



# BLAD: Adaptive Load Balanced Scheduling and Operator Overlap Pipeline For Accelerating The Dynamic GNN Training

Kaihua Fu
midway@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Quan Chen*
chen-quan@cs.sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Yuzhuo Yang
yang-yz@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Jiuchen Shi
shijiuchen@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Chao Li
lichao@cs.sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Minyi Guo*
guo-my@cs.sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

## ABSTRACT

Dynamic graph networks are widely used for learning time-evolving graphs, but prior work on training these networks is inefficient due to communication overhead, long synchronization, and poor resource usage. Our investigation shows that communication and synchronization can be reduced by carefully scheduling the workload. And the execution order of operators in GNNs can be adjusted without hurting training convergence. We propose a system called **BLAD** to consider the above factors, comprising a *two-level load scheduler* and an *overlap-aware topology manager*. The scheduler allocates each snapshot group to a GPU, alleviating cross-GPU communication. The snapshots in a group are then carefully allocated to processes on a GPU, enabling overlap of compute-intensive NN operators and memory-intensive graph operators. The topology manager adjusts the operators' execution order to maximize the overlap. Experiments show that BLAD achieves 27.2% speed up on training time on average without affecting final accuracy, compared to state-of-the-art solutions.

## CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**.

## KEYWORDS

GNN Training, Machine Leearning, Artifitial Intelegence, GPU

**ACM Reference Format:**
Kaihua Fu, Quan Chen*, Yuzhuo Yang, Jiuchen Shi, Chao Li, and Minyi Guo*. 2023. BLAD: Adaptive Load Balanced Scheduling and Operator Overlap Pipeline For Accelerating The Dynamic GNN Training. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3581784.3607040

*Quan Chen and Minyi Guo are the corresponding authors.


*SC '23, November 12–17, 2023, Denver, CO, USA*

ACM ISBN 979-8-4007-0109-2/23/11…$15.00
https://doi.org/10.1145/3581784.3607040

## 1 INTRODUCTION

Graph Neural Networks (GNNs) are widely used in graph-based intelligent applications (e.g., recommendation systems, traffic forecasting) [44, 51]. While graph structure usually evolves with time in real-life circumstances, researchers have developed many dynamic GNNs (e.g., LRGCN [25], TGCN [8], A3TGCN [4]) for encoding and analyzing the temporal relational data [4, 5, 8, 9, 25, 26, 33, 34, 47, 50]. Popular industrial GNN frameworks [13, 42, 45] also support to train dynamic GNN models on the temporal evolving graph data.

Snapshots-based representation is the method for building dynamic graph datasets that capture the temporal evolution of graph structure [22]. A dynamic graph dataset comprises multiple discrete snapshots, each conducted by a static graph to represent the graph topology structure over a period of time. Figure 1 summarises the typical workflow to train a dynamic GNN model [4, 34]. In one training epoch, the dynamic GNN model learns the graph structure of each snapshot. Moreover, to learn the evolving snapshot structure over a period of time, the input for training a snapshot is a group that combines the current snapshot with adjacent previous snapshots. Multiple GNN-RNN pairs learn the above snapshot groups, while GNNs learn snapshots' structure independently and the Recurrent Neural Network (RNN) [39] learns the temporal features between snapshots, respectively.

Multiple GPUs/nodes are often used to accelerate the training, as training one snapshot involves processing multiple snapshots within the group and each snapshot containing a large number of graph vertices. State-of-the-art work enables two common schemes to distribute the training load. Vertex partition (e.g. Aligraph, DGL, PyGT [37, 42, 45]) distributes the vertices of each graph snapshot to different GPUs, and snapshot partition(e.g. ESDG [7]) distributes the snapshots to different GPUs. As shown in Figure 1, neighbor features are transferred between GPUs with the vertex partition, and hidden states are transferred with the snapshot partition. And our investigations indicate that both schemes are inefficient due to the *high communication overhead*, the *long synchronization*, and the *poor resource usage*.

As for the first inefficiency, frequent communication between GPU devices leads to high overhead. Our measurements show that up to 48.8% and 21.4% of time is spent on communication when we use two GPUs (one GPU on each node) to train a dynamic GNN model with vertex partition and snapshot partition respectively.

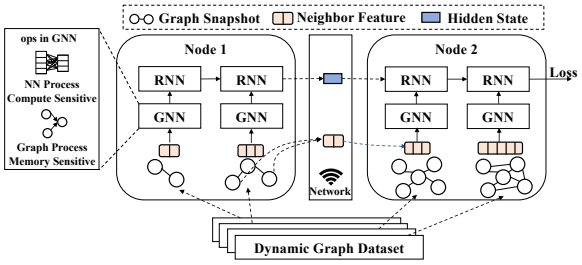



**Figure 1: Training a dynamic GNN model on a GPU cluster.**

As for the second inefficiency, snapshots are often grouped to provide low and scalable communication. The snapshots are partitioned using the static even strategy, where each GPU processes the same number of snapshots [7]. The solution ignores performance disparities that arise due to changes in the workload across different snapshots. As the temporal graph structures evolve in real time, the vertices and edges of each snapshot also vary. For example in Figure 1, "Node 1" trains snapshots with 2 and 3 vertices, and "Node 2" contains snapshots with 5 and 5 vertices, respectively. Thus, the varying loads of different snapshots result in imbalanced training duration on GPUs, leading to significant overhead during parameter synchronization of the distributed GPUs.

As for the third inefficiency, prior work runs the GNN model in the topology sequential order like the DNN framework. However, the sequential execution in dynamic GNN models may lead to low GPU utilization. This is because the dynamic GNN models comprise compute-sensitive NN-based operators and memory-intensive graph operators with irregular random access patterns. While the operators are executed in turn, either the streaming multiprocessors (SMs) or the global memory bandwidth of a GPU are idle during training. For example, when the graph operators are executed, the computation operators are idle and lose the potential to fully utilize GPU SMs' resources.

We propose an efficient training scheme named **BLAD** to resolve the above problems based on two insights. 1) An efficient partitioner is required to balance the load of GPUs while mitigating the communication overhead, as the snapshots in a graph dataset evolve in runtime. 2) Overlapping graph operators and NN operators improves GPU resource efficiency. The training scheme should schedule the execution order of the operators to maximize the overlap without hurting the accuracy.

