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

Yaqi Xia, Zheng Zhang, Donglin Yang, Chuang Hu, Xiaobo Zhou, *Senior Member, IEEE*, Hongyang Chen, *Senior Member, IEEE*, Qianlong Sang, Dazhao Cheng, *Senior Member, IEEE*

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], [12]. However, scaling TGNN training introduces two primary performance issues. Firstly, TGNN requires the latest temporal dependencies based on the interaction of all vertices, 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* to ensure the continuously evolving of temporal 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 [13], and AGL [14] 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 [15] 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

- *Yaqi Xia, Zheng Zhang, Chuang Hu, Qianlong Sang and Dazhao Cheng are with the School of Computer Science, Wuhan University, Hubei 430072, China. (E-mail: {yaqixia, zzhang3031, handc, qlsang, dcheng}@whu.edu.cn.)*
- *Donglin Yang is at Nvidia Corp. (E-mail: dongliny@nvidia.com)*
- *Xiaobo Zhou is with IOTSC & Department of Computer and Information Sciences, University of Macau, Macau. (E-mail: waynexzhou@um.edu.mo.)*
- *Hongyang Chen is with the Research Center for Graph Computing, Zhejiang Lab, Hangzhou 311100, China. (E-mail: dr.h.chen@ieee.org.)*

(Corresponding author: Qianlong Sang, Dazhao Cheng.)

host-to-device memory transfers. ESDG [16] 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 [17] have been developed to maintain dependencies for distributed GNN training. However, our experimental results reveal that both DepCache and DepComm mechanisms are found to be inefficient for TGNN training due to the heavy cross-device communication overhead (see Table 2).

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 [18].

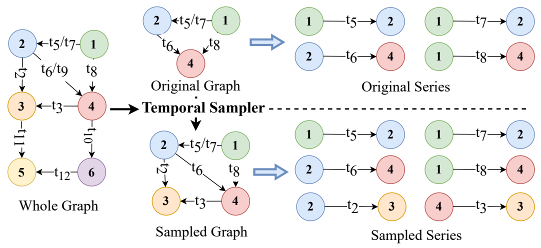


Fig. 1: The dynamic graph sampling process.

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.

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 TGL 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 Temporal 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

