Z. Ge, L. Wang, Z. Zhang, and Y. Qi, "Agl: a scalable system for industrial-purpose graph machine learning," in *VLDB 2020*, 2020, pp. 3125–3137.

[14] M. Y. Wang, "Deep graph library: Towards efficient and scalable deep learning on graphs," in *ICLR workshop on representation learning on graphs and manifolds 2019*, 2019.

[15] V. T. Chakaravarthy, S. S. Pandian, S. Raje, Y. Sabharwal, T. Suzumura, and S. Ubaru, "Efficient scaling of dynamic graph neural networks," in *IEEE SC 2021*, 2021, pp. 1–15.

[16] Q. Wang, Y. Zhang, H. Wang, C. Chen, X. Zhang, and G. Yu, "Neutronstar: distributed gnn training with hybrid dependency management," in *ACM SIGMOD 2022*, 2022, pp. 1301–1315.

[17] Y. Xia, Z. Zhang, H. Wang, D. Yang, X. Zhou, and D. Cheng, "Redundancy-free high-performance dynamic gnn training with hierarchical pipeline parallelism," in *ACM HPDC 2023*, 2023, pp. 119–132.

[18] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified np-complete problems," in *Proceedings of the sixth annual ACM symposium on Theory of computing*, 1974, pp. 47–63.

[19] U. Feige and R. Krauthgamer, "A polylogarithmic approximation of the minimum bisection," *SIAM Journal on Computing*, vol. 31, no. 4, pp. 1090–1118, 2002.

[20] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: efficient training of giant neural networks using pipeline parallelism," in *NeurIPS 2019*, 2019, pp. 103–112.

[21] Y. Li, M. Yu, S. Li, S. Avestimehr, N. S. Kim, and A. Schwing, "Pipe-sgd: a decentralized pipelined sgd framework for distributed deep net training," in *NeurIPS 2018*, 2018, pp. 8056–8067.

[22] S. Gandhi, A. P. Iyer, H. Xu, T. Rekatsinas, S. Venkataraman, Y. Xie, Y. Ding, K. Vora, R. Netravali, M. Kim *et al.*, "P3: Distributed deep graph learning at scale," in *USENIX OSDI 2021*, 2021, pp. 551–568.

[23] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: an imperative style, high-performance deep learning library," in *NeurIPS 2019*, 2019, pp. 8026–8037.

[24] NVIDIA, "Optimized primitives for collective multi-gpu communication," https://github.com/NVIDIA/nccl.

[25] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania *et al.*, "Pytorch distributed: experiences on accelerating data parallel training," in *VLDB 2020*, 2020, pp. 3005–3018.

[26] K. Leetaru and P. A. Schrodt, "Gdelt: Global data on events, location, and tone, 1979–2012," in *ISA annual convention 2013*, vol. 2, no. 4. Citeseer, 2013, pp. 1–49.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[28] J. Bruck, C.-T. Ho, S. Kipnis, and D. Weathersby, "Efficient algorithms for all-to-all communications in multi-port message-passing systems," in *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, 1994, pp. 298–309.

[29] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "Aligraph: a comprehensive graph neural network platform," in *VLDB 2019*, 2019, pp. 2094–2105.

[30] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, "Distdgl: distributed graph neural network training for billion-scale graphs," in *IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3) 2020*. IEEE, 2020, pp. 36–44.

[31] D. Zheng, X. Song, C. Yang, D. LaSalle, Q. Su, M. Wang, C. Ma, and G. Karypis, "Distributed hybrid cpu and gpu training for graph neural networks on billion-scale graphs," in *ACM KDD 2022*, 2022, pp. 4582—-4591.

[32] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. Kalamkar, N. K. Ahmed, and S. Avancha, "Distgnn: Scalable distributed training for large-scale graph neural networks," in *IEEE SC 2021*, 2021, pp. 1–14.

[33] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu, "Dgcl: An efficient communication library for distributed gnn training," in *ACM EuroSys 2021*, 2021, pp. 130–144.

[34] C. Wang, D. Sun, and Y. Bai, "Pipad: Pipelined and parallel dynamic gnn training on gpus," in *ACM PPoPP 2023*, 2023.

[35] A. McCrabb, H. Nigatu, A. Getachew, and V. Bertacco, "Dygraph: a dynamic graph generator and benchmark suite," in *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) 2022*, 2022, pp. 1–8.

[36] B. Rozemberczki, P. Scherer, Y. He, G. Panagopoulos, A. Riedel, M. Astefanoaei, O. Kiss, F. Beres, G. López, N. Collignon *et al.*, "Pytorch geometric temporal: Spatiotemporal signal processing with neural machine learning models," in *ACM CIKM 2021*, 2021, pp. 4564–4573.

Yaqi Xia (yaqixia@whu.edu.cn) received his BS and MS degrees in Electrical Engineering from the Xidian University in 2018 and 2021, respectively. He is currently pursuing his Ph.D. in Computer Science at Wuhan University. His research interests are distributed deep learning model training and deployment, and graph neural network (GNN) optimization.

Zheng Zhang (zzhang3031@whu.edu.cn) received his B.S degree in Computer Science from School of Computer Science, Wuhan University in 2017. He is currently pursuing his Ph.D in Computer Science at Wuhan University. His research interests are distributed deep learning model training and deployment, DNN network optimization.

Donglin Yang (dongliny@nvidia.com) received his B.S. degree in Electrical Engineering from Sun Yat-sen University and his Ph.D. in the Computer Science Department at the University of North Carolina at Charlotte in 2022. He is currently a Deep Learning Software Engineer at NVIDIA, working on TensorFlow Core/XLA.

Chuang Hu (handc@whu.edu.cn) received his B.S and M.S. degrees in Computer Science from Wuhan University in 2013 and 2016. He received his Ph.D degree from the Hong Kong Polytechnic University in 2019. He is currently an Associate Researcher in the School of Computer Science at Wuhan University. His research interests include edge learning, federated le

Xiaobo Zhou (waynexzhou@um.edu.mo) received the B.S, M.S, and the Ph.D degrees in Computer Science from Nanjing University, in 1994, 1997, and 2000, respectively. He was a professor with the Department of Computer Science, University of Colorado, Colorado Springs. He is currently a distinguished professor in the State Key Laboratory of Internet of Things for Smart City & the Department of Computer and Information Sciences at University of Macau. His research interests include distributed systems, cloud computing, data centers, data parallel, distributed processing, and autonomic. He was the recipient of the NSF CAREER Award in 2009

Hongyang Chen (hongyang@zhejianglab.com) received the B.S. and M.S. degrees from Southwest Jiaotong University, Chengdu, China, in 2003 and 2006, respectively, and the Ph.D. degree from The University of Tokyo, Tokyo, Japan, in 2011. He is currently a Senior Research Expert with Zhejiang Lab. His research interests include data-driven intelli- gent systems, graph machine learning, big data mining, and intelligent computing. He is also an adjunct professor with Hangzhou Institute for Advanced Study, The University of Chinese Academy of Sciences, and Zhejiang University, China.

Dazhao Cheng (dcheng@whu.edu.cn) received his B.S and M.S degrees in Electrical Engineering from the Hefei University of Technology in 2006 and the University of Science and Technology of China in 2009. He received his Ph.D from the University of Colorado at Colorado Springs in 2016. He was an AP at the University of North Carolina at Charlotte in 2016-2020. He is currently a professor in the School of Computer Science at Wuhan University. His research interests include big data and cloud computing.

Redundancy-Free High-Performance Dynamic GNN Training with Hierarchical Pipeline Parallelism

Yaqi Xia
yaqixia@whu.edu.cn
School of Computer Science
Wuhan University

Zheng Zhang
zzhang3031@whu.edu.cn
School of Computer Science
Wuhan University

Hulin Wang
wonghulin@whu.edu.cn
School of Computer Science
Wuhan University

Donglin Yang
dongliny@nvidia.com
Nvidia Corporation

Xiaobo Zhou
waynexzhou@um.edu.mo
IOTSC, University of Macau
Macau S.A.R

Dazhao Cheng
dcheng@whu.edu.cn
School of Computer Science
Wuhan University

ABSTRACT

Temporal Graph Neural Networks (TGNNs) extend the success of Graph Neural Networks to dynamic graphs. Distributed TGNN training requires efficiently tackling temporal dependency, which often leads to excessive cross-device communication that generates significant redundant data. However, existing systems are unable to remove the redundancy in data reuse and transfer, and suffer from severe communication overhead in a distributed setting.

This paper presents Sven, an algorithm and system co-designed TGNN training library for the end-to-end performance optimization on multi-node multi-GPU systems. Exploiting dependency patterns of TGNN models and characteristics of dynamic graph datasets, we design redundancy-free data organization and load-balancing partitioning strategies that mitigate the redundant data communication and evenly partition dynamic graphs at the vertex level. Furthermore, we develop a hierarchical pipeline mechanism integrating data prefetching, micro-batch pipelining, and asynchronous pipelining to mitigate the communication overhead. As the first scaling study on the memory-based TGNNs training, experiments conducted on an HPC cluster of 64 GPUs show that Sven can achieve up to 1.7x-3.3x speedup over the state-of-art approaches and a factor of up to 5.26x communication efficiency improvement.

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**.

KEYWORDS

Distributed training, dynamic GNN, pipeline parallelism

ACM Reference Format:
Yaqi Xia, Zheng Zhang, Hulin Wang, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. 2023. Redundancy-Free High-Performance Dynamic GNN Training with Hierarchical Pipeline Parallelism. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23), June 16–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3588195.3592990

HPDC '23, June 16–23, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0155-9/23/06…$15.00
https://doi.org/10.1145/3588195.3592990

1 INTRODUCTION

Graphs, which are ubiquitous in various domains [1, 32], are efficient representations to encode read-world objects into relational data structures. The past few years have witnessed increasing interest in applying **G**raph **N**eural **N**etworks (GNNs) to graph data [6, 17, 21, 23, 25, 40], such as node classification [28], link prediction [53], and graph classification [10]. Recent breakthroughs in GNNs, e.g. GCN [17] and GraphSAGE[12], focus on static graphs, i.e., graphs with fixed nodes and edges. Graphs in many real-world applications, however, are more in line with dynamic graphs. The fundamental difference between static and dynamic graphs lies in the nature that dynamic graphs are constantly evolving, which means a dynamic graph provides additional temporal information. Recently proposed **T**emporal **G**raph **N**eural **N**etworks (TGNNs) such as TGN [35], APAN [49], and JODIE [18] learn both temporal and topological relationships simultaneously by integrating the ever-changing nature in the embedding information, which outperforms static GNNs on many downstream tasks [34, 56].