BLAD comprises a *two-level load scheduler*, and an *overlap-aware topology manager*. The scheduler partitions the graph dataset into snapshot groups. All groups without data dependency are scheduled to one GPU, and each GPU/node is assigned multiple groups. In this way, the cross-GPU communication can be reduced. On each GPU, the scheduler launches two processes to train different graph snapshots, enabling the opportunity to support the overlap execution order of the compute-intensive operators and memory-intensive operators. The scheduler then carefully assigns the snapshots in the group to each process and ensures that the training workloads on the GPU processes are similar to minimize the synchronization. Moreover, the topology manager adjusts the execution order of the model in the processes according to the duration of the operators to maximize the computation and memory access overlap.

This paper makes three main contributions.

- **Comprehensive characterization of dynamic GNN training.** We identify the root cause of the low GPU utilization problems of dynamic GNN training. The analysis motivates us to design the BLAD scheme.
- **The design of the load-aware scheduler.** We propose a two-level dynamic graph dataset partition strategy to ensure the low communication overhead inter-GPU and the training load balance intra-GPU.
- **The design of the adaptive model topological deployment.** The strategy adjusts the execution order of the training model to overlap the computation and memory-sensitive operators in different processes as much as possible without affecting the final training accuracy.

We evaluate BLAD using both one node with 2 Nvidia 2080Ti GPUs and a DGX-2 machine with 8 Nvidia V100 GPUs. BLAD effectively increases the throughput of training dynamic GNN models by up to 79.1% compared with Aligraph [45] and 27.2% compared with ESDG [7] without impacting the final training accuracy.

## 2 RELATED WORK

There has been a large amount of prior work focusing on improving the training efficiency of GNN training using data parallelism or model parallelism techniques [10, 18, 31, 41, 45, 46].

Aligraph, DGL, PyG [13, 42, 45] enabled data parallelism techniques to improve training efficiency in the distributed cluster. They treated the GNN model as a whole part without considering the various characteristics of different operators. As to pipeline techniques optimization, Dorylus [41] recognized the resource sensitivity of different operators and proposed training pipelines. They deploy different operators in virtual machines or lambda instances to get both high performance and scalability. GNNLab and SALIENT [19, 46] enabled the sample-computation pipeline to improve the training speed by mitigating data preprocessing bottlenecks. However, these researches are only applicable to the static GNN model rather than the temporal dynamic GNN model.

As for temporal dynamic GNN models, prior works focused on efficient and scalable training schemes [7, 37, 42, 45]. ESDG proposed an efficient scaling dynamic GNN training scheme [7]. They propose a hybrid technique of data parallelism and model parallelism to ensure the scalability of training dynamic temporal graph networks. Cambricon-G [40] proposed an efficient accelerator to avoid unnecessary graph topology updates for dynamic GNNs. Dynagraph and TGOpt [16, 43] analyze and reduce the redundant graph feature computation in the pipeline. On the contrary, BLAD focus on load balance between training pipeline stages for dynamic GNNs, while all of the above works fail to consider this bottleneck.

Besides, some other works focus on reducing communication overhead between training workers. General GNN training frameworks [6, 18, 30, 42, 45, 48] enabled graph partition algorithms (such as Metis [21]) to separate graph datasets into several subsets, while each worker trains for one subset. However, such vertex-partition methods require heavy graph structure and feature moving over the network leading to high overheads. $P^3$ [15] proposes a hash-partition method to reduce the feature moving during training, while PaGraph and NeuGraph [27, 28] cached the most frequently

**Table 1: Comparison between BLAD and prior works**

| | Dorylus | Aligraph | ESDG | *P*3 | BLAD |
|---|---|---|---|---|---|
| **Dynamic graph** | | ✔ | ✔ | | ✔ |
| **Load balance** | ✔ | ✔ | | | ✔ |
| **Low communication** | | | ✔ | ✔ | ✔ |
| **Low overhead** | | | ✔ | | ✔ |
| **Op overlap** | | | | | ✔ |

**Table 2: Experimental setups**

| | **Specification** |
|---|---|
| **Hardware-1** | Intel Xeon Gold 5120 CPU @ 2.20GHz<br>2 Nvidia GeForce RTX 2080Ti GPU |
| **Hardware-2** | Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz<br>2 DGX-2 machine with 4 V100s-SXM3 GPU each |
| **Software** | Ubuntu 16.04.5 LTS with kernel 4.15.0-43-generic<br>CUDA Driver 410.78 CUDA SDK 10.0 CUDNN 7.4.2<br>PyTorch1.9.0 DGL 0.8.0 |
| **Benchmark** | WD-GCN (WG) [29], TGCN (TG) [8],<br>EvolveGCN (EG) [34] |
| **Dataset** | Arxiv (AR) [17], Products (PR) [17], Reddit (RE) [36]<br>AS-733 (AS) [24], Reddit-body (RB) [23] |

visited vertices feature to overlap data moving and computation. However, the above studies do not consider the dynamic feature of temporal graph datasets. ESDG [7] proposed a snapshot-partition scheme to overcome vertices feature moving for temporal graph neural networks. However, the snapshot partition still requires the communication of temporal features between snapshots leading to extra communication overhead.

Table 1 compares BLAD with prior related work.

## 3 BACKGROUND AND MOTIVATION

In this section, we first introduce the steps to train dynamic GNNs. Then, we show the poor efficiency of state-of-the-art schemes for training dynamic graph neural networks. Moreover, we analyze the training steps to identify the root causes of the inefficiency.

### 3.1 Steps of Training A Dynamic GNN Model

As the dynamic graph structure evolves temporally, the snapshot-based representation has become a mainstream method to represent dynamic graphs [22]. A dynamic graph dataset contains a snapshot sequence $D = (G_1, G_2, \cdots, G_T)$ with $T$ timestamps, and each snapshot $G_t = (V_t, E_t)$ is one static graph topology representative of the dynamic graph structure at the *t-th* time intervals.

Figure 2 shows the typical workflow of training a snapshot with a dynamic GNN model. When the model learns the graph features in one time interval, snapshots at several sequential former time intervals are also required to provide temporal features. Therefore, a dynamic GNN training framework consists of multiple GNN-RNN pairs to learn the above group of snapshots. Each GNN backbone learns the graph features of the snapshot in one interval independently. After each GNN backbone follows an RNN module. The RNN trains the output of GNN and learns temporal features in the time sequential order.