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
```

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

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 probabilistically identify the candidate edges from all past neighbors. As illustrated in Figure 1, vertex 2 is connected to vertex 3 at timestamp t_2. Since the timestamp for vertex 2 in the original graph is t_6, and t_2 precedes t_6, this connection is included in the sampled graph. Conversely, the connection between vertex 2 and vertex 4 at t_9 is not sampled since it occurs after t_6. Besides, 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. 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.

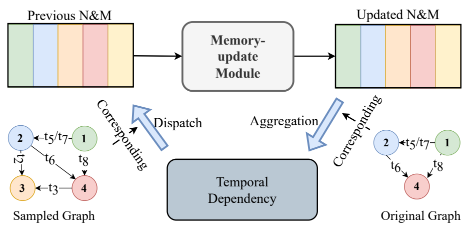


Fig. 2: Workflow of the dispatch and aggregation.

2.2 Redundancies in Temporal Dependencies

During the training process, TGNN performs two essential steps: dispatch and aggregation, illustrated in Figure 2. **(1) Dispatch phase:** TGNN fetches the latest N&Ms corresponding to the sampled graph from all temporal dependency, which is then used to train the model and calculate updated N&Ms. It's worth noting that for one mini-batch, the N&Ms of the same vertex can be reused multiple times. **(2) 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.,

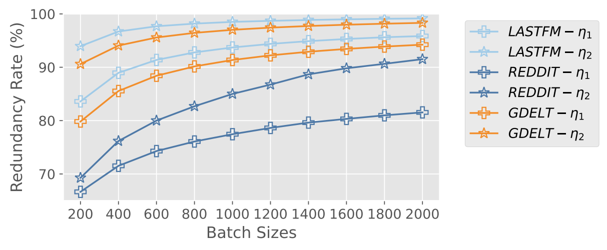


Fig. 3: Redundancy rate of the dispatch and aggregation phase.

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

$(1-4, t_8)$) 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.

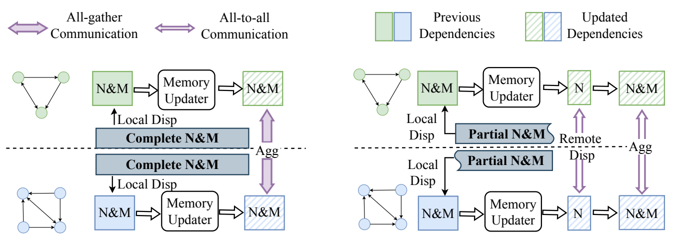
$$\eta_1 = \frac{N_1^r}{N_1}, \quad \eta_2 = \frac{N_2^r}{N_2} \tag{1}$$

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.

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



(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.

GNN frameworks, such as Deep Graph Library (DGL) [15], 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 [17] 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

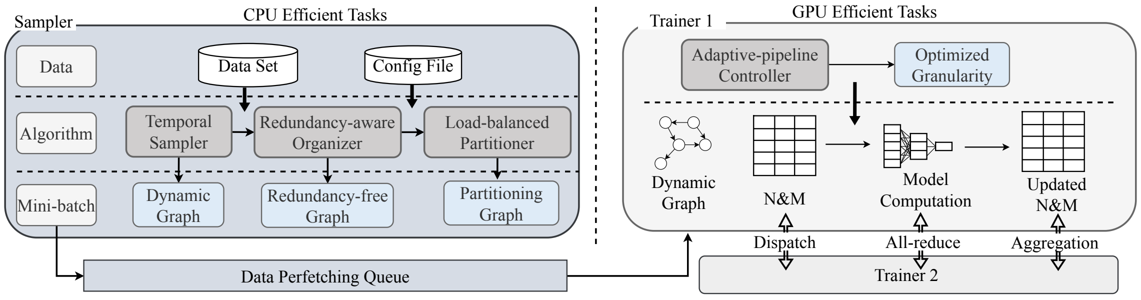


Fig. 5: The system architecture of Sven, featuring algorithm-system co-design.

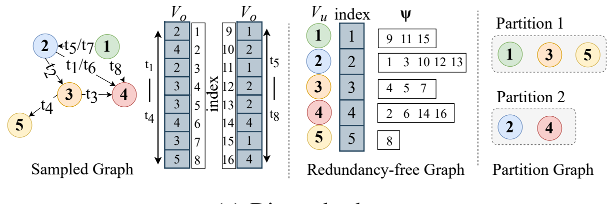
computation. Both DepCache and DepComm are evaluated on a 4-node 16-GPU cluster. We select three representative TGNN models: TGN [7], APAN [8], and JODIE [9] with dataset LASTFM [9]. The physical cluster and model configurations can be found in Section 5.1. Table 2 summarizes the time consumption of 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.

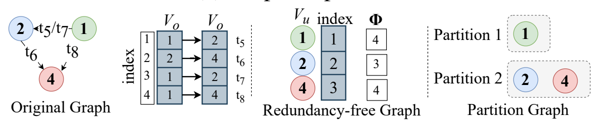
In summary, we are motivated by the experimental results and analysis that it is necessary to design a more efficient approach tailored for TGNNs training workload to minimize the transfer volume and mitigate the cross-device communication time consumption from a holistic perspective for improving training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



(a) Dispatch phase.



(b) Aggregation phase.

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

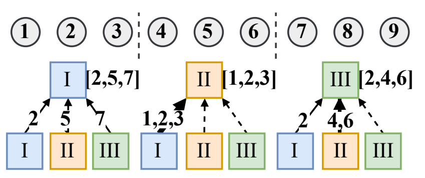
it features a redundancy-free and load-balancing sampler that reduces and balances communication volume across partitions. At the system level, it features an efficient trainer that utilizes a hierarchical pipelining mechanism with adaptive tuning.

3.1 Redundancy-free & Load-balance Sampler

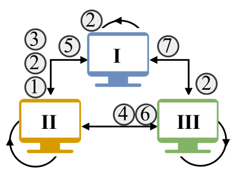
3.1.1 *redundancy-free data organization*

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.

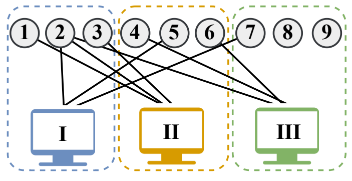
We propose a redundancy-free algorithm to organize the extracted vertices IDs, which is illustrated in Algorithm 2. First, we flatten the interaction series of a graph into a vector according to the order of timestamps. Note that the vector V_o of the sampled graph in Figure 6a is converted at the vertex 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 Φ. If v_i is already contained in V_u,



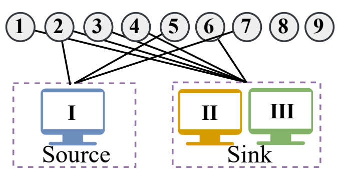
(a) Above is an illustration of vertex-level graph partitioning, and below is the communication topology for each trainer.



(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.

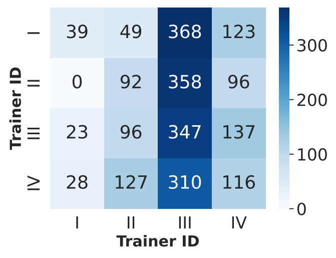
Fig. 7: A graph partition illustration with three trainers and nine vertices.