Motivation. Distributed GNN training over static graphs has been a major solution for handling large-scale datasets [9, 27, 43]. Richer information in dynamic graphs asks for massive parallel and distributed computation in processing TGNNs [56]. To capture temporal dependencies, many TGNNs exploit a module named *node memory and message* that summarizes the historical behavior of each node, namely memory-based TGNNs [18, 35, 49]. To scale TGNN training, temporal dependencies lead to two main performance issues. First, TGNN requires the latest temporal dependencies based on the vertices of all interactions, which results in rich redundant dependency data. Second, temporal dependencies and model parameters are distributed across devices in a large-scale GPU cluster. The training process requires collective operations such as *all-to-all* or *all-gather* to dispatch dependencies to the desired processors and aggregate them after updating. Furthermore, gradients of model parameters are synchronized through an *all-reduce* cross-device communication. In a distributed setting, the communication overhead leads to significant time consumption and limits the performance of TGNN training. Our measurement reports that the time proportion of communication can be more than 70% when training the JODIE model [18]. Thus, the performance

bottlenecks motivate us to reduce the communication volume by removing the redundant data as well as to mitigate the communication overhead by hierarchical pipeline parallelism.

Limitation of state-of-art approaches. Currently, well-known GNN systems like DGL [46], PyG [7], and AGL [52] are efficient and flexible so that developers can expeditiously deploy and implement their static GNN models for application development. Unfortunately, these frameworks have little or no extension to dynamic graphs. To provide efficient training primitives for dynamic graphs, several frameworks have been designed for training TGNNs. DGL only provides some naive interfaces to support dynamic graph training. However, the interfaces are inefficient and lack compatibility with distributed environments. TGL [56] is a framework for large-scale offline TGNN training which supports general TGNN variants training on one-node multi-GPUs. However, TGL maintains temporal dependencies on the host memory by a shared-memory mechanism, which limits itself in scaling out to a large-scale cluster. In addition, TGL suffers from severe communication overhead due to massive host-to-device memory copies. ESDG [4] proposes a graph difference-based algorithm to reduce the communication traffic and extend TGNN training to multi-node multi-GPU systems. However, ESDG only supports training with Discrete-time Dynamic Graphs (DTDGs), also known as snapshot-based graphs. As pointed out by TGN [35], Continuous-time Dynamic Graphs (CTDGs) are the most general formulation compared to others, such as Spatio-Temporal and DTDGs. Therefore, the schemes proposed in ESDG can not be extended to general TGNN model training.

Very recently, dependency-cached (DepCache) and dependency-communicated (DepComm) mechanisms [47] have been developed to maintain dependencies for distributed GNN training. However, we find that both DepCache and DepComm mechanisms are inefficient for TGNN training because of the heavy cross-device communication overhead. Our measurement reveals that the time consumption of communication can be up to 2.7x that of computation (the former takes 401.2 ms while the latter takes 146.5 ms within a mini-batch), which can result in trivial performance benefits with pipeline parallelism.

Key insights and contributions. We present Sven, an algorithm and system co-designed framework for high-performance distributed TGNN training on multi-node multi-GPU systems. Specifically, Sven is designed for TGNN training on CTDGs. We provide insight into the potential improvement in redundant dependencies and the margin between the time consumption of communication and computation. Instead of resolving these two challenges separately, we address them from a holistic perspective and in a collaborative manner.

We analyze the TGNN model behavior of handling temporal dependencies, which dispatches dependencies at the current state for computation and aggregates the latest dependencies after computation. Those models result in a tremendous volume of redundant data at the dispatch and aggregation stages because one vertex may interconnect with other vertices multiple times. We measure the redundancy ratio on various dynamic graph datasets and effectively exploit the potential communication reduction of temporal dependencies by extracting the redundancy-free graph from the original graph. To achieve balanced communication across various devices, we leverage the observed alternate pattern of high- and low-frequency vertices and propose a simple yet efficient strategy to partition a dynamic graph at the vertex level.

Furthermore, we augment DepCache and DepComm mechanisms [47] to maintain temporal dependencies for distributed TGNN training. We propose to reduce the communication volume by removing data redundancy, which mainly comes from the temporal dependency communication, including all-gather operation for DepCache and all-to-all operation for DepComm. To further mitigate the impact of communication overhead, we propose an adaptive micro-batch pipelining approach to reduce the system overhead incurred by the all-to-all operation and develop an asynchronous pipelining method to hide the all-reduce communication.

In summary, we highlight the contributions as follows.

- We conduct the in-depth characterization and the analysis of the communication volume of temporal dependencies for the distributed TGNN training, and reveal significant redundancy in cross-device dependency transfer and severe performance degradation of the state-of-the-art approaches.
- We present the algorithm and system co-design for distributed TGNN training on multi-node multi-GPU systems, featuring algorithm-level optimizations for efficient dependency handling and system-level components for end-to-end training performance improvement.
- We design and develop a set of innovative strategies and mechanisms for redundancy-free data organization, interval-based vertex-level graph partitioning, and hierarchical pipeline parallelism that holistically integrates data prefetching, micro-batch pipelining, and asynchronous pipelining.
- We implement the proposed innovations into a high-performance library for distributed TGNN training, namely Sven, and integrate Sven into PyTorch. To the best of our knowledge, Sven is the first scaling study specifically for temporal GNN training on Continuous-time Dynamic Graphs.

Experimental results in a 16-node 64-GPU HPC system demonstrate that Sven can achieve up to 5.26x communication efficiency improvement compared to DepComm and 1.7x-3.3x speedup over the state-of-art methods, i.e., 1.7x over DepComm, 1.8x over DepCache and 3.3x over TGL. Results also show Sven achieves up to 37x speedup in the system of 64 GPUs, demonstrating good scalability.

In the rest of this paper, Section 2 presents the background of TGNN training and motivation studies. Sections 3 and 4 describe the design and implementation of Sven, respectively. Section 5 presents the experimental setup, results, and analysis. Section 6 discusses the related work. Section 7 concludes the paper.

2 BACKGROUND AND MOTIVATION

2.1 Temporal Graph Neural Network

2.1.1 Dynamic GNN. GNN is a type of neural network which directly operates on the graph structure. A static GNN takes a fixed graph as input and encodes the graph data as node embeddings which are applied to downstream tasks, such as link prediction [28] and node classification [53]. In real-world scenarios such as biological interactomes and social networks, graphs tend to be dynamic and evolve over time. Recently, two representative models are proposed to describe the dynamic graphs, i.e., DTDGs and CTDGs. DTDGs consist of a series of static graphs with intercepted time

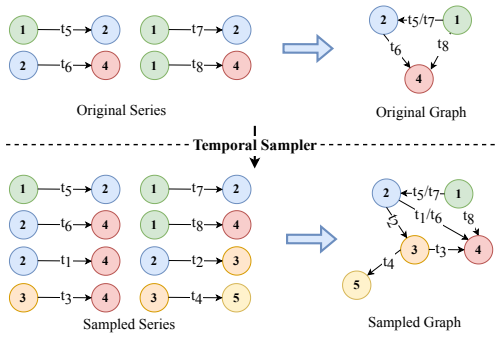


Figure 1: The dynamic graph sampling process.

intervals, while CTDGs capture continuous dynamics of temporal graph data. Since CTDGs are more general and flexible, this paper focuses on CTDGs. Dynamic GNNs have focused on two aspects of dynamicity, the graph structure, and the temporal dependencies. With these features, dynamic GNNs are able to combine techniques for structural information encoding with techniques for temporal information encoding.

2.1.2 Training process of dynamic GNN. In general, the training process of TGNN is composed of two parts, sampler and trainer.

Sampler. Similar to static graph sampling, the idea of dynamic graph sampling is to group the neighbor vertices of the target vertex. However, dynamic graph sampling is more complicated because it considers the timestamps of the neighbors. The temporal sampler needs to identify the candidate edges probabilistically from all past neighbors. As shown in Figure 1, the sampled series/sampled graph contains both the original series/graph and interactions that occurred in the past with its neighbors.

Trainer. There is an essential mechanism between static graph training and dynamic graph training, i.e., **N**ode memory and **M**essage (N&M). N&M summarizes the history of the vertices in the past, which provides enough information to generate the dynamic node embedding at the current state. Typically, a TGNN model consists of two components, a memory-update module, and a temporal GNN module. The former updates the N&M data by learning the temporal information of the input. The latter integrates the updated N&M and topological information of the graph to produce the node embeddings for downstream tasks.

Algorithm 1 describes the training process of one iteration for TGNN training. Table 1 summarizes important notations. In the mini-batch training, given an original graph G, sampled graph G', its corresponding features, and N&M, the key objective of the model is to encode each vertex into an embedding and utilize them for the prediction of downstream tasks.

First, a *mem* function is applied to refresh the behavior of vertices by summarizing the message and previous node memory (Line 2), which is denoted as the memory-update module. Commonly used memory-update modules are Gated Recurrent Units [5], Long Short-Term Memory [37], etc. Afterward, the model launches a *msg* function to combine the information of the latest interacting neighbors of the vertices to update its message, including timestamps,

Algorithm 1: Training process of TGNN of one iteration

input : original graph $G = \xi(t_s, t_e) = \{(v_i(t), v_j(t)) \| t \in (t_s, t_e)\}$, sampled graph $G' = \xi(t'_s, t'_e) = \{(v_i(t), v_j(t)) \| t \in (t'_s, t'_e)\}$, edge features $\{\boldsymbol{e} | e \in G'\}$, node features $\{\boldsymbol{v} | v \in G'\}$, node memory $\boldsymbol{S} = \{\boldsymbol{s}_i(t^-) | v_i \in G'\}$, model parameters $\boldsymbol{W}$

output : updated parameters $\boldsymbol{W}$

1 **for** $v_i(t) \in G' = \xi(t'_s, t'_e), t \leftarrow t'_s$ **to** t'_e **do**
2 $\boldsymbol{s}_i(t) = mem(\boldsymbol{m}_i(t^-), \boldsymbol{s}_i(t^-))$
3 $\boldsymbol{m}_i(t) = msg(\boldsymbol{s}_i(t), \boldsymbol{s}_j(t), t, \boldsymbol{e}_{i,j}(t))$
4 $\boldsymbol{z}_i(t) = emb(i, t) = emb(\boldsymbol{s}_i(t), \boldsymbol{s}_j(t), t, \boldsymbol{e}_{i,j}(t), \boldsymbol{v}_i, \boldsymbol{v}_j)$
5 **end**
6 **for** $v_i(t) \in G = \xi(t_s, t_e) = \{(v_i(t), v_j(t))\}, t \leftarrow t_s$ **to** t_e **do**
7 **for** $t_1, \ldots, t_b \leq t$ **do**
8 $\boldsymbol{s}_i(t) \leftarrow agg(\boldsymbol{s}_i(t_1), \ldots, \boldsymbol{s}_i(t_b))$
9 $\boldsymbol{m}_i(t) \leftarrow agg(\boldsymbol{m}_i(t_1), \ldots, \boldsymbol{s}_i(t_b))$
10 **end**
11 **end**
12 compute the loss according to specific task

Table 1: Notations.