Equation 1 summarizes the step of training a snapshot $S$ in the dataset. Assume the length of the snapshot group is $l$. In the equation, $A_t$ and $X_t$ represent the adjacency matrix of the graph structure and vertex features of snapshot $t$, $W_{GNN}$ and $W_{RNN}$ are the

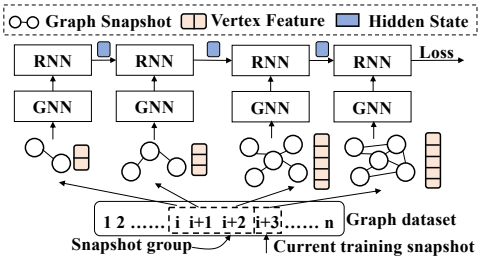



**Figure 2: The workflow of training one snapshot in dynamic GNN model.**

to-be-trained parameters of the GNN backbones and the RNN modules, respectively. $Y_t$ is the graph feature processed by the GNN, which is the input of the RNN. $h_t$ is the hidden state of snapshot $t$ executed by the RNN module to indicate temporal features. $Z_t$ is the output feature of the snapshot $t$.

$$\begin{aligned} &Output = Z_S \\ &Z_t, h_t = RNN(W_{RNN}, Y_t, h_{t-1}) \\ &Y_t = GNN(W_{GNN}, A_t, X_t), t = (S-l, \cdots S) \end{aligned} \tag{1}$$

In general, a training epoch is processed in the following steps. 1) The training process randomly selects the vertices in the dynamic graph based on the pre-defined batch size. 2) The dataloader samples the neighbors of selected vertices in each snapshot to form the input snapshot topology. 3) The training model shuffles the training order of all the snapshots in the dataset. 4) and trains the snapshot in each time interval with the shuffled order. On training snapshot at time intervals $S$, the model extracts the snapshot group $G = (G_{S-l}, \cdots, G_S)$ as the input data and learns the feature according to Equation 1. 5) After training snapshots of all intervals in the batch, the dataloader will start a new input batch until all vertices are selected.

### 3.2 Inefficiency of Current Schemes

We use three dynamic GNNs and five graph datasets to test the performance of current dynamic GNN training schemes. *AS-733* and *Reddit-body* are real-life datasets, while *Arxiv*, *Products* and *Reddit* are three generated datasets based on the static structure to represent larger graph topology. The five datasets have different connectivities between the vertices. We run the experiments on a server with 2 Nvidia RTX 2080Ti GPUs. Table 2 summarizes the detailed hardware and software setup.

We use Aligraph [45] and ESDG [7] to be the representation implementations of the vertex partition-based training and snapshot partition-based training respectively. With Aligraph, a graph cut algorithm is used to split all the snapshots in the graph dataset into multiple subgraphs. Each process on one GPU trains one of the subgraphs of the snapshots at all time intervals. With ESDG, the snapshots are evenly divided into training processes according to the number of snapshots. Each process trains several complete snapshots with part of time intervals.

Figure 3 shows the GPU resource utilization of the benchmarks with Aligraph and ESDG. The $x$-axis shows the combination of the graph and the dynamic GNN model. For instance, "AR-WG" means

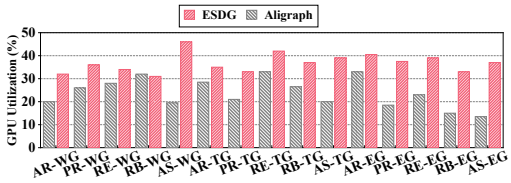



**Figure 3: The GPU utilization of the benchmarks.**

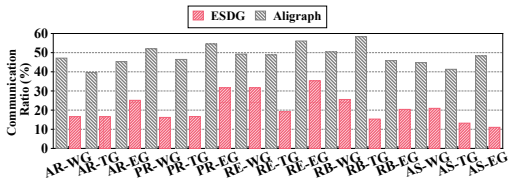



**Figure 4: The percentage of time spent on the data communication with Aligraph and ESDG.**

the graph Arxiv is used to be the input dataset of the model WD-GCN. The GPU utilization is obtained with Nvidia's official tool, *nvidia-smi* [2]. As observed from the figure, the GPU utilization is lower than 45.1% in all the test cases. In general, the low utilization is due to the high communication overhead and the global synchronization overhead. The low utilization results in low training throughput in consequence.

## 3.3 Root Causes of the Inefficiency

We find that the *large communication overhead*, the *long synchronization time*, and the *sequential operator execution* are the root causes of the inefficiency.

*3.3.1 Large communication overhead.* Figure 4 shows the percentage of time spent on the cross-GPU communication with Aligraph and ESDG. As observed, 48.5% and 21.1% of the training time is spent on communication with Aligraph and ESDG.

With Aligraph, a GPU only has part of the vertices of each snapshot, the neighbors of the vertices may not be local. The GPU needs to pull neighbor vertices from remote nodes, and transfer the neighbors' feature from the host memory to GPU global memory. With ESDG, each GPU independently trains the complete snapshot of partial time interval, but has to obtain the hidden state from the previous snapshot to learn the temporal feature. The GPU needs to pull the intermediate hidden state from the previous snapshots from remote nodes at the forward period, and the gradient of the hidden state during the backward period.

Although ESDG already reduces the communication overhead, the overhead is still not negligible.

*3.3.2 Long Synchronization Time.* While ESDG has lower communication overhead, it introduces extra global synchronization overhead between training nodes. Figure 5 shows the percentage of synchronization time across the two GPUs. The synchronization time ratio is defined as the difference between the model computation duration on the two GPUs normalized to the slower one. As shown in the figure, the synchronization at the end of the training takes 31.4% of the training time on average (up to 46.9%).

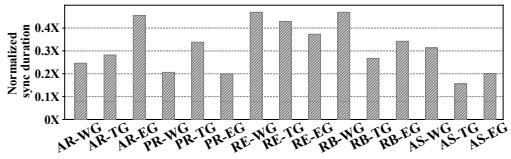



**Figure 5: The percentage of synchronization time between GPUs under ESDG.**

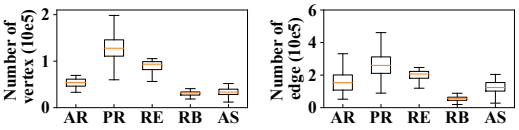



**Figure 6: The number of vertex and edge of the dynamic GNN input graph snapshots.**

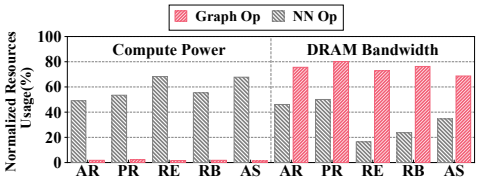



**Figure 7: The diversity sensibility of NN-based and graph-based operators.**

The synchronization is long because the graph structure evolves temporally and snapshots at different time intervals have different loads. While the snapshots have different numbers of vertices and edges, the duration needed to train them varies. Figure 6 shows the distributions of the number of vertices and the number of edges in the graph snapshots. As observed, the number of vertices and edges varies between snapshots on the five datasets. A snapshot may have 6.95X more vertices than another snapshot. By evenly allocating the snapshots, they have different progresses in the training and suffer from long synchronization overhead.