```
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 $V_u = \{v_{1'}, \dots, v_{N'}\}$, location
         index for aggregation $\Phi$, location index for
         dispatch $\Psi$
1  Flattening input graph $G$ into vector $V_o$, where
     $V_o = \{(v_i(t_s), v_j(t_s) \dots, (v_m(t_e), v_n(t_e)\}$
2  for $v_i \leftarrow v_i(t_s)$ to $v_n(t_e)$ do
3      $v_i \in \xi_j$, where $j$ is the index of $\xi_j$ in $G$
4      if $v_i$ is not $\in V$ then
5          $V_u = V_u + \{v_i\}$
6          $\Phi = \Phi + \{j\}$
7      else
8          replace the original index of $v_i$ in $\Phi$ with $\{j\}$
9      end
10     find the index $l$ of $v_i$ in $V_u$, $\Psi = \Psi + \{l\}$
11 end
```

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 $\mathcal{O}(NN')$.

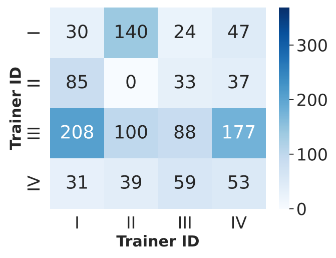
3.1.2 *Graph partitioning*

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 7a. Based on the requirements of different trainers for N&M, a com-



Trainer ID	I	II	III	IV
I	39	49	368	123
II	0	92	358	96
III	23	96	347	137
IV	28	127	310	116

(a) Rang-based partition.



Trainer ID	I	II	III	IV
I	30	140	24	47
II	85	0	33	37
III	208	100	88	177
IV	31	39	59	53

(b) Interval-based partition.

Fig. 8: Heatmap of the residual matrix with 4 trainers.

munication topology for the all-to-all operation is established, shown in Figure 7b. Since all-to-all communication requires synchronization across trainers, the overall communication overhead is determined by the slowest communication link.

Graph partitioning problem formulation

1)Problem formulation: Existing framework[15], [18] 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 7c. Assuming the graph is $G = (V, E)$, with a set of trainers $T = \{t_1, \dots, t_M\}$, a set of vertices $V' = \{v_1, \dots, v_N\}$, and a set of edges $E = \{e_1, \dots, e_P\}$, where T and V' satisfy $T \cup V' = V$. We partition the set V into subsets $W_1, \dots, W_M$ while ensuring that for $\forall i, j \in 1, \dots, M$:

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

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

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 $t_i$
1  Initialize sets $W_i = \{t_i\}$ containing just the trainer
2  Set $U = V \setminus (W_1 \cup W_2 \cup \ldots \cup W_M)$ as the set of
     unlabeled vertices
3  while $U \neq \emptyset$ do
4      for each trainer $t_i \in T$ do
5          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
6          Find approximate BiCut of $G'$, partitioning $U$
             into $U_i$ and $U_i'$ which can overlap.
7          $W_i = W_i \cup U_i$
8          Calculate the balance cost $\zeta(W)$
9          Reassign intersection vertices $U' = U_i \cap U_i'$ to
             sink, calculate new balance cost $\zeta'(W)$
10         if $\zeta(W) \geq \zeta'(W)$ then
11             $W_i = W_i \setminus U'$ and $U_i = U_i \setminus U'$
12         end
13         $U = U \setminus U_i$
14     end
15 end
```

We introduce the *balance cost* between various subgraphs, denoted as $\zeta(W)$, where $\zeta(W) = \max\{\zeta(W_j)|j = 1, \ldots, M\}$. $\zeta(W_j)$ 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 constraints 3and 4, 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[19], regarding pairings of trainers as binary splittings. Bisection cut problems can be approximately solved in polynomial time using existing algorithms like BiCut [20]. 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 7d.