Notation	Description
v_i	vertex i
$\boldsymbol{v}_i, \boldsymbol{e}_{ij}$	node feature of vertex i and edge feature of edge ij
$\boldsymbol{s}_i(t), \boldsymbol{m}_i(t)$	node memory and message of vertex i at time t
t_v^-	time when $\boldsymbol{s}_v$ is updated
$\xi(t_s, t_e)$	all time series between time t_s and time t_e

node memory, and edge features (Line 3). In the vast majority of cases, msg is the identity [35] function, which simply concatenates the inputs. After that, the temporal GNN module utilizes *emb* function to project feature information and memory information that is involved in the interaction events into the embedding space (Line 4). Then, TGNN updates the N&M with the result from the computation with an aggregation function *agg* (Lines 8 and 9). TGN first proposes two efficient aggregation functions: *most recent message*, which keeps only the most recent message for a given node, and *mean message*, which averages all messages for a given node. TGL demonstrates that there is no significant difference between these two methods, thus we follow TGL's approach and adopt the most recent message policy in this paper because of higher efficiency. Finally, an optimizer such as Stochastic Gradient Descent [33] is adopted to minimize the loss function with the temporal embedding $\boldsymbol{z}_i(t)$ of vertex i at timestamp t according to the downstream application for end-to-end training (Line 12).

2.2 Redundancies in Temporal Dependencies

In the training process, TGNN needs to dispatch N&Ms for computation and aggregate the latest N&Ms after computation due to the most recent policy. As shown in the Figure 2, in the dispatch phase, TGNN requires the latest N&M of corresponding vertices in

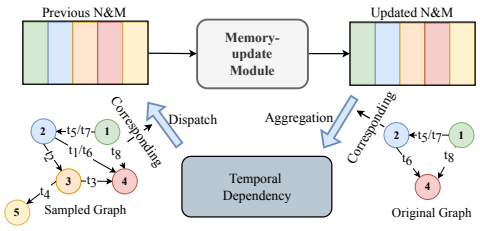


Figure 2: Workflow of the dispatch and aggregation.

sampled graph to calculate the updated N&M. For one mini-batch graph, the N&M of the same vertex can be reused multiple times. In the aggregation phase, TGNN aggregates all events involving the same vertex in the original graph. As indicated in line 3 of Algorithm 1, the result of the updated message is determined by the node memory of the combined source and target vertex. Therefore, among multiple events of the same vertex, if an interaction should be kept depends not only on whether it is the latest for this vertex but also on if that of the connected vertex.

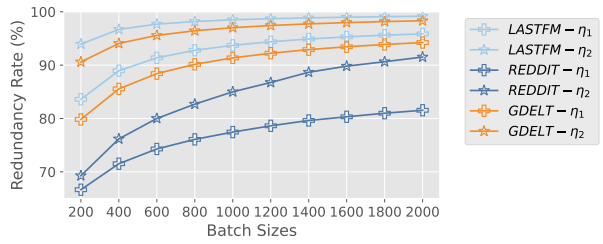
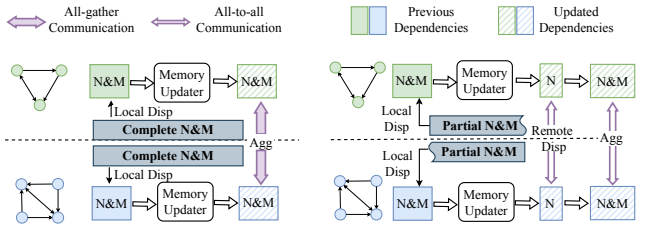


Figure 3: Redundancy rate of dataset LASTFM, REDDIT, and GDELT on various batch sizes. Each dataset corresponds to two lines, representing the dispatch (η_1) and aggregation (η_2) phases respectively.

Taking Figure 1 as an example, there are three interactions with respect to vertex 1 in the sampled series $((1-2,t_5),(1-2,t7),(1-4,t_8))$, TGNN models require three N&Ms of vertex 1 for computation. Thus, the memory-update module generates three updated N&M for vertex 1. Meanwhile, the most recent interaction event of the original series is required to update the N&M in the aggregation phase. Therefore, only the latest N&M corresponding to the vertices of the original graph is retained to update the previous N&M. In this example, among all interactions involved by vertex 1, $(1-4,t_8)$ are restored. However, other interactions related to vertex 1 may not be discarded, as they may be utilized by other adjacent vertex. $(1-2,t_7)$ is discarded by vertex 1 but is required by vertex 2. Following the above principle, N&M data continues to evolve during training. Therefore, there exist vast redundant data in the process of dispatching and aggregating the N&M dependencies. Specially, we define the redundancy rate of the former (i.e. dispatch) and latter (i.e. aggregation) as η_1 and η_2 respectively.

$$\eta_1 = \frac{N_1^r}{N_1}, \quad \eta_2 = \frac{N_2^r}{N_2} \tag{1}$$



(a) DepCache. **(b) DepComm.**

Figure 4: The workflows for the disp (dispatch) and agg (aggregation) stages for DepCache and DepComm. (a) The memory update phase with two trainers of DepCache. A gray block represents that each trainer caches a complete N&M. (b) The memory update phase with two trainers of DepComm. A gray block represents that each trainer maintains a disjoint N&M subset.

in which N_1 and N_2 are the numbers of sampled and original nodes respectively, and N_1^r and N_2^r are the number of duplicated nodes of sampled and original series respectively.

Figure 3 illustrates the detailed redundancy rate under various batch sizes and datasets. We can observe that the proportion of redundant nodes is quite large, i.e. more than 80% for most cases. Especially, the ratio η_1 is close to 100%, when the batch size for LASTFM is increased to 2000. With the increase in batch size, the redundancy rate also increases. The observed data redundancies result in poor training system performance. Therefore, we are motivated to design and develop a redundancy-aware strategy tailored for N&M dispatch and aggregation.

2.3 Cross-Device Communication Overhead

With the development of TGNN models, there is a trend that aims to distribute the training process across devices to accelerate the training process. State-of-the-art distributed GNN frameworks, such as Deep Graph Library (DGL) [46], only support static GNNs in distributed settings. The key challenge to distributing dynamic GNNs lies in efficiently dealing with N&M dependencies which keep evolving during training iterations.

Recently, NeutronStar [47] proposes an adaptive approach to partition and distribute vertex dependencies across multiple devices, which implements hybrid dependency management mechanisms including DepCache and DepComm. Here, we introduce the idea of DepCache and DepComm in TGNNs training. To support distributed training, both of them partition N&M dependencies among trainers but DepCache requires duplicating N&N locally. As illustrated in Figure 4a, the key idea of DepCache is that each trainer maintains a complete N&M so that dependencies are readily prepared locally in the dispatch phase (called local dispatch). After that, DepCache performs the forward/backward propagation within each process following the data-parallel scheme. However, in the aggregation phase, a trainer needs to collect updated N&Ms for all trainers to update the local N&M. Essentially, all-gather operation is required at this stage. In contrast to DepCache, DepComm

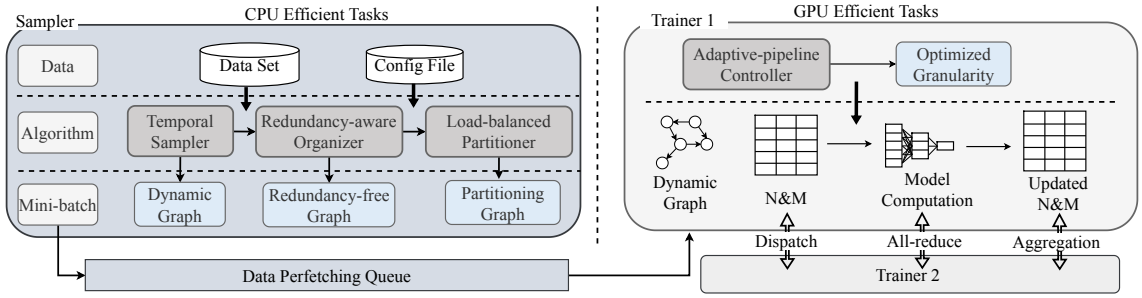


Figure 5: The system architecture of Sven, featuring algorithm-system co-design.

Table 2: Breakdown of DepCache and DepComm. A2A/AG Comm. represents all-to-all and all-gather communication for N&M dependencies. AR Comm. represents all-reduce communication for model parameter synchronization. Comp. represents model computation.

		Time (ms) / percentage(%)		
Method	model	A2A/AG Comm.	AR Comm.	Comp.
DepCache	TGN	129.5/31.1	144.5/34.6	143.1/34.3
	APAN	302.9/25.7	193.1/16.4	683.2/57.9
	JODIE	218.9/40.0	182.3/33.3	146.5/26.7
DepComm	TGN	102.5/26.5	137.2/35.4	147.8/38.1
	APAN	247.9/19.4	324.0/25.3	707.1/55.3
	JODIE	82.8/19.3	197.0/45.9	149.2/34.8

distributes independent N&M subsets across all trainers and collects the required N&M remotely as needed, which is illustrated in Figure 4b. DepComm firstly dispatches the N&M dependencies locally (i.e. local dispatch) and then dispatches updated node memory across trainers (i.e. remote dispatch) for subsequent embedding calculations. Since DepComm partitions the N&M dependencies into disjoint subsets among trainers, the remote dispatch requires all-to-all operation. Similarly, the aggregation phase requires another all-to-all operation after the forward/backward propagation. Finally, both DepCache and DepComm synchronize parameter gradients via all-reduce collective operation.

However, we find both DepCache and DepComm are inefficient due to cross-device communication overhead. We conduct experiments on various TGNNs to better understand the performance bottleneck of the two state-of-the-art mechanisms. In particular, we summarize the distributed training process as three phases, all-gather/all-to-all, all-reduce, and computation. Both DepCache and DepComm are evaluated on a 4-node 16-GPU cluster. We select three representative TGNN models: TGN [35], APAN [49], and JODIE [18] with dataset LASTFM [18]. The physical cluster and model configurations can be found in Section 5.1. Table 2 summarizes the time consumption of different phases for the two mechanisms.

From the result, it can be seen that the communication overhead of N&M dependencies is a dominant factor for DepCache performance. This is due to the transfer volume of all-gather being proportional to the number of trainers. As Table 2 shows, cross-device communication of N&M dependencies occupies a relatively considerable proportion, which ranges from 25.7% to 40.0%. Furthermore, since each trainer hosts the entire N&M dataset, the device has to consume more memory to store it. Subsequently, the risk of out-of-memory (OOM) increases dramatically under memory pressure circumstances. In contrast to DepCache, DepComm adopts model parallelism to tackle N&M dependencies, which is more memory-efficient compared to DepCache. However, the required all-to-all operation still causes significant overhead in the end-to-end training, which takes up to 26.5% of training time for TGN. In addition, both DepCache and DepComm suffer from all-reduce overhead, which accounts for up to 45.9% of the training time. Considering all these system overheads, it is necessary to reduce the impact from communication on training.

In summary, we are motivated by the experimental results and analysis that it is necessary to design a more efficient approach tailored for TGNNs training workload to minimize the transfer volume and mitigate the cross-device communication time consumption from a holistic perspective for improving training performance.