*3.3.3 Sequential Operator Execution.* A dynamic GNN model normally has graph aggregation operators and feature update operators. The aggregation operator accesses the feature vector from the neighbor of learned vertices and aggregates these vectors to the learned vertices (referred to be *graph op*). The update operator performs neural network computations such as fully connected to transform dimensions of the vertices' feature (referred to be *NN op*). When executing the GNN backbone, the GPU runs the two operators sequentially as they have logic dependencies.

We take the popular GNN backbone, GCN [12], as an example, and experiments with other GNN backbones show similar results. Figure 7 shows the compute power usage and the global DRAM bandwidth usage of the graph ops and NN ops. We collect the metrics using Nvidia Nsight compute system [1]. The "Compute Power" usage is represented by the single precision flops normalized to the peak single precision flops of the GPU. As observed, the graph op consumes 1.9% and 58.7% of computation ability and global memory bandwidth resources respectively, and the NN op consumes 74.7% and 34.8% of the two types of resources respectively.

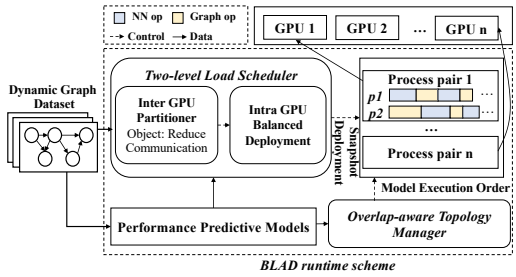



**Figure 8: The design overview of BLAD.**

The two types of operators have stable characteristics with different graph datasets. As the operators are executed sequentially, either the computation ability or the global memory bandwidth of the GPU is under-utilized during the entire training.

## 4 BLAD DESIGN METHODOLOGY

We therefore propose a dynamic GNN training system **BLAD** that improves the overall training throughput according to the above analysis. In this section, we present the overview of our design.

Figure 8 shows the design overview of BLAD. It comprises a *two-level load scheduler*, and an *overlap-aware topology manager*. With BLAD, multiple processes are launched on each GPU to process the snapshot groups, in order to enable the concurrent overlapped operator execution. In this case, the load scheduler analyzes the load of every to-be-processed snapshot group, and allocates the snapshot groups to the GPUs. For each GPU, it further allocates the snapshots in each group to its processes. The topology manager adjusts the execution order of the graph op and the NN op in the GNN, in order to maximize the overlap of graph ops and NN ops from different processes. We have proved that the training results are not affected when the order of the operators is adjusted (Section 6.3).

Because there is no data dependency between snapshot groups and the load scheduler allocates each snapshot to one single GPU, the heavy cross-GPU communication can be eliminated. Meanwhile, the communication and synchronization overhead in the snapshot can be done through efficient in-GPU global memory transmission. The challenging part is that the computational load of a snapshot in the snapshot groups is affected by the number of vertices and edges. The snapshots in a group should also be carefully allocated to the processes, so that they may complete at the same time, minimizing the synchronization overhead.

When processing each graph snapshot, the topology manager adjusts the execution order of the NN ops and the graph ops. The challenging part here is that the duration of each operator is also affected by the number of vertices and edges in the snapshot. In this case, it is hard to match the NN ops and graph ops between the processes on the same GPU. For instance, if the NN op in one process is long but the graph op in the other process is short, the overlap is still negligible even if they are scheduled to run simultaneously. To this end, the topology manager adjusts the operator execution orders to form compute and data access stages of similar duration between the processes on the same GPU (Section 6).

BLAD works in the following steps to train several epochs of a dynamic GNN model $G$. One epoch represents the BLAD training of all of the graph vertices in all snapshots in the dataset. 1) BLAD trains the first epoch to profile the graph load of each snapshot. 2) From the second epoch, BLAD randomly selects a batch of graph vertices, and obtains the input graph of the corresponding graph vertices in all snapshots with neighbors. 3) BLAD next divides the snapshot groups of the training vertices batch into GPUs. And on each GPU, BLAD further split the snapshot group into processes according to the load of each input graph in the snapshot group. 4) The model topology deployment strategy determines the GNN execution order for each input graph to form NN-aggregation overlap. 5) BLAD executes the current batch and starts the next batch return to step 2) until all vertices in the datasets are trained.

We implement BLAD on top of Pytorch [3] and DGL [42] library. Pytorch and DGL execute the NN-based and graph operators on the GPU respectively. To enable multiple process training on GPUs, we utilize the *torch.distributed* package. Communication between processes is facilitated through CUDA IPC [35]. The BLAD methodology is generally applicable and is not limited to any particular GNN training framework.

## 5 TWO-LEVEL LOAD SCHEDULING

In this section, we propose the two-level load scheduler. The top-level scheduler proposes the snapshot group partition to reduce cross-GPU communication (Section 5.1). The bottom-level scheduler introduces a load balance scheme inside each GPU (Section 5.2).

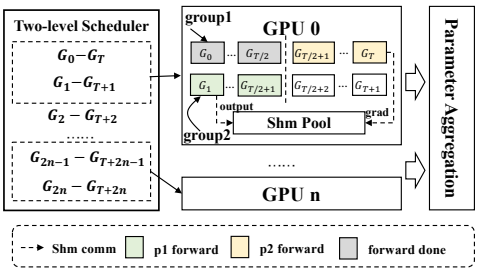
### 5.1 Inter GPU Snapshot-group Partition

Both vertex-based partition and snapshot-based partition suffer from significant communication overhead, because the subgraphs have data dependencies on neighbor features for vertex-based partition, and snapshots on multi GPUs have dependencies on intermediate RNN hidden states for snapshot-based partition.

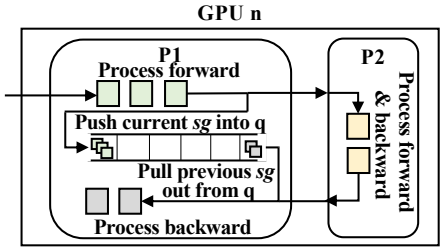
Figure 9(a) shows the strategy of our inter-GPU partition. By design, when learning a snapshot in dynamic GNN training, several adjacent snapshots generated in previous time intervals are required to provide temporal features [7]. Leveraging this design choice, a snapshot group includes the current snapshot and the adjacent historical snapshots. Due to the datasets containing multiple snapshot groups and each snapshot group can be trained independently, BLAD partitions the dataset in the granularity of the snapshot group into GPUs, while one GPU trains a subset of the snapshot group. For example, as shown in the figure, GPU 0 processes the snapshot groups corresponding to $G_T$ and $G_{T+1}$, while GPU $n$ handles the groups corresponding to $G_{T+2n-1}$ and $G_{T+2n}$.