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 invokes BiCut to find a cut surrounding the source and sink nodes. This cut partitions all unassigned vertices into U_i and U_i', where U_i and U_i' are not disjoint. (Line 6), which run in $\mathcal{O}(N^2 \log^2 N)$ time. Then it adds U_i's side vertices to subgraph W_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. Thus, the complexity of Re-FlexBiCut is $\mathcal{O}(M(N^2 \log^2 N + P))$.

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.

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.

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

Communication volume for DepComm.

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

Communication volume for Sven.

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

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,

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

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

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

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

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.

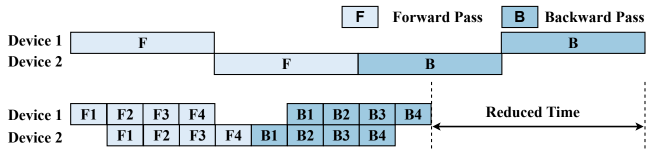
3.3 Hierarchical Pipeline Parallelism

3.3.1 *Prefetching*

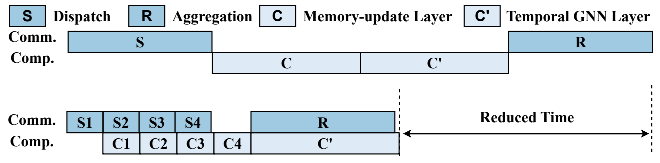
Sven incorporates a prefetch mechanism to conceal the overhead of generating batch data. Note that in heterogeneous GPU-CPU clusters, GPUs are responsible for the compute-intensive training tasks and CPUs are in charge of processing input batches. It can be noticed that the input batch data only has 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.

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 [21], 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



(a) Pipeline parallelism in GPipe.



(b) The proposed micro-batch pipeline parallelism

Fig. 9: Pipeline parallelism in GPipe and Sven.

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 the batch size is represented by bs, 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 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 [22], 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 [23], 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 [24] 1.12.0 and DGL [15] 0.9.0 with CUDA 11.7 and NCCL [25] 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

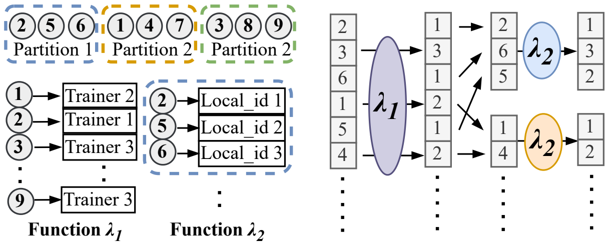
We implement the hierarchical pipeline parallelism mechanism based on the communication package provided by PyTorch 'DistributedDataParallel' [26]. NCCL [25] all-to-all collective operator is adopted to dispatch and aggregate N&M dependencies among GPUs. NCCL all-reduce collective operator is used to 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' [26] 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.



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

Fig. 10: Implementation of dual-hash mapping.

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	11K	672K	$2.7e^6$	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 [27] 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 [28] as the optimizer.