3 SVEN SYSTEM DESIGN

We design and develop Sven to facilitate the end-to-end TGNN training performance in a collaborative manner. Sven innovations are two-fold, minimizing data transfer volume via redundancy-free graph organization and intra-batch N&M reuse, and mitigating the system overhead among different operations through multi-level pipelining. Figure 5 presents the system architecture of Sven. Sven consists of algorithm-level optimization for efficient N&M dependencies handling and system-level components for coordinating the training process. At the algorithm level, it features a redundancy-free and load-balancing sampler that reduces and balances communication volume across partitions. At the system level, it features an efficient trainer that utilizes a hierarchical pipelining mechanism with adaptive tuning.

3.1 Redundancy-free & Load-balance Sampler

3.1.1 Redundancy-aware data organization. As illustrated in Section 2.2, there are plenty of overlaps among intra-batch dependencies at the dispatch and aggregation stages. By extracting the overlap topology among input graphs from the original series and sampled series, the redundant data transmission can be reduced. Due to the requirements of dispatch and aggregation of TGNN, the

Algorithm 2: Redundancy-free graph organization.

input : Input graph $G = \xi(t_s, t_e) = \{(v_i(t), v_j(t)) \| t \in (t_s, t_e)\}$, Output vertices $V_u = \{\emptyset\}$

output: Output vertices $V_u = \{v_{1'}, \ldots, v_{n'}\}$, location index for aggregation $\mathcal{In}$, location index for dispatch $\mathcal{Inr}$

Flattening input graph G into vector V_o, where $V_u = \{(v_i(t_s), v_j(t_s) \ldots, (v_m(t_e), v_n(t_e)\}$
for $v_i \leftarrow v_i(t_s)$ **to** $v_n(t_e)$ **do**
  $v_i \in \xi_j$, where j is the index of ξ_j in G
  if v_i *is not* $\in V$ **then**
    $V_u = V_u + \{v_i\}$
    $\mathcal{In} = \mathcal{In} + \{j\}$
  else
    replace the original index of v_i in $\mathcal{In}$ with $\{j\}$
  end
  find the index l of v_i in V_u, $\mathcal{Inr} = \mathcal{Inr} + \{l\}$
end

de-duplication algorithm should be tailored to the data dependency of the input graph, as we discussed in Section 2.2. Figure 6 illustrates the data organization for the dispatch and aggregation phases.

We propose a redundancy-aware algorithm to organize the extracted vertices IDs, which is illustrated in Algorithm 2. First, we flatten the interaction series of a graph into a vector according to the order of timestamps, as shown in the left column of Figure 1. Note that the vector V_o of the sampled graph in Figure 6a is converted at the vertex level while the vector of the original graph in Figure 6b is converted at the interaction level. The conversion is straightforward, so we utilize the former to represent them uniformly. Then, we traverse the vertices in vector V_o. For each vertex v_i, the algorithm first checks whether it has appeared. If not, we append it to the vertices set V_u. We locate the position of the interaction ξ corresponding to v_i in the $\xi(t_s, t_e)$ and record its index j in $\mathcal{In}$. If v_i is already contained in V_u, we use j to replace the index that appeared before. In this way, we filter out the redundant interaction in the aggregation phase, which is shown in the middle column of Figure 6b. Furthermore, we locate the index l of v_i in V_o and append l in $\mathcal{Inr}$. When the traversal is finished, V_u stores all the redundancy-free vertices. The input graph from the redundancy-free graph organization is required to be restored after the dispatch phase. Therefore, as shown in the middle column of Figure 6a, $\mathcal{Inr}$ is used to convert the input graph back to the state after the dispatch phase. In the other word, the redundancy-free strategy only removes redundant data during the communication process, while the input to the TGNN remained unchanged, ensuring that the output of the TGNN is equivalent. Given the lengths of vertices vector V_o and output vertices V_u, n and n' respectively, the complexity of Algorithm 2 is $O(nn')$.

3.1.2 Graph partitioning. While the GNN framework partitions input at the snapshot/graph level in training a static GNN model, Sven partitions a dynamic input graph at the vertex level. Since

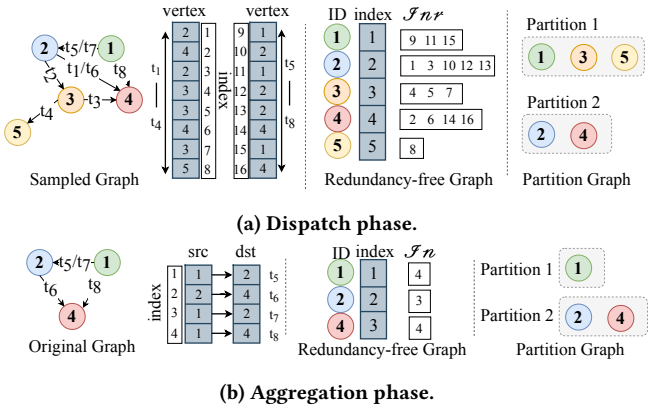


Figure 6: Redundancy-free and load-balanced data organization.

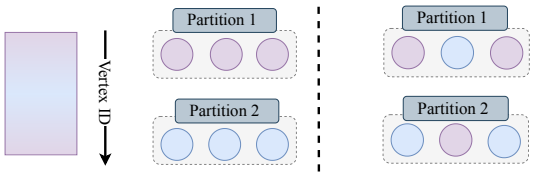


Figure 7: Partitioning a dynamic graph. A purple circle represents a hot vertex and a blue one represents a cold vertex.

an N&M dependency uniquely corresponds to a vertex, good vertex partitioning leads to balanced communication across trainers. State-of-art GNN frameworks DGL [46] and PowerGraph [11] provide range-based and random vertex-level partitioning methods for static graphs respectively. However, a dynamic graph structure usually follows skewed vertex distribution. To better understand the characteristics of vertices, we count the occurrence frequency of each vertex traversing through the time series in chronological order, which is presented as the heat map in Figure 7a. Since a certain number of vertices interact with other vertices multiple times within a period, the high-frequency vertices and low-frequency vertices are distributed alternately. Figure 7 shows two partitioning strategies with a dynamic graph at the vertex level. Figure 7a arranges all hot vertices into a partition and so do cold vertices. Figure 7b assigns hot and cold vertices into two partitions. We can observe that the partitioning strategy in Figure 7b is better than that in Figure 7a because the hot and cold vertices are distributed more evenly which results in balanced communication. Based on the above observation, we propose a simple yet efficient scheme tailored for dynamic graph datasets, named interval-based partitioning. Specifically, as shown in the right columns of Figure 6a and 6b, we split a redundancy-free graph as well as location index (i.e. $\mathcal{Inr}$ and $\mathcal{In}$) according to the vertex ID at a constant interval granularity. Sven's scheme ensures that hot and cold vertices in a graph reside across various devices uniformly, achieving load balancing of N&M dependencies.

3.1.3 Communication volume analysis. We analyze the incurred communication volume when accessing N&M dependencies for DepCache, DepComm, and Sven. To be fair, the proposed redundancy-aware data organization and graph partitioning are applied to DepComm and DepCache as well. We assume that each all-to-all or all-gather operation achieves ideal load-balancing.

We make the following definitions.

1. The number of sampled and original vertices is N_1 and N_2 respectively, and redundancy ratio of them is η_1 and η_2 respectively;
2. The unit memory size of data is C_0;
3. The size of cluster, i.e., the number of trainers is M.

We denote dim_msg, dim_mem, and dim_edge to represent the dimensions of the message, node memory, and edge feature, respectively. Thus,

$$dim\_msg = 2 * dim\_mem + dim\_edge. \tag{2}$$

Communication volume for DepCache. In DepCache, there is only one all-gather operation to aggregate N&M dependencies for each step. Therefore, the communication volume of DepCache becomes

$$C_{DepCache} = N_2 * (1-\eta_2) * (dim\_msg + dim\_mem) * \frac{2(M-1)^2}{M} * C_0 \tag{3}$$

Communication volume for DepComm. As for DepComm, since it dispatches the updated node memory, there is no redundant data. The first all-to-all communication volume becomes

$$C_{1,DepComm} = N_1 * dim\_mem * \frac{2(M-1)}{M} * C_0 \tag{4}$$

After that, the communication volume of the second all-to-all at the aggregation stage becomes

$$C_{2,DepComm} = N_2 * (1-\eta_2) * (dim\_msg + dim\_mem) * \frac{2(M-1)}{M} * C_0 \tag{5}$$

Thus, the total amount of communication traffic of DepComm is

$$C_{DepComm} = C_{1,DepComm} + C_{2,DepComm} \tag{6}$$

Communication volume for Sven. As for Sven, the communication volume of all-to-all at the dispatch phase is

$$C_{1,Sven} = N_1 * (1-\eta_1) * (dim\_msg + dim\_mem) * \frac{2(M-1)}{M} * C_0 \tag{7}$$

Then, we aggregate the updated N&M from other workers. Similarly, the second all-to-all communication volume is:

$$C_{2,Sven} = N_2 * (1-\eta_2) * (dim\_msg + dim\_mem) * \frac{2(M-1)}{M} * C_0 \tag{8}$$

Thus, the total amount of communication is:

$$C_{Sven} = C_{1,Sven} + C_{2,Sven} \tag{9}$$

Throughout the analysis, we draw the following conclusions.

1.It can be inferred from Equation 3 that, when the number of workers increases, deficiency of DepCache becomes more prominent due to its communication volume increasing proportionally with the number of trainers.

2.As shown in Equation 4, DepComm is more sensitive to the number of vertices in the sampled graph since it cannot efficiently tackle N&M dependencies at the dispatch phase.

3.Sven adopts the model-parallel scheme to tackle N&M dependencies. Its communication volume is independent of the number of trainers, per Equations 7 and 8. Besides, both the dispatch and aggregation phases benefit from the proposed redundancy-aware strategy.

Furthermore, we define the number of vertices of a redundancy-free graph at the dispatch phase as T times that at the aggregation phase. If the number of trainers is satisfied with the following condition,

$$M > T + 2 \tag{10}$$

we can conclude that condition $C_{Sven} < C_{DepCache}$ holds. Typically, T is less than two according to our measurement.

In addition, we define $P = dim\_mem/dim\_edge$, when the following condition holds, that is,

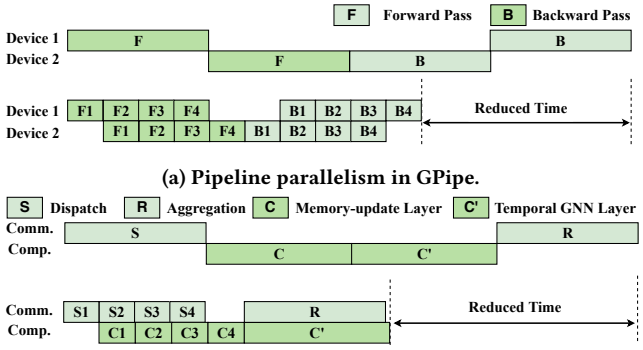
$$\eta_1 > \frac{2}{3} - P \tag{11}$$

we can derive $C_{Sven} < C_{DepComm}$. As shown in Figure 3, we can observe that the redundancy ratio η_1 is over $\frac{2}{3}$, therefore, Sven performs better than DepComm in the cases studied in this paper. In a nutshell, Sven eliminates the scalability bottleneck of DepCache but also reduces the redundant transfer incurred by DepComm at the dispatch phase.