Because there is no data dependency between snapshot groups, each GPU can train the assigned snapshot groups independently. Meanwhile, each snapshot group includes all the required snapshots for the current trained snapshot, i.e. the last snapshot in the group, and all the graph vertices of snapshots are stored in the local GPU, there is no need to transfer intermediate hidden state/gradients between GPUs or communicate neighbor features. The only inter-GPU communication occurs during the all-to-all gradient aggregation to update the model's parameters after the GPU completes training one snapshot group.

As for the intra-GPU level, BLAD further partition each snapshot group into two processes (to be analyzed in Section 5.2). Due to the data dependency between snapshots in one group, the processes




(a) The partition strategy at iner GPU and intra GPU level.




(b) The design of the snapshot group queue to handle RNN dependency.

**Figure 9: The design of the two-level load scheduler.**

in one GPU have to transfer intermediate hidden state/gradients. However, we utilize the global shared memory as the communication backend through the processes in the same GPU and the intra-GPU communication can be negligible.

## 5.2 Intra GPU Load Balanced Partition

On each GPU, BLAD launches two processes and forms a training pipeline to improve the GPU resource efficiency. Figure 9(a) also depicts the design of the partition strategy at the intra-GPU level. BLAD partitions the snapshots of each snapshot group into two processes. The two processes concurrently train part of each snapshot group to support the overlapped operator execution order.

However, several problems have to be resolved to build an efficient overlapped pipeline. First, the RNN modules in the dynamic GNN models have data dependencies between snapshots, which makes it challenging to train multiple snapshots simultaneously. If one process completes the forward stage of one group, the backward stage cannot be executed until the remaining part of the group has been processed by the other process. Second, since the load of snapshots varies in the datasets, leading to the processing time of each snapshot also varies. Therefore, a static partition may not maintain low synchronization all the time. In order to address the above issues, we define a training scheme for processes to support parallel training under the RNN dependencies and then propose a partitioning strategy to keep the pipeline balance.

*5.2.1 Handling the RNN dependencies.* Figure 9(a) shows the execution flow of processes in one GPU. BLAD partitions snapshots in each snapshot group into two parts to the two processes, denoted as $sg_i^a$ and $sg_i^b$ where $sg_i$ represents the $i$-th snapshot group. Once a process completes the forward stage in $sg_i$, it transfers the intermediate output to the next process and begins training $sg_{i+1}$ immediately. As an example, *p1* in the GPU processes the forward stage of $sg_1^a$, which contains snapshots from $G_0$ to $G_{T/2}$. When *p1* finishes the forward stage of $sg_1^a$, it immediately starts processing $sg_2^a$ while transferring the output to *p2* which simultaneously processes the forward stage of $sg_1^b$ containing the remaining part of $sg_1$, i.e. $G_{T/2+1}$ to $G_T$.

As the process skips the backward stage of the part with data dependencies, it has to be processed in later the training pipeline period. BLAD proposes a snapshot group queue for each process to store the *sg* with dependencies as shown in Figure 9(b), similar to DNN parallel training technology [31]. When a process finishes the forward stage of a snapshot group *sg* with dependency, the group is pushed into queue *q*. Once the process receives the dependent gradient from the next process, the *sg* can be pulled out of the queue and the backward stage can be executed.

*5.2.2 Load balanced partition.* In order to maintain a balanced training pipeline and mitigate synchronization overhead in the pipeline, BLAD needs to carefully partition snapshots in each group into two processes. However, balancing the workload between processes is challenging since adjusting the number of snapshots for one process will affect the load of all processes in the pipeline. For example, if *p1* trains one less snapshot in the first group, *p2* will train an extra remaining snapshot in the next period, leading to a higher load for *p2*. Conversely, *p1*'s workload will decrease in the next period when it processes the backward stage of the first group with one less snapshot. As a result, the pipeline may become less efficient in the next period.

We therefore propose a dynamic optimization-based load balance strategy as shown in Equation 2 to resolve the above challenge, where $OPT(i, p_i)$ indicates the total time consumption for training from the first to the $i$-th snapshot group when *p1* executes $p_i$ snapshots in the $i$-th snapshot group. The $group$ is the size of the snapshots in one group. $l(i, j)$ and $b(i, j)$ represent the forward and backward duration of the $j$-th snapshot in the $i$-th group respectively, which is obtained from the GNN performance model. Suppose there are $m$ snapshot groups deployed to the GPU, then the balancer determines the number of snapshots trained in processes for each snapshot group by calculating $min_{p_m=1}^{group} OPT(m, p_m)$.

$$
\begin{aligned}
OPT(i,p_i) &= \begin{cases} \sum_{j=1}^{p_i} l(i,j), if(i=0) \\ min_{p_{i-1}=1}^{group}(OPT(i-1,p_{i-1}) + max(T_{p1},T_{p2})), else \end{cases} \\
T_{p1} &= \sum_{j=1}^{p_i} l(i,j) + \sum_{j=1}^{p_{i-1}} b(i-1,j) \\
T_{p2} &= \sum_{j=p_{i-1}}^{group} (l(i-1,j) + b(i-1,j))
\end{aligned}
\tag{2}
$$

In order to obtain $l(i, j)$ and $b(i, j)$ of snapshots in Equation 2, it's necessary to know the duration of each operator with various determined load input graphs. Similar to previous researches [11, 14, 20, 49], We offline profile the execution time of operators in the model at typical loads and build a lightweight linear regression (LR) predictor for each operator. Generally, after the GPU obtains the input graph of each snapshot, the predictor takes the number of vertices, the number of edges, and the number of the feature dimensions of the graph as inputs and estimates the duration of each operator in the graph model.

**Table 3: Communication Volume of vertex partition, snapshot partition and snapshot-group partition (group for short).**

| Method | GNN param | RNN param | Features |
|---|---|---|---|
| **Vertex** | $p * W_{GNN}$ | $p * W_{RNN}$ | $T * prob * N_{khop} * F$ |
| **Snapshot** | | $p * W_{RNN}$ | $2 * N_{vertex} * T * F$ |
| **Group** | $p * W_{GNN}$ | $p * W_{RNN}$ | |

## 5.3 Reduce Communication with BLAD

In this section, we analyze the communication overhead of the different partitions. Suppose we train the dynamic GNN in a cluster with $p$ nodes, and each snapshot group contains $T$ snapshots, while each snapshot has $N_{vertex}$ vertices and $N_{khop}$ k-hop neighbor vertices. And the dimension of the vertex feature vector is $F$. Table 3 compares the communication overhead breakdown of the snapshot-group partition with the vertex partition and snapshot partition. The first two "param" column records the communication column of parameter synchronization while the "Features" column records the feature communication.