5.1.3 Methodology.

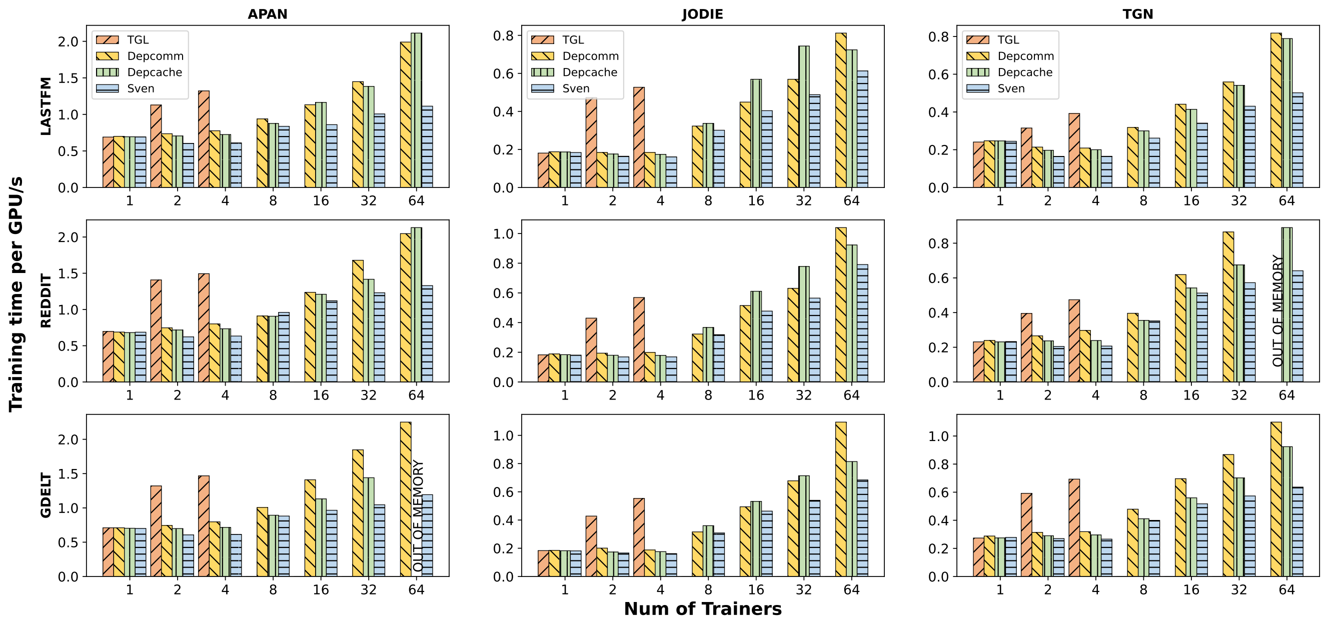
We compare the performance of Sven with TGL [10], DepCache, and DepComm implementation. TGL is a distributed training framework for TGNNs aimed at single-node multi-GPU systems. It adopts the share-memory approach to store N&M in host memory [15], which relies on the GLOO backend for communication. However, 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

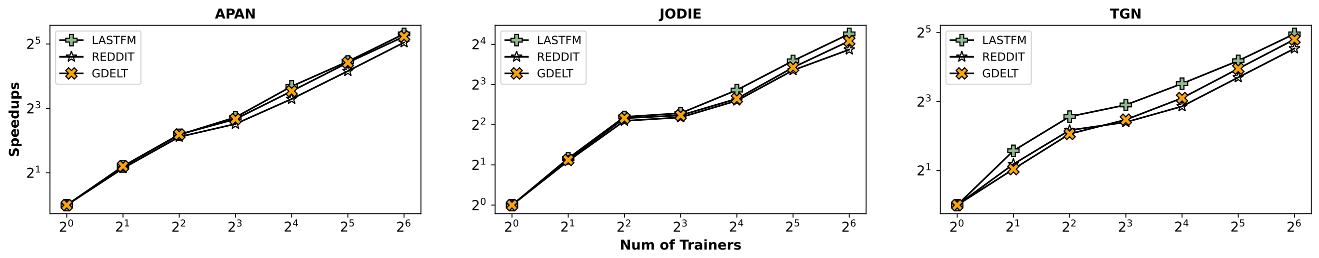
Firstly, we study the overall performance comparison with Sven and other state-of-the-art approaches and the scaling behavior of Sven.

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 time of one iteration per GPU.



(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).

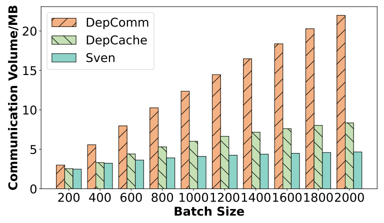
Fig. 11: Comparisons of Sven with three state-of-the-art approaches in terms of training time and scaling speedup.

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.