3.2 Hierarchical Pipeline Parallelism

3.2.1 Prefetching. Sven deploys a prefetching mechanism to improve the efficiency of generating input graphs from the sampler to the trainer. Note that in heterogeneous GPU-CPU clusters, GPUs are responsible for the compute-intensive training tasks and CPUs are in charge of processing input batches. It can be noticed that the input batch data only has downstream consumers and are independent of each other. Sven develops a queue-based prefetch component. Concerning the sampler, Sven prepares mini-batch graph topology and applies the redundancy-free and vertex-level graph partitioning algorithms, after which the process graph structures are cached into the queue. Afterward, the trainer fetches the mini-batch from the cache queue directly without waiting for the processing stage. With the prefetch mechanism, the sampling stage can be performed in parallel with other stages, resulting in the reduction of end-to-end training time.

3.2.2 Adaptive micro-batch pipelining. As analyzed in section 2.3, the performance of TGNN models is limited by the required all-to-all operation for N&M dependencies during the dispatch and aggregation phases, which becomes a significant bottleneck when scaling out the training process. To mitigate the impact of the cross-device communication, an effective solution is pipeline parallelism, which is first introduced in GPipe [14]. As shown in Figure 8a, Gpipe partitions a mini-batch into smaller micro-batches, enabling GPUs to process multiple micro-batches concurrently while preserving the sequential dependency of the network. Inspired by GPipe, we introduce micro-batch parallelism in TGNN training to achieve end-to-end speedup. As presented at the top of Figure 8b, traditional data parallel training only allows for one mini-batch to be active for computation or communication at any given time. Especially, we separate the TGNN training process into four stages, dispatch, memory-update layer, temporal GNN layer, and aggregation. We split a micro-batch into several micro-batches and pipeline their execution one after the other. As shown in the bottom of Figure 8b, we



(a) Pipeline parallelism in GPipe.

(b) The proposed micro-batch pipeline parallelism

Figure 8: Pipeline parallelism in GPipe and Sven.

develop a micro-batch mechanism to dispatch communication and computation of memory update layers. Upon completing the first communication for a micro-batch, processors asynchronously execute computation while simultaneously starting to receive another mini-batch. Besides, the second communication starts as soon as the previous one is finished. Afterward, because there is no dependency between the aggregation communication and the calculation of temporal GNN layers, they can be executed simultaneously. Through the two-level pipeline parallelism, the majority of bubble time is eliminated in the training process.

Adaptive granularity configuration. The effectiveness of pipeline parallelism is largely determined by the granularity, i.e., the number of micro-batch partitions n. A coarse-grained granularity may fail to take the benefit of pipelining. On the other hand, an overly fine-grained granularity may lead to GPU underutilization. Therefore, it is necessary to search for the optimized n at runtime. As shown in Figure 3, we note that the redundancy ratio is approximately reciprocal to the batch size, and thus we define the function between the two variables as follows,

$$\eta = A + \frac{B}{bs} \tag{12}$$

where η and bs represent the redundancy ratio and the batch size respectively, and A and B are the scale factor and the offset which are determined by the characteristics of datasets. We consider that the transmission speed of system hardware is constant and we define it as S. Therefore, the optimized granularity is proportionally increasing as the transfer time increases. Based on the hypotheses and the communication volume formulation presented by Equations 7 and 8, the function mapping the relationship between the optimized solution and the batch size can be formulated as follows,

$$n = \lfloor (A * bs + B) * \frac{1}{s} \rfloor \tag{13}$$

With Equation 13, Sven adaptively tunes n for the optimized performance based on different model configurations and system hardware. We train the model for several iterations and collect the required data in Equation 13. Since the training process may endure thousands of iterations, the time consumption of profiling can be negligible. Note that splitting mini-batch (its size equals bs) into n micro-batches (its size equals m) does not affect the training convergence, because 1) the total number of batches per parameter update is unchanged, i.e. $m * n{=}bs$, and 2) differently from CNN models where BatchNorm operations are influenced by the batch size, GNN adopts LayerNorm instead, which is irrelevant to the batch size. As a result, though we split the mini-batch into several micro batches, the accumulated gradients per mini-batch are unchanged, achieving the same accuracy.

3.2.3 Asynchronous pipelining. Finally, we adopt asynchronous pipelining to alleviate the overhead caused by all-reduce operations. Inspired by PipeSGD [22], we deploy pipelined training with a width of two iterations taking advantage of both synchronous and asynchronous training. Sven proposes to introduce staleness to allow the backward propagation and optimization phases to be executed simultaneously with the all-reduce operation . To ensure the convergence of the model, Sven bounds the staleness to one step. Therefore, the weight updates can be expressed as follows,

$$\omega_{t+1} = \omega_t - \alpha \cdot \nabla f(\omega_{t-1}) \tag{14}$$

where ω_t contains the model weight parameters after t iterations, which is one for Sven, ∇f is the gradient function, α is the learning rate and ω_{t-1} is the weight used in iteration t. As pointed out by p^3 [9], a potential concern with applying unbounded stale gradient techniques is the negative impact on the convergence and accuracy of the network. However, the bounded delay eliminates the above issues and ensures identical model accuracy as that of data parallelism while significantly reducing the training time. We demonstrate the accuracy of Sven's asynchronous pipelining in Section 5.6.

4 IMPLEMENTATION

Sven is an end-to-end distributed TGNN training library implemented on top of PyTorch [30] 1.12.0 and DGL [46] 0.9.0 with CUDA 11.7 and NCCL [29] 2.10.3. Sven divides the workload into two processes: one process dispatches GPU computation and communication within the Trainer, while the other process manages CPU sampling and computation within the Sampler. We implement the hierarchical pipeline parallelism mechanism based on the communication package provided by PyTorch 'DistributedDataParallel' [20]. Specifically, samplers communicate with each other through the GLOO backend while trainers communicate with each other using the NCCL backend. The code of Sven will be open-sourced. A few key components and functionalities are implemented as follows.

4.1 The Sampler

We reused the temporal sampling module from the open-source project TGL, which is capable of sampling the original series and generating the sampled graphs. Following TGL, we create a 'DGLBlock' to represent a sampled graph. On top of the 'DGLBlock', we redefine a graph operator as 'rf_graph', which is an abstract combination of the redundancy-free graph organization and load-balanced partitioning strategy. Prior to each mini-batch training, it can communicate with all samplers in the cluster and automatically filter out redundant vertices, as well as partition the vertices in the temporal graph. The sampler obtains the vertex IDs corresponding to the redundant graph and the node IDs corresponding to the partitioned graph for each mini-batch. The preprocessed information

will be used for the Trainer to dispatch and aggregate remote N&M dependencies required for subsequent training.

4.2 The Trainer

Sven distributes N&M dependencies across GPUs in the model-parallel scheme while running the model computation part in the data-parallel scheme. NCCL [29] all-to-all collective operator is adopted to dispatch and aggregate N&M dependencies among GPUs. NCCL all-reduce collective operator is used to synchronize gradients before being used for weight updating. An individual CUDA stream is created to split the execution of the mini-batch into several micro-batch linear sequences that belong to a specific device and it is independent of other streams. As for the asynchronous pipelining mechanism, we override the vanilla all-reduce in 'DistributedDataParallel' [20] and register it with the 'communication hook' interface to achieve bounded staleness.

5 EVALUATION

5.1 Experimental Setup

5.1.1 Physical cluster. Our experiments are conducted on an HPC cluster with 16 nodes. Each node is equipped with 4 GPUs as well as 2 CPUs and has 128GiB RAM (shared by the 4 GPUs). Each GPU is NVIDIA Tesla V100 with 16GiB HBM and each CPU has 20 cores of 2.4GHz Intel Xeon E5-2640 v4. These nodes are connected by 56Gbps HDR Infiniband.

Table 3: Dataset characteristics. The timestamp represents the maximum edge timestamp while Dims represent the dimensions of edge features.

Dataset	Nodes	Edges	Timestamp	Dims
REDDIT	9K	157K	$2.7e^6$	172
LASTFM	2K	1.3M	$1.3e^8$	127
GDELT	17K	191M	$1.8e^5$	182

5.1.2 Datasets and TGNN models. Table 3 lists the major parameters of dynamic graph datasets that we used in the experiments. We utilize three datasets provided by TGL [56] in the evaluation. REDDIT [35] as well as LASTFM [18] are medium-scale and bipartite dynamic graphs. GDELT [19] is a large-scale dataset containing 0.2 billion edges. Furthermore, we select three representative TGNN models including APAN [49], JODIE [18], and TGN [35] for performance evaluation. All three variants are memory-based TGNN models. Specifically, TGN utilizes Gated Recurrent Units [5] as its memory-update module, while APAN uses Long Short-Term Memory [37] and JODIE uses Recurrent Neural Network [51]. The message size of the three models is set to 10 for high GPU utilization. The batch size is set to 1,200 by default. For all experiments, we adopt Adam [16] as the optimizer.

5.1.3 Methodology. We compare the performance of Sven with TGL [56], DepCache, and DepComm implementation. TGL is a distributed training framework for TGNNs aimed at single-node multi-GPU systems. It adopts the share-memory approach to store N&M in host memory [46], which relies on the GLOO backend for communication. However, cross-machine communication using GLOO backend is much slower than that of NCCL, which results in significant performance loss when scaling out the training process. Therefore, all measurements of TGL are conducted on the single-machine setup. To enable training TGNN with a large batch size, we adopt the 'random chunk scheduling' policy introduced by TGL for all methods.

5.2 Sven Performance Analysis

Firstly, we study the overall performance comparison with Sven and other state-of-the-art approaches and the scaling behavior of Sven.