For vertex partition, because part of the neighbor vertices of each snapshot are deployed on the remote nodes, the neighbor feature communication happens when the GNN layers are executed. The "*Features*" column in Table 3 summarizes the overhead of neighbor communication, where $T * N_{khop} * F$ represents the neighbor vertices feature vector of all snapshots and $prob$ indicates the probability that the one neighbor is not on the local node. For snapshot partition [7], GPUs need to transfer the intermediate data/gradient of previous time intervals when RNN layers are executed.

On the contrary, with BLAD, each GPU processes independent snapshot groups and a snapshot is totally executed on one GPU, thus eliminating the neighbor communication and intermediate data/gradient transmission. While BLAD has no feature communication, the communication of BLAD is smaller than the vertex partition. However, compared with the snapshot partition, BLAD introduces extra GNN parameters synchronization overhead shown in the "GNN param" column, because GNNs are trained on GPUs independently. It is worth noting that the size of the GNN parameter matrix $W_{GNN}$ is smaller than the intermediate feature data $N_{vertex} * F$. At the same time, the length of snapshot group $T$ is larger than $p$ in order to ensure that each GPU handles at least one snapshot in ESDG [7]. Therefore, the communication overhead of the snapshot group partition is less than the snapshot partition.

# 6 OVERLAP-AWARE TOPOLOGY MANAGER

In this section, we first investigate the potential of overlapping computation-sensitive operators and memory-sensitive operators for improving training efficiency, and then we propose our adaptive topology manager to determine the execution order dynamic graph model between co-locate process pairs. Finally, we prove that the modified model by our strategy is equivalent to the original model.

## 6.1 Possible Overlap Opportunities

As analyzed, when training dynamic graph datasets, graph aggregation operators hardly use GPU computation resources, and NN-based operators also hardly fully utilize memory bandwidth. Therefore, the sequential execution of model operators leads to low utilization. However, by co-locating multiple dynamic GNN model instances on one GPU, the GPU has the potential to execute both computation and memory-sensitive operators simultaneously, improving the utilization of these two types of GPU resources.

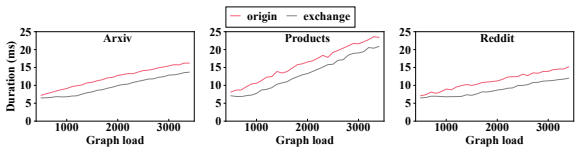



**Figure 10: The opportunity of comupte-sensitive operators and memory-sensitive operators overlap.**

To investigate the potential of improving training through operator overlap, we co-locate two processes on one GPU training the GCN model with "origin" and "exchange" strategies respectively. With the origin strategy, both two models have the original operator execution order, i.e. the NN-based operator is executed after the graph operator. With the exchange strategy, one process retains the original model, while the other process has the opposite order, i.e. the NN operator is executed before executing the graph operator.

Figure 10 shows the duration of one iteration for snapshots with the two strategies. The $x$-axis represents the number of vertices of the input graph. As shown, for the three graph datasets, the exchange strategy reduces the duration in all graph load, by an average of 25.9%, 23.3%, and 27.6%, respectively. Through reordering the execution of one model, the computation and memory-sensitive operators in different models can be executed simultaneously, thus mitigating the resource contention caused by executing the same type of operators.

## 6.2 Model Topology Adjustment Problem

When managing the model topology, two challenges have to be resolved in BLAD. Firstly, data dependencies exist between the operators. BLAD must carefully select the exchangeable operators to ensure to maintain the training accuracy. Secondly, the execution time of each operator is different, because the input graph of process training has the various number of vertices and edges. As a result, it is hard to match the NN ops and graph ops between the processes on the same GPU with a static execution order.

Figure 11 shows the design principle of the topology manager. A dynamic GNN model alternately contains multiple GNN and RNN layers. Each GNN layer consists of an operator pair comprising an NN-based operator and a graph aggregation operator. The first and second GNN layers handle the 2-hop and 1-hop neighbors of the vertices respectively. Between the two GNN layers contain one RNN layer to process temporal features. The RNN layer includes several fully connected operators to process hidden states, thus we treat the RNN layer as the NN-based operator.

Our strategy changes the execution order of operators according to the execution time of each operator in co-located processes, to maximize the overlap of NN-based operators and graph operators. For example, in Figure 11, if we exchange the execution order of the first NN-based operator and the graph operator pair in $p_2$, i.e. the graph operator is executed after the NN-based operator, the GPU has more computation and memory overlap with process $p_1$ during GPU training.

In detail, we adjust the execution order of operator pairs at the granularity of intra-GNN layers, as exchanging operators within

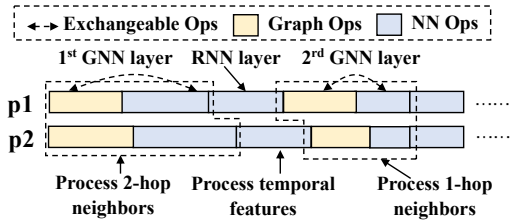



**Figure 11: NN-based operator and graph operator overlap of colocate processes in one GNN layer.**

a GNN operator pair is mathematically equivalent (to be proven in Section 6.3). We do not alter the execution order of operators in different GNN layers, as they involve data dependencies. For instance, in a two-layer GNN network, the second GNN layer cannot begin processing the 1-hop graph vertices until all of the 2-hop graph vertices have been processed by the operators in the first GNN layer.

We define a deployment reward $DR$ to measure the overlap of computation and memory-sensitive operators of co-located processes as shown in Equation 3. $DR$ is defined as the $XOR$ operation of two model topological functions, where $f$ is a binary function whose values are set to 1 and 0 respectively if the current time model is executing computation (e.g. feature update and RNN) or memory sensitive (e.g. graph aggregation) operators.

$$DR(t_1, t_2) = \Sigma_{t=t_1}^{t_2} (f_{p_1}(t) \oplus f_{p_2}(t)) \tag{3}$$

Algorithm 1 summarizes the topology manager of BLAD that determines the execution order of operators for co-locate processes according to the following steps. First, the scheduler pushes the initial operator of each training model into the operator execution queue of the corresponding process and sets the initial duration of the two processes to 0. The scheduler continuously selects processes with shorter duration and pushes operators to the execution queue. (line 4-6). If there is a commutative operator, the scheduler attempts to exchange the operator pairs and select execution order with larger $\Delta DR$ incremental value according to Equation 3 (line 7-11). The scheduler inserts the op into the process in the order corresponding to the larger $\Delta DR$ value and updates the improved duration of the process (line 12-14).