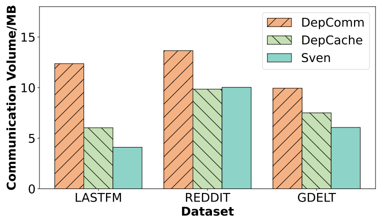
Compared to DepComm. Sven achieves up to 1.89x speedup against DepComm. The advantages of Sven are more prominent for those models with higher N&M dimensions, e.g. APAN, and datasets with higher redundancy ratios, e.g. LASTFM and GDELT. As revealed by Equation 11, if the dimension of N&M scales up and 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.

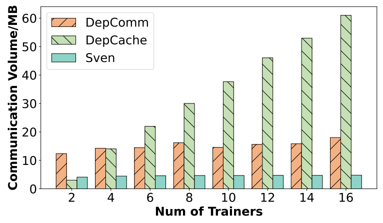
Compared to DistTGL. DistTGL [29] is another efficient and scalable approach for training TGNNs on distributed GPU clusters. In this section, we compare the average training time per GPU for Sven and DistTGL under different GPU setups. Both use the TGN model and the LASTFM dataset, and the results are shown in the Figure 13. Our results show that Sven achieves a 1.1x average speedup over DistTGL, with a particularly notable 1.3x speedup at configurations of up to 4 GPUs. This improvement is mainly attributed to DistTGL's reliance on host memory for storing N&M, which introduces significant PCIe transfer overhead from CPU to



(a) Impact of batch size.



(b) Impact of datasets.



(c) Impact of trainers.

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

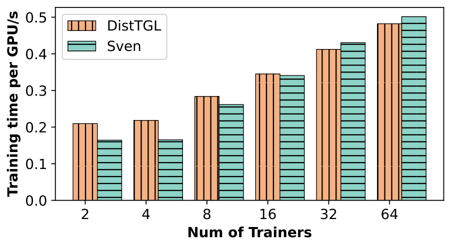


Fig. 13: Overall performance comparisons with DistTGL.

GPU. However, as the number of GPUs increases, DistTGL demonstrates its scalability advantages. Specifically, at 32 and 64 GPU settings, DistTGL outperforms Sven in terms of training efficiency. This is because DistTGL ensures that at least a complete copy of N&M data is maintained on each machine, effectively avoiding the overhead of cross-machine N&M data transmission.

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, 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*

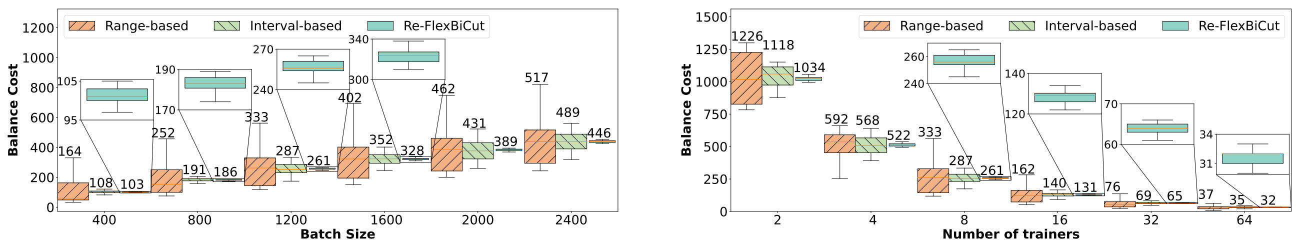
We investigate how various batch sizes affect communication volume. As presented in Figure 12a, batch size has the most considerable impact on DepComm because DepComm cannot handle the burdensome N&M dependencies at the dispatch phase. It can also be derived from Equation 8 that the transfer increases 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 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



(a) Batch size sensitivity

(b) Number of trainers sensitivity

Fig. 14: The communication balance cost of range-based, interval-based and Re-FlexBiCut.

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 [30].

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 2.

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.

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 15 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

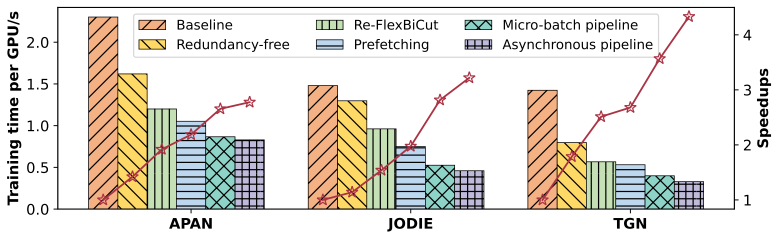


Fig. 15: The overall performance breakdown of Sven.