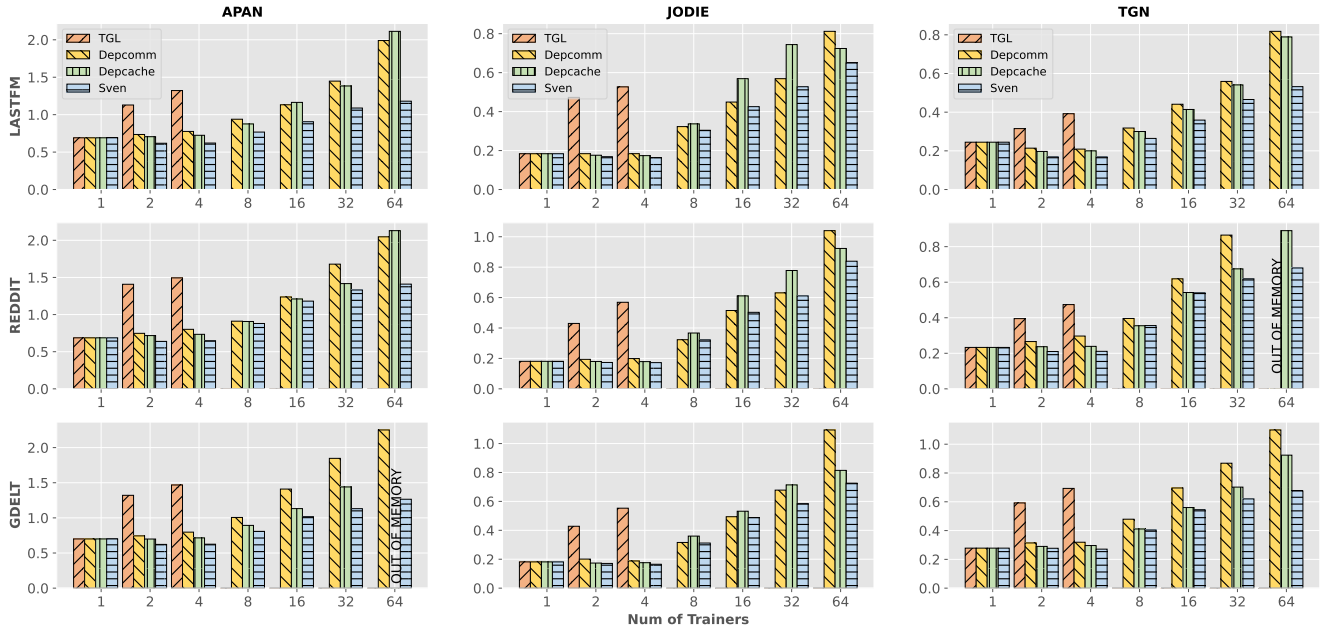
5.2.1 Performance comparison. Figure 9a presents the results of various model and dataset combinations. The y-axis in the histograms represents the average execution seconds of one iteration.

Compared to TGL. In a single-node setup, TGL performs the worst compared to other methods. Sven achieves up to 3.31x speedup against TGL. Since TGL caches N&M on the host memory, moving burdensome data dependencies from the CPU to the GPU through the PCIe bus introduces additional I/O overhead. In contrast, Sven, DepComm, and DepCache place N&M on the GPU to eliminate the overhead.

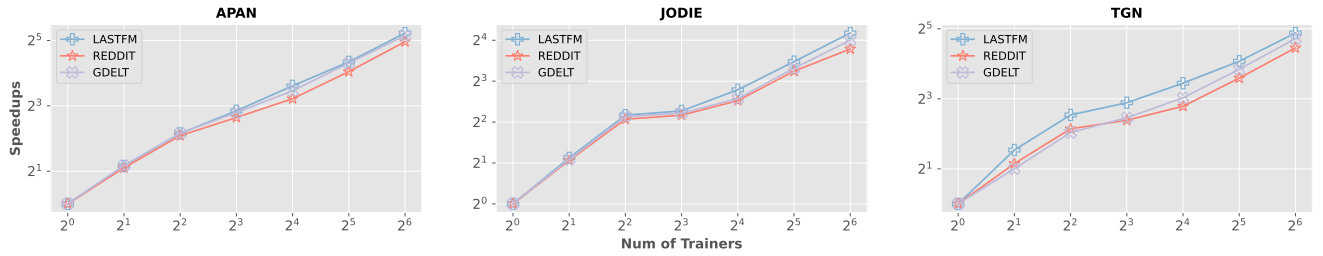
Compared to DepComm. Sven achieves up to 1.78x speedup against DepComm. The advantages of Sven are more prominent for those models with higher N&M dimensions, e.g. APAN, and datasets with higher redundancy ratios, e.g. LASTFM and GDELT. As revealed by Equation 11, if the dimension of N&M scales up and the repetition rate of the input graph increases, the discrepancy between the two sides of the inequality grows more pronounced, i.e., the disparity of communication cost between Sven and DepComm becomes prominent. The evaluation results are consistent with Equation 11, demonstrating the effectiveness of the proposed redundancy-aware strategy. Moreover, DepComm encounters the OOM (Out-of-memory) issue when training the TGN model with REDDIT datasets, which is caused by the data redundancy with the memory-update phase that cannot be handled by the redundancy-aware strategy.

Compared to DepCache. Sven can improve up to 1.80x speedup against DepCache. From the experimental results, with APAN and TGN models, the superiority of Sven over DepCache increases when scaling up the cluster. As demonstrated in Equation 3, DepCache is very sensitive to the size of the cluster, meaning that the communication volume of DepCache is proportional to the number of trainers. However, it should be noted that Sven only outperforms DepCache slightly on the JODIE model. This is because the JODIE model does not have a temporal sampler process, which greatly reduces the benefits of the redundancy-free approach. Furthermore, DepCache maintains a whole copy of N&M on each trainer, which makes it encounter the OOM problem when training the APAN model with the GDELT dataset.

5.2.2 Scaling study. Figure 9b summarizes the speedup curves for all datasets. Taking $M = 1$ as the reference point, the plot presents the speedup achieved as we increase the number of processors. For model APAN, the speedup is up to 37x, 31x, and 35x for datasets LASTFM, REDDIT, and GDELT respectively at $M = 64$, as against the ideal value of 64x. The best speed gain is up to 29x for model



(a) Overall performance comparisons (experimental results are shown for each data-model pair).



(b) Scalability of Sven (speedup on all datasets for different processors with $M = 1$ as the reference).

Figure 9: Comparisons of Sven with three state-of-the-art approaches in terms of training time and scaling speedup.

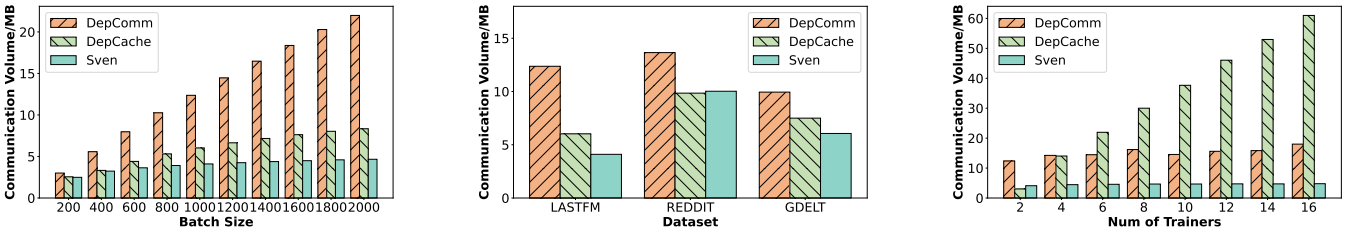
TGN at $M = 64$. The scalability of the model JODIE is relatively weak, as its maximum speedup is less than 20x. Compared with models APAN and TGN, JODIE does not require the process of sampling, which leads to a small number of sampled nodes. This greatly reduces its communication overhead for N&M. However, the time of model synchronization becomes the main bottleneck. In addition, we find that for various models, the speedup trends of different datasets are identical. LASTFM has the highest speedup, followed by GDELT, and REDDIT has the least. This is consistent with the order of redundancy rates for each dataset, illustrating the effectiveness of Sven's redundancy-aware strategy. Finally, compared to the scenario of $M = 4$, we observe a drop in speedup at $M = 8$. When $M > 4$, the cross-machine communication over the network is introduced, resulting in reduced speedup gains.

5.3 Communication efficiency analysis

As we discussed in section 3.1.3, the redundancy of dynamic graphs and the number of workers affect the communication volume of various methods. To study the communication efficiency among DepComm, DepCache, and Sven, we vary the batch size, datasets, and the number of workers. We set the default batch size, datasets, and the number of trainers to 1000, LASTFM, and 4, respectively.

5.3.1 Impact of batch size. We investigate how various batch sizes affect communication volume. As presented in Figure 10a, batch size has the most considerable impact on DepComm because DepComm cannot handle the burdensome N&M dependencies at the dispatch phase. It can also be derived from Equation 4 that the transfer increases linearly with batch size. DepCache is not sensitive to batch size compared to DepComm, because the redundancy-aware strategy can filter out redundant data at the aggregation phase. Note that Sven outperforms all schemes. Sven achieve communication efficiency from 1.25x to 5.26x compared to DepComm, and 1.16x to 2.01x compared to DepCache. The results demonstrate that Sven gains the maximum benefit from the redundant-aware strategy and it is more tolerant of the variation in batch size.

5.3.2 Impact of datasets. Figure 3 and Table 3 show that the redundancy ratio and dimension of edge features vary across different



(a) Impact of batch size. **(b) Impact of datasets.** **(c) Impact of trainers.**

Figure 10: The communication volume of DepComm, DepCache, and Sven.

datasets. Thus, we evaluate the effect of the two factors on communication volume. Figure 10b shows that Sven achieves an average communication efficiency of 2.09x and up to 3.30x compared to DepComm, and 1.23x and up to 1.47x of that compared to DepCache. Note that the results show the most prominent gap between DepComm and Sven with dataset LASTFM due to its highest redundancy ratio and smallest dimension of the edge feature. Similarly, we can observe the narrowest margin for REDDIT because of the fewest redundant vertices of REDDIT.

5.3.3 Impact of trainers. To evaluate the impact of the number of trainer processes on the communication volume, we vary the number of trainers for the LASTFM dataset, ranging from 2 to 16. Figure 10c shows Sven achieves an average of 2.29x and 6.11x communication efficiency against DepComm and DepCache, respectively. From Equation 4 to Equation 9, we know that the performance of both DepComm and Sven is independent of the number of trainers because both adopt model parallelism to handle the N&M dependencies. However, the communication volume of DepCache is proportional to the number of trainers. Compared to the results in Section 5.2.1, we notice that the reduction in communication volume does not result in a corresponding improvement in end-to-end performance, which is due to the fact that the time complexity of the all-gather communication is less than that of all-to-all communication [2].

In a nutshell, the analysis demonstrates that Sven removes redundant N&M dependencies and achieves the best scalability.

5.4 Sven performance breakdown

We discuss the performance breakdown of Sven's key components. We regard the naive data parallelism as the reference baseline, and continuously assemble individual components to examine performance improvement. Figure 11 shows the performance of TGNN models APAN, JODIE, and TGN using dataset LASTFM. The left y-axis represents the training time for one iteration and the right one represents the speedups against the naive implementation.

We integrate the redundancy-aware strategy to examine its performance benefit, which mainly achieves communication efficiency improvement. Compared to the naive baseline, the speedup is up to 1.42x, 1.14x, and 1.78x for training models APAN, JODIE, and TGN. As we analyzed in Section 5.2.1, the communication volume of model JODIE is insignificant compared to that of the other two TGNN models. The removal of the sampling module results in less traffic reduction after extracting redundant parts. Therefore, the redundancy-aware strategy yields the lowest benefit for JODIE. Furthermore, we study the effect of different vertex partitioning strategies. We adopt range-based partitioning as the default method. For models APAN, JODIE, and TGN, Sven's vertex-level partitioning strategy obtains 8.7%, 23.6%, and 28.7% improvement respectively, because it is more balanced and able to reduce the amount of N&M dependencies communication. We verify the effectiveness of the prefetching mechanism. Prefetching brings 1.31x, 1.45x, and 1.21x speedups on models APAN, JODIE, and TGN respectively. The results demonstrate that Sven can schedule CPU and GPU operations simultaneously and effectively hide the data preparation time. The next component is the adaptive micro-batch pipelining, in which we overlap the dispatch all-to-all communication with memory update layer computation and overlap aggregation all-to-all time with temporal GNN layer time. For models APAN, JODIE, and TGN, we get a performance boost of 20.5%, 25.3%, and 16.9% respectively after integrating the micro-batch component. Lastly, we examine the efficiency of the proposed asynchronous pipelining. For models APAN, JODIE, and TGN, the speed is increased by 6.1%, 14.4%, and 21.5% respectively.