**Algorithm 1** The algorithm of topology adjustment manager

**Require:** The operator list of two processes $O_k = op^1_{p_k}, op^2_{p_k} \cdots op^{n_k}_{p_k}$
**Require:** Model topology function $f_{p_k}$
1: $EXE_k \leftarrow \phi$
2: $T_{p_1}, T_{p_2} \leftarrow 0$
3: **while** exists operator not pushed into $EXE^k$ **do**
4: $loc = min(T_{p_1}, T_{p_2})$
5: $op$=$O_{loc}$.pop()
6: $\Delta DR \leftarrow DR(op.start, op.end)$ with $f_{loc}$
7: **if** $op$.commutative() **then**
8: $f'_{loc} \leftarrow 1 - f_{loc}$
9: Calculate $\Delta DR'$ with $f'_{loc}$
10: **if** $\Delta DR' > \Delta DR$ **then**
11: set $\Delta DR \leftarrow \Delta DR'$ and exchange the operator pair
12: $EXE_{loc}$.push(op)
13: $T_{p_{loc}} \leftarrow T_{p_{loc}}$ + $op.end$ - $op.start$
14: remove $op$ from $O_{loc}$
15: **Return:** $EXE_k$

**Table 4: Aggregation of different graph convolution.**

| Layer | Aggregation Formulas |
|---|---|
| **ChebConv** | $Agg(X) = \theta_k T_k(L)X$ |
| **GCNConv** | $Agg(X) = L_{sym}X$ |
| **RGCNConv** | $Agg(X) = \hat{A}_\phi X$ |

## 6.3 Mathematical Equivalence Proof

We prove that BLAD does not affect the training efficiency after exchanging the operator execution order. As analyzed, the graph convolution layer of the dynamic GNNs comprises feature update and graph aggregation operators, which are used to learn the feature from the vertices themselves and their neighbors' features, respectively. The update and aggregation operators are usually processed by the weight matrix and the graph filter, respectively.

In general, Equation 4 summarizes the processing operators of one graph layer, where $f(W, X)$ and $Agg(H, X)$ represent update and aggregation operators respectively, and $W$ and $H$ represent NN-based matrices and graph filters respectively. In particular, if NN and graph filters can be summarized into linear matrices, Equation 4 can be expressed as matrix multiplication $HXW$. Since the matrix multiplication satisfies the associative law, the exchange of the execution order of the two operators will not affect the final results.

$$f(W, Agg(H, X)) = HXW \tag{4}$$

We investigate the popular graph convolution backbones used by dynamic graphs in PyGT model zoo [37], including Chebyshev convolution, GCN convolution, and RGCN convolution [12, 38] as shown in Table 4. Although the above convolution layer aggregation operators have different implementations, they can all be summarized as linear matrix multiplication. For Chebyshev convolution, $H$ can be expressed as $k$-order Chebyshev polynomial with coefficient $\theta_k T_k(L)$. For GCN convolution, the aggregation can be written as symmetric Laplace matrix $L_{sym} = D^{-1/2}\hat{A}D^{-1/2}$, where $\hat{A} = A + I$ and $D_{ii} = \sum_j A_{ij}$. And for RGCN convolution, the $H$ is calculated by the adjacency matrix $\tilde{A}_\phi = (D_\phi)^{-1}\hat{A}_\phi$ ,where $\hat{A}_\phi = A_\phi + I$ and $(D_\phi)_{ii} = \sum_j (\hat{A}_\phi)_{ij}$ and $\phi$ denoted the edge relation type. Furthermore, as for feature update, the above convolution layers all enable linear operators, which can be directly expressed as the feature multiplied by the parameter matrix $W$.

# 7 EVALUATION OF BLAD

In this section, we first evaluate the performance of BLAD in improving the training throughput, and show its impact on the training accuracy and the effectiveness of each proposed module. Then, we evaluate BLAD on a distributed platform. Lastly, we discuss the runtime overhead.

## 7.1 Experimental Setup

We evaluate BLAD using two hardware platforms. Table 2 already summarizes the software and hardware experimental configurations. The first platform is a machine equipped with two Nvidia RTX 2080Ti GPUs, another platform has two DGX-2 machines with eight V100 GPUs [32]. BLAD does not rely on any special hardware features of the two GPUs. It is applicable to other generations of GPUs. Except for Section 7.6, the experiments are done on the machine with two 2080Ti GPUs.

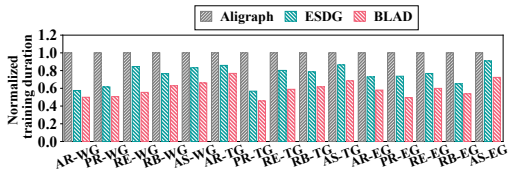



**Figure 12: The normalized duration of training one epoch on benchmarks with Aligraph, ESDG and BLAD.**

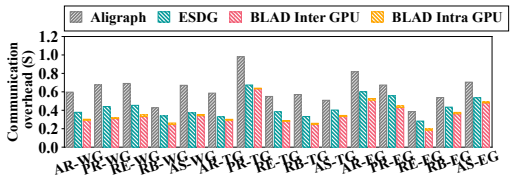



**Figure 13: The communication overhead in one training batch with Aligraph, ESDG and BLAD.**

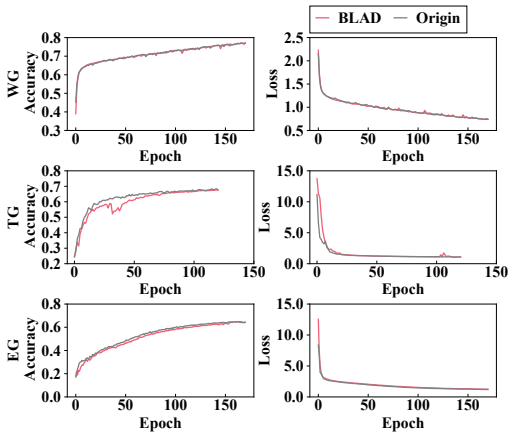



**Figure 14: The accuracy and loss at training with BLAD and the original model under Arxiv graph dataset.**

We use three representative dynamic GNNs (*WD-GCN (WG)*, *TGCN (TG)*, *EvolveGCN (EG)*) that combine different RNN and GNN backbones. The three dynamic GNNs use GCN [12] as the backbone. Each GCN model has two layers with one feature update operator and one graph aggregation operator. The feature update operator uses the fully-connected layer. In EvolveGCN, the RNN processes the GNN parameters of snapshots, while TGCN and WDGCN deal with intermediate vertex features of snapshots.