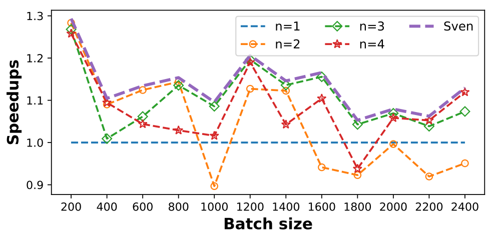


Fig. 16: Effectiveness of adaptive granularity.

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%

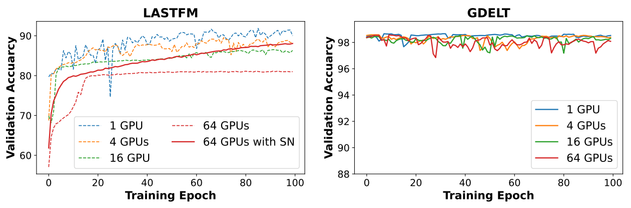


Fig. 17: Validation accuracy across various GPUs setting.

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 16 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.

5.7 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). In this section, we set the batch size to 600, consistent with the original TGL [10]. 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.

Next, we test the model accuracy under different GPU setups. We use the TGN model and the LASTFM and GDELT datasets for testing, ranging from 1 GPU to 64 GPUs. The model accuracy convergence curves are shown in Figure 17. In the TGL paper, the random chunk scheduling technique was introduced to enable larger batch sizes for TGNNs, a method we have also adopted in Sven. Our experiments show that this technique performs better on a large dataset, i.e., GDELT. As the number of GPUs increases from 1 to 64, the validation accuracy does not drop significantly, indicating a negligible loss in accuracy. However, on smaller datasets like LASTFM, we observed a noticeable decrease in accuracy with increasing GPU count, particularly at 64 GPUs. Inspired by DistTGL, we have implemented a strategy of incorporating static node memory to enhance model accuracy. We adopted the method described in [31] to obtain pre-trained static node memory. By incorporating static node memory, we found a significant improvement in model accuracy at 64 GPUs (shown as "64 GPU with SN" in the legend, where SN is the abbreviation for static node memory), bringing it on par with the accuracy achieved at 4 GPUs. By using this method of additionally incorporating static node memory, we can scale TGNNs training to larger computational clusters while ensuring acceptable model accuracy.

6 Related Work

Systems for static GNNs. Over recent years, several frameworks have focused on scaling GNN training. PaGraph [32], GNNLab [33], and BGL [34] adopt different caching strategies to accelerate multi-GPU feature retrieval for GNNs. DSP [35] employs a collective sampling primitive to enable efficient multi-GPU sampling. Distributed GNN systems such as AliGprah [36], DistDGL [37], DistDGL_v2 [38], and Euler [39] adopt dependency-cached approaches. They prepare dependencies data locally and work together with data parallel techniques. Frameworks such as DistGNN [40], DGCL [41], ROC [42] and Dorylus [43] 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], [44]. 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 [45] and ESDG [16] 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 [46] 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 [47] and TGL [10] are two general frameworks built on top of PyG [13] and DGL [15] to support TGNNs training. However, they provide limited support for extending dynamic graph training. For instance, TGL is constrained to single-machine, multi-GPU environments and fails to address temporal dependency issues in distributed settings. The recently proposed DistTGL [29] enhances TGL for multi-machine training by introducing epoch parallelism and memory parallelism techniques, thereby reducing cross-trainer communication overhead and improving training stability. Nevertheless, DistTGL does not resolve the redundancy in updating temporal dependencies during training, nor does it address the graph partitioning challenge, resulting in increased memory requirements. Furthermore, DistTGL's reliance on pre-storing mini-batches before training introduces non-trivial preprocessing time. In contrast, Sven effectively tackles these challenges through the integrated design of algorithm and system. Algorithmically, Sven minimizes and balances the communication volume of N&M synchronization by employing redundancy-free data organization and ReFlexBiCut graph partitioning. Systematically, it introduces hierarchical pipeline parallelism to alleviate the impact of mini-batch data generation and communication.