Overall, the maximum speedup is close to 4x, and the minimum speedup is over 2.5x against the naive baseline. The result demonstrates the effectiveness of Sven's components.

5.5 Effectiveness of granularity configuration

We examine the effectiveness of the adaptive pipeline granularity configuration of Sven, which is based on a hypothesis that n is monotonically increasing as communication volume increases. Intuitively, when the batch size continues to increase, the all-to-all communication time will increase accordingly, which requires a fine-granularity configuration. We compare the performance with various batch sizes on the APAN-LASTFM model dataset pair. We regard the implementation without micro-batch pipelining as the baseline, in which n equals 1. Figure 12 shows that when the batch size is smaller than 800, $n = 2$ is the best option. When the batch size is between 800 and 2,000, $n = 3$ maintains the most impressive performance. When batch size is larger than 2,000, $n = 4$ becomes the optimal option. Sven performs the best in all situations.

5.6 Accuracy validation

We validate the prediction accuracy and study the impact of asynchronous pipelining. We evaluate the accuracy of TGL and Sven on dataset LASTFM within 4 GPUs (one physical node) for a fair

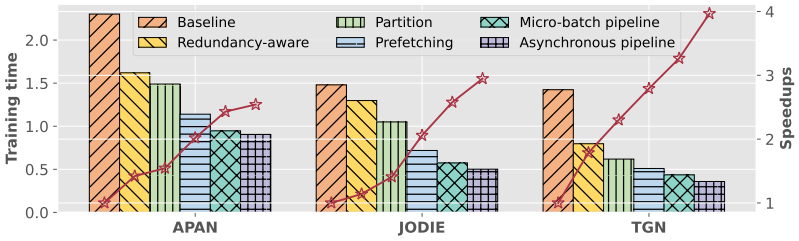


Figure 11: The overall performance breakdown of Sven.

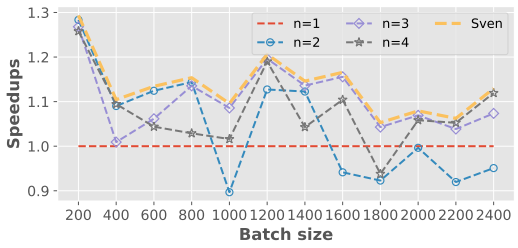


Figure 12: Effectiveness of adaptive granularity.

Table 4: Average precision for link prediction task.

	APAN	JODIE	TGN
TGL	80.19%	77.69%	88.42%
Sven	80.93%	78.12%	88.37%

comparison. Table 4 presents the accuracy comparison between TGL and Sven in the link prediction task. Following TGL, we adopt average precision to measure the accuracy on positive and negative test edges. We set the maximum training epoch to 100 and evaluate the accuracy of the model with the best performance on the validation set. We can observe that Sven achieves similar or higher precision than TGL among APAN, JODIE, and TGN. The experimental results demonstrate that Sven ensures the same accuracy as TGL does with asynchronous pipelining.

6 RELATED WORK

Systems for static GNNs. Several GNN systems are developed recently to support large-scale GNN training and tackle remote dependency communication [7, 8, 13, 15, 24, 31, 38, 39, 41, 42, 45, 46, 48, 50]. Distributed GNN systems such as AliGprah [57], DistDGL [54], DistDGL_v2 [55], and AGL [52] adopt dependency-cached approaches. They prepare dependencies data locally and work together with data parallel techniques. Frameworks such as DistGNN [27] and DGCL [3] exploit dependency-communicated schemes to transfer dependency data between processors.

Systems for temporal GNNs. Temporal GNNs can capture temporal information and topological structure, and thus outperform static GNNs in many applications [56]. Over the past few years, several research efforts have been made to achieve efficient end-to-end TGNNs training [4, 26, 44, 56]. Many of them [4, 26, 44] concentrate on the DTDGs, which represent a dynamic graph as a sequence of snapshots sampled at regular intervals. PiPAD [44] and ESDG [4] both recognize the topological similarity between snapshot graphs and extract overlapping parts to avoid unnecessary data transmission. However, the data in CTDGs is not discrete, so these methods are not applicable to more general scenarios. Besides, PiPAD utilizes pipelined and parallel mechanisms to execute computation and communication asynchronously, which is orthogonal to our hierarchical pipeline parallelism. DynaGraph [26] also discovers the reuse opportunity in DTDGs and proposes cached message-passing to reduce communication overhead, which is similar to our redundancy-free strategy. However, our method is capable of handling the situation where the topology of the graph keeps evolving. PyTorch Geometric Temporal [36] and TGL [56] are two general frameworks built on top of PyG [7] and DGL [46] to support TGNNs training. However, they provide limited support for extending dynamic graph training. TGL only supports single-machine multi-GPU scenarios and is unable to tackle the temporal dependency issue on distributed training systems. NeutronStar [47] analyzes two state-of-the-art designs for resolving the issue of vertex dependencies, namely DepCache, and DepComm. The idea behind DepCache is to cache dependencies locally and prepare the required dependencies before training. The key idea of DepComm is that dependencies are distributed among trainers and remote communication of dependencies is performed as needed. Instead of following DepCache to cache the complete N&M, Sven distributes the data dependencies across machines in a model-parallel scheme, which enables training large graphs. Compared with DepComm, which introduces two dispatch phases, including local and remote dispatch, Sven only requires one dispatch. This design can increase the data dependencies reusing possibility. Furthermore, we propose a general redundant data elimination strategy to reduce redundancy. To the best of our knowledge, Sven is the first scaling study specifically for temporal GNN training on CTDGs.

7 CONCLUSION

This paper presents Sven, a redundancy-free high-performance library for dynamic GNN training, which optimizes the end-to-end performance with hierarchical pipeline parallelism. With efficient redundancy-aware data organization and load-balancing partitioning strategies, Sven addresses the key challenges of excessive data transferring. Sven holistically integrates data prefetching, adaptive micro-batch pipelining parallelism, and asynchronous pipelining to tackle the communication overhead. Experimental results show that Sven achieves up to 1.7x-3.3x speedup, i.e., 1.7x over DepComm, 1.8x over DepCache and 3.3x over TGL, and 5.26x communication efficiency improvement compared to DepComm.

ACKNOWLEDGMENTS

Dazhao Cheng is the corresponding author. This work was supported by Zhejiang Lab Open Research Project (K-2022PI0AB01) and the Special Fund of Hubei Luojia Laboratory (220-100016). The numerical calculations in this paper have been done on the supercomputing system in the Supercomputing Center of Wuhan University.

REFERENCES

[1] Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, and Koray kavukcuoglu. 2016. Interaction networks for learning about objects, relations and physics. In *NeurIPS 2016*. 4509–4517.

[2] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, and Derrick Weathersby. 1994. Efficient algorithms for all-to-all communications in multi-port message-passing systems. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*. 298–309.

[3] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: An efficient communication library for distributed GNN training. In *ACM EuroSys 2021*. 130–144.

[4] Venkatesan T Chakaravarthy, Shivmaran S Pandian, Saurabh Raje, Yogish Sabharwal, Toyotaro Suzumura, and Shashanka Ubaru. 2021. Efficient scaling of dynamic graph neural networks. In *IEEE SC 2021*. 1–15.

[5] Kyunghyun Cho, Bart van Merriënboer, Çağlar Gulçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *EMNLP 2014*. 1724–1734.

[6] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In *NeurIPS 2016*. 3844–3852.

[7] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[8] Qiang Fu, Yuede Ji, and H Howie Huang. 2022. TLPGNN: A Lightweight Two-Level Parallelism Paradigm for Graph Neural Network Computation on GPU. In *ACM HPDC 2022*. 122–134.

[9] Swapnil Gandhi, Anand Padmanabha Iyer, Henry Xu, Theodoros Rekatsinas, Shivaram Venkataraman, Yuan Xie, Yufei Ding, Keval Vora, Ravi Netravali, Miryung Kim, et al. 2021. P3: Distributed deep graph learning at scale. In *USENIX OSDI 2021*. 551–568.

[10] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *ICML 2017*. PMLR, 1263–1272.

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

[12] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *NeurIPS 2017*. 1025–1035.

[13] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and bridging the gaps in current GNN performance optimizations. In *ACM PPoPP 2021*. 119–132.

[14] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. GPipe: efficient training of giant neural networks using pipeline parallelism. In *NeurIPS 2019*. 103–112.

[15] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *MLSys 2020*, 187–198.

[16] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

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

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

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

[20] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. PyTorch distributed: experiences on accelerating data parallel training. In *VLDB 2020*. 3005–3018.

[21] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[22] 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. In *NeurIPS 2018*. 8056–8067.

[23] Qi Liu, Maximilian Nickel, and Douwe Kiela. 2019. Hyperbolic graph neural networks. In *NeurIPS 2019*. 8230–8241.

[24] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. Neugraph: parallel deep neural network computation on large graphs. In *USENIX ATC 2019*. 443–457.

[25] Diego Marcheggiani and Ivan Titov. 2017. Encoding Sentences with Graph Convolutional Networks for Semantic Role Labeling. In *EMNLP 2017*. Association for Computational Linguistics, 1506–1515.

[26] Andrew McCrabb, Hellina Nigatu, Absalat Getachew, and Valeria Bertacco. 2022. DyGraph: a dynamic graph generator and benchmark suite. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) 2022*. 1–8.

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

[28] Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodola, Jan Svoboda, and Michael M Bronstein. 2017. Geometric deep learning on graphs and manifolds using mixture model cnns. In *IEEE CVPR 2017*. 5115–5124.

[29] NVIDIA. [n.d.]. Optimized primitives for collective multi-GPU communication. https://github.com/NVIDIA/nccl.

[30] 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. In *NeurIPS 2019*. 8026–8037.

[31] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. Sancus: sta le n ess-aware c omm u nication-avoiding full-graph decentralized training in large-scale graph neural networks. In *VLDB 2022*. 1937–1950.

[32] Siyuan Qi, Wenguan Wang, Baoxiong Jia, Jianbing Shen, and Song-Chun Zhu. 2018. Learning human-object interactions by graph parsing neural networks. In *ECCV 2018*. 401–417.

[33] Herbert Robbins and Sutton Monro. 1951. A stochastic approximation method. In *The annals of mathematical statistics 1951*. JSTOR, 400–407.