NeutronStar [17] 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.

Acknowledgement

This work was supported by the National Key Research and Development Program of China (2023YFE0205700), National Natural Science Foundation of China (62302348, 62341410), Fundamental Research Funds for the Central Universities (2042023kf0132), General Program of Hubei Provincial Natural Science Foundation of China (2023AFB831), Special Fund of Hubei Luojia Laboratory (220100016) and the Science and Technology Development Fund (FDCT) Macau SAR (File no. 0078/2023/AMJ). The numerical calculations in this paper have been done on the supercomputing system in the Supercomputing Center of Wuhan University.

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, 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, 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, pp. 5115–5124.

[5] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," in *NeurIPS*, 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*. 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, pp. 2628–2638.

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

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

[12] Z. Zhang, Y. Xia, H. Wang, D. Yang, C. Hu, X. Zhou, and D. Cheng, "Mpmoe: Memory efficient moe for pre-trained models with adaptive pipeline parallelism," *IEEE TPDS*, 2024.

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

[14] 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, pp. 3125–3137.

[15] 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.

[16] 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, pp. 1–15.

[17] 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, pp. 1301–1315.

[18] 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, pp. 17–30.

[19] 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.

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

[21] 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, pp. 103–112.

[22] 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, pp. 8056–8067.

[23] 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, pp. 551–568.

[24] 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, pp. 8026–8037.

[25] NVIDIA, “Optimized primitives for collective multi-gpu communication,” https://github.com/NVIDIA/nccl.

[26] 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, pp. 3005–3018.

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

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

[29] H. Zhou, D. Zheng, X. Song, G. Karypis, and V. Prasanna, “Disttgl: Distributed memory-based temporal graph neural network training,” in *IEEE SC*, 2023, pp. 1–12.

[30] 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.

[31] M. S. Hussain, M. J. Zaki, and D. Subramanian, “Global self-attention as a replacement for graph convolution,” in *ACM SIGKDD*, 2022, pp. 655–665.

[32] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, “Pagraph: Scaling gnn training on large graphs via computation-aware caching,” in *ACM SoCC*, 2020, pp. 401–415.

[33] J. Yang, D. Tang, X. Song, L. Wang, Q. Yin, R. Chen, W. Yu, and J. Zhou, “Gnnlab: a factored system for sample-based gnn training over gpus,” in *ACM EuroSys*, 2022, pp. 417–434.

[34] T. Liu, Y. Chen, D. Li, C. Wu, Y. Zhu, J. He, Y. Peng, H. Chen, H. Chen, and C. Guo, “Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing,” in *USENIX NSDI*, 2023, pp. 103–118.

[35] Z. Cai, Q. Zhou, X. Yan, D. Zheng, X. Song, C. Zheng, J. Cheng, and G. Karypis, “Dsp: Efficient gnn training with multiple gpus,” in *ACM PPoPP*, 2023, pp. 392–404.

[36] 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, pp. 2094–2105.

[37] 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)*. IEEE, 2020, pp. 36–44.

[38] 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, pp. 4582—-4591.

[39] Alibaba, “Euler,” https://github.com/alibaba/euler, 2020.

[40] 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, pp. 1–14.

[41] 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, pp. 130–144.

[42] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, “Improving the accuracy, scalability, and performance of graph neural networks with roc,” 2020, pp. 187–198.

[43] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, K. Vora, R. Netravali, M. Kim, and G. H. Xu, “Dorylus: Affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads,” in *USENIX OSDI*, 2021.

[44] H. Wang, D. Yang, Y. Xia, Z. Zhang, Q. Wang, J. Fan, X. Zhou, and D. Cheng, “Raptor-t: A fused and memory-efficient sparse transformer for long and variable-length sequences,” *IEEE TC*, 2024.

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

[46] 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, pp. 1–8.

[47] 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, 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.

Qianlong Sang (qlsang@whu.edu.cn) received his BS degree in Cyber Science and Engineering from Wuhan University in 2022. He is currently pursuing his Ph.D. in Computer Science at Wuhan University. His research interests include edge computing and big data systems.

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.