[34] Giulio Rossetti and Rémy Cazabet. 2018. Community discovery in dynamic networks: a survey. In *ACM computing surveys (CSUR) 2018*. 1–37.

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

[36] Benedek Rozemberczki, Paul Scherer, Yixuan He, George Panagopoulos, Alexander Riedel, Maria Astefanoaei, Oliver Kiss, Ferenc Beres, Guzmán López, Nicolas Collignon, et al. 2021. Pytorch geometric temporal: Spatiotemporal signal processing with neural machine learning models. In *ACM CIKM 2021*. 4564–4573.

[37] Jürgen Schmidhuber, Sepp Hochreiter, et al. 1997. Long short-term memory. In *Neural Comput 1997*. 1735–1780.

[38] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *USENIX OSDI 2021*.

[39] Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2020. Reducing communication in graph neural network training. In *IEEE SC 2020*. IEEE, 1–14.

[40] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2018. Graph attention networks. In *ICLR 2018*. 4.

[41] Keval Vora. 2019. LUMOS: dependency-driven disk-based graph processing. In *USENIX ATC 2019*. 429–442.

[42] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. 2017. Coral: Confined recovery in distributed asynchronous graph processing. In *ACM ASPLOS 2017*. ACM New York, NY, USA, 223–236.

[43] 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)*.

[44] Chunyang Wang, Desen Sun, and Yuebin Bai. 2023. PiPAD: Pipelined and Parallel Dynamic GNN Training on GPUs. In *ACM PPoPP 2023*.

[45] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. 2021. FlexGraph: a flexible and efficient distributed framework for GNN training. In *ACM EuroSys 2021*. 67–82.

[46] Minjie Yu Wang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds 2019*.

[47] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. Neutronstar: distributed GNN training with hybrid dependency management. In *ACM SIGMOD 2022*. 1301–1315.

[48] Shaoqi Wang, Oscar J Gonzalez, Xiaobo Zhou, Thomas Williams, Brian D Friedman, Martin Havemann, and Thomas Woo. 2020. An efficient and non-intrusive GPU scheduling framework for deep learning training systems. In *IEEE SC 2020*. IEEE, 1–13.

[49] 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 *ACM SIGMOD 2021*. 2628–2638.

[50] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *USENIX OSDI 2021*. 515–531.

[51] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

[52] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. 2020. AGL: a scalable system for industrial-purpose graph machine learning. In *VLDB 2020*. 3125–3137.

[53] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. In *NeurIPS 2018*. 5171–5181.

[54] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. Distdgl: distributed graph neural network training for billion-scale graphs. In *IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3) 2020*. IEEE, 36–44.

[55] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, Qidong Su, Minjie Wang, Chao Ma, and George Karypis. 2022. Distributed hybrid CPU and GPU training for graph neural networks on billion-scale graphs. In *ACM KDD 2022*. 4582—-4591.

[56] 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. In *VLDB 2022*.

[57] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: a comprehensive graph neural network platform. In *VLDB 2019*. 2094–2105.

Summary of changes

Dear Editor and Reviewers:

We would like to submit the revised version of our paper titled "Redundancy-free and load-balanced TGNN training with hierarchical pipeline parallelism." This revised version presents significant advancements and improvements over the conference version, which was accepted as the best paper candidate in the main program of the 32nd International Symposium on High-Performance Parallel and Distributed Computing.

The primary focus of the conference version was to introduce Sven, an algorithm and system co-designed TGNN training library for multi-node multi-GPU systems, aimed at reducing redundancy in data transfer and mitigating cross-device communication overhead. However, we identified an issue with the graph partitioning method employed, leading to communication imbalance across devices.

In this journal version, we have made the following key enhancements and extensions:

1. We conduct a detailed investigation of the communication imbalance issue among various devices caused by graph partitioning and propose a holistic solution to address it. First, we identify the imbalanced communication load of existing coarse-grained partitioning approaches. The performance results of these methods can be found in Figure 7. Then, we formulate the graph partitioning problem and prove that it is an NP-hard problem. The illustration of the problem and the proof of NP-hardness are presented in Figure 8 and Appendix A respectively. Additionally, we develop a novel approach called Re-FlexBiCut to approximate balanced minimum cuts by constructing source/sink graphs for each partition and to allow reassignment of vertices between partitions to improve balance. The detailed description and approximation ratio of Re-FlexBiCut can be found in Algorithm 3 and Appendix B respectively.
2. We add an implementation of mapping mechanism from global vertex IDs to their partition-local IDs to enable the proposed vertex-level graph partitioning. We establish robust mappings between vertices and their partition locations where dependency data is stored. The implementation uses a dual-hash mapping strategy with two hash functions, the establishment and utilization of dual-hash function can be found in Figure 10.
3. We added comparative experiments on communication balance under various batch sizes and numbers of trainers. The experiments results demonstrate that Re-FlexBiCut consistently outperforms other methods, reducing communication balance cost by up to 59.2%. The performance of Re-FlexBiCut compared with other baselines in terms of balance cost can be found in Figure 13. Moreover, we integrate Re-FlexBiCut into Sven and rerun the overall performance experiment and breakdown experienment. We find that Re-FlexBiCut can bring up to 1.41x speed improvement compared to baseline. The updated results can be found in subsection 5.2 and subsection 5.5.
4. The paper has been reorganized to accommodate the additional materials. We have moved the proofs of the theorems, derivations of communication volume, and micro-batch granularity

search to the appendix without affecting the logical flow of the paper. Additionally, revisions have been made to the Introduction section.

Overall, there are 40% or more new and important materials in this journal submission to TPDS.

Thank you for taking the time to review the manuscript.

Best regards,
Yaqi Xia, Zheng Zhang, Donglin Yang, Chuang Hu, Xiaobo Zhou, Hongyang Chen and Dazhao Cheng

APPENDIX A
PROOF FOR THEOREM 1

Theorem 1. *CBG-P problem is NP-hard for any $M \geq 4$.*

Proof: We show the CBG-P cut problem is NP-hard by reducing from the known NP-hard bisection problem[18]. In the bisection problem, we are given a graph G and number L, and asked if G's vertices can be partitioned into two equal-sized sets such that the cut between them has load capacity $\leq$ L. The reduction constructs a new graph H by connecting G to 4 new nodes a, b, c, d with specific capacity edges. It shows:

- If G has a bisection of capacity $\leq L$, we can create a 4-way cut of H with max capacity $\leq B$.
- If H has a 4-way cut with max capacity $\leq B$, the cut between c and d gives a bisection of G with capacity $\leq L$.

Therefore, if we could solve CBG-P cut on H in polynomial time, we could also solve bisection on G in polynomial time. Since bisection is NP-hard, CBG-P cut must also be NP-hard. □

APPENDIX B
PROOF FOR THEOREM 2

Theorem 2. *The approximation ratio for the Re-FlexBicut algorithm is $\mathcal{O}(\log^2 N)$.*

Proof: Let B' be the optimal load bound for the Re-FlexBiCut. Now consider an iteration of our algorithm with some guess value $B \geq B'$. We use binary search on B to find a near-optimal value. We first utilize the bisection[18] (α, ρ)-approximation subroutine to find cuts around each terminal. By definition of the bisection approximation ratio, each cut has capacity at most ρB. Next, we analyze the size of the cut returned by bisection. Let U be the set of unassigned nodes at the start of the iteration, and U' the remaining nodes after. We show that if a feasible Re-FlexBiCut exists for capacity B, then $|U'| \leq (1+\rho) - 1|U|$. This follows from the approximation ratio definition, which lower bounds the size of the cut output by bisection. Iteratively applying the binary search procedure, we have that in $\mathcal{O}(\log_{1+\beta} N)$ iterations, all nodes will be assigned if a feasible solution exists for the guess B. Therefore, our overall algorithm yields an $\mathcal{O}(\rho \log_{1+\beta} N)$ approximation for bisection. BiCut[19] give an improved $(\mathcal{O}(\log N), 1)$-approximation for bisection. Applying the (α, β) value, our algorithm achieves an $\mathcal{O}(\log^2 N)$ approximation. □

APPENDIX C
DERIVATION OF THE COMMUNICATION VOLUME FORMULA

We make the following definitions.

1. The number of sampled and original vertices is N_1 and N_2 respectively, and redundancy ratio of them is η_1 and η_2 respectively;
2. The unit memory size of data is C_0;
3. The size of cluster, i.e., the number of trainers is M.

We denote d_{msg}, d_{mem}, and d_{edge} to represent the dimensions of the message, node memory, and edge feature, respectively. Thus,

$$d_{msg} = 2 * d_{mem} + d_{edge}. \tag{14}$$

Communication volume for DepCache. In DepCache, there is only one all-gather operation to aggregate N&M dependencies for each step. Therefore, the communication volume of DepCache becomes

$$C_{DepCache} = N_2 * (1-\eta_2) * (d_{msg}+d_{mem}) * \frac{2(M-1)^2}{M} * C_0 \tag{15}$$

Communication volume for DepComm. As for DepComm, since it dispatches the updated node memory, there is no redundant data. The first all-to-all communication volume becomes

$$C_{1,DepComm} = N_1 * d_{mem} * \frac{2(M-1)}{M} * C_0 \tag{16}$$

After that, the communication volume of the second all-to-all at the aggregation stage becomes

$$C_{2,DepComm} = N_2 * (1-\eta_2) * (d_{msg}+d_{mem}) * \frac{2(M-1)}{M} * C_0 \tag{17}$$

Thus, the total amount of communication traffic of DepComm is

$$C_{DepComm} = C_{1,DepComm} + C_{2,DepComm} \tag{18}$$

Communication volume for Sven. As for Sven, the communication volume of all-to-all at the dispatch phase is

$$C_{1,Sven} = N_1 * (1-\eta_1) * (d_{msg} + d_{mem}) * \frac{2(M-1)}{M} * C_0 \tag{19}$$

Then, we aggregate the updated N&M from other workers. Similarly, the second all-to-all communication volume is:

$$C_{2,Sven} = N_2 * (1-\eta_2) * (d_{msg} + d_{mem}) * \frac{2(M-1)}{M} * C_0 \tag{20}$$

Thus, the total amount of communication is:

$$C_{Sven} = C_{1,Sven} + C_{2,Sven} \tag{21}$$

APPENDIX D
DERIVATION OF THE GRANULARITY FORMULA

As shown in Figure 3, we note that the redundancy ratio is approximately reciprocal to the batch size, and thus we define the function between the two variables as follows,

$$\eta = A + \frac{B}{bs} \tag{22}$$

where η and bs represent the redundancy ratio and the batch size respectively, and A and B are the scale factor and the offset which are determined by the characteristics of datasets. We consider that the transmission speed of system hardware is constant and we define it as S. Therefore, the optimized granularity is proportionally increasing as the transfer time increases. Based on the hypotheses and the communication volume formulation presented by Equations 9, the function mapping the relationship between the optimized solution and the batch size can be formulated as follows,

$$n = \lfloor (A * bs + B) * / S \rfloor \tag{23}$$

With Equation 12, Sven adaptively tunes n for the optimized performance based on different model configurations and system hardware.