Five dynamic graph datasets are used to evaluate BLAD's performance. *AS-733* and *Reddit-body* (*AS* and *RB* for short) [23, 24], are real-life datasets that capture traffic flow routing and networks of hyperlink topology. Additionally, we build three generated datasets, *Arxiv* (*AR*), *Products* (*PR*), and *Reddit* (*RE*) based on static graphs to evaluate BLAD on larger datasets. We create snapshots of these datasets by randomly deleting some of the edges in the static graph. The evolution trend of the number of edges between snapshots in each generated dataset is the same as that of AS.

We compare BLAD with Aligraph [45] and ESDG [7] that use vertex partition and snapshot partition to train dynamic GNNs, respectively. With Aligraph, Metis algorithm [21] is used to offline partition vertices in the graph dataset to training nodes. At runtime, each node trains partial local vertices of the snapshots with all snapshots. With ESDG, snapshots in a snapshot group are evenly distributed to the GPUs according to the time interval. At runtime, each node trains a series of complete snapshots of partial intervals.

## 7.2 Training Throughput

In this experiment, we first evaluate BLAD in reducing the training time. Figure 12 reports the normalized time of training an epoch with Aligraph, ESDG, and BLAD, in the $3 \times 5 = 15$ test cases. The $x$-axis shows the test cases. "AR-WG" represent the case that training a *WD-GCN* model using the dataset *Arxiv*. As observed, BLAD achieves speeds up at 1.27X and 1.79X on the time of training an epoch compared with ESDG and Aligraph, respectively.

BLAD achieves the best performance because it minimizes the cross-GPU communication, alleviates the synchronization overhead, and overlaps the compute-intensive operators and memory access-intensive operators. On the contrary, Aligraph suffers from heavy communication overhead between GPUs. With the vertex partition, a vertex needs to obtain data from its neighbor vertices on other GPUs. Similarly, by adopting the snapshot partition in ESDG, the snapshots in the same snapshot group are processed on different GPUs. GPUs still need to synchronize. By mapping snapshots evenly to training processes, the load is not balanced.

In terms of reducing the communication overhead, Figure 13 shows the communication time in a training batch with Aligraph, ESDG and BLAD. As observed, BLAD reduces the communication time by 46.8% and 17.4% on average, compared with Aligraph and ESDG. BLAD reduces the communication overhead by assigning an entire snapshot group to a GPU. In this case, BLAD suffers neither neighbor aggregation nor intermediate feature/gradient communication. The communication only happens between the processes on a GPU. BLAD uses GPU global shared memory to support efficient inter-process communication. Intra-GPU communication only accounts for 5.8% on average.

## 7.3 Training Accuracy

It is crucial to ensure training accuracy when we improve the training throughput. In this experiment, we show the training accuracy and loss of the dynamic GNN during the training process with BLAD. The training accuracy and loss are compared with the counterparts of the original training method. The original training method uses a single GPU to process each snapshot one by one sequentially.

As the example, Figure 14 shows the accuracy and loss of the three dynamic GNN models on the dataset Arxiv. We train 175 epochs for WG and EG, and 125 epochs for TG to ensure they are at the convergence precision. Experiments with other datasets show similar results.

As observed from this figure, BLAD achieves nearly the same training accuracy and loss compared with the original training method. In the worst case, the difference between the accuracy of BLAD and "origin" training scheme is 1.2%. The optimizations in BLAD do not degrade the training accuracy noticeably.

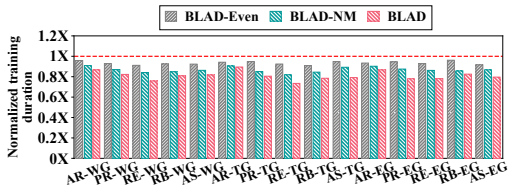



**Figure 15: The normalized duration on training one epoch with BLAD-Even, BLAD-NM and BLAD.**

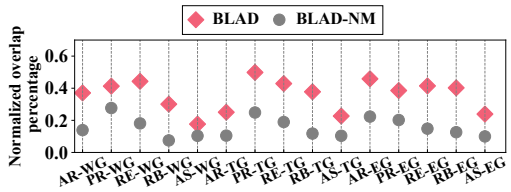



**Figure 16: The NN-based and graph aggregation operators overlap percentage with BLAD and BLAD-NM.**

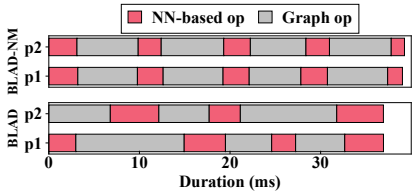



**Figure 17: The operator execution timeline when training the EG model on Arxiv dataset with BLAD and BLAD-NM.**

## 7.4 Effectiveness of Topology Manager

In this experiment, we evaluate the effectiveness of the topology adjustment strategy for maximizing the compute-intensive and memory access-intensive operator overlap. We compare BLAD with BLAD-NM, a variant of BLAD that disables the topology manager. With BLAD-NM, the two processes on the same GPU run the operators in the default order.

Figure 15 shows the time of training an epoch with BLAD and BLAD-NM. The time is normalized to the epoch time with ESDG. As observed, the time of training an epoch increased by 7.3% on average with BLAD-NM, compared with BLAD. The overlap helps in reducing the training time. Figure 16 shows the overlap percentage of the NN-based and the graph aggregation operators normalized to the time of training an epoch. As observed, in 15.5% and 35.9% of the time, the NN operators and graph operators overlap with BLAD-NM and BLAD, respectively.

Figure 17 shows an operator execution timeline example in one snapshot group with BLAD and BLAD-NM. The data is collected when training the EG model on the Arxiv dataset. And other cases show similar results. As observed, by fine-tuning the execution order of the training operators co-located in the GPU, BLAD greatly improves the overlap of NN-based operators and the graph aggregation operators. The improved overlap in turn enables the GPU to better utilize the GPU SM and memory bandwidth simultaneously.

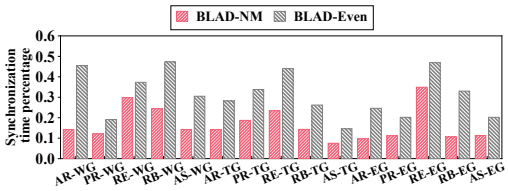



**Figure 18: The percentage of synchronization time with BLAD-NM and BLAD-Even.**

## 7.5 Effectiveness of the Load Balance Strategy

In order to minimize the synchronization overhead, BLAD uses a load balance strategy to distribute the snapshots to the processes on a GPU. In this experiment, we evaluate the effectiveness of this load balanced strategy. We compare BLAD-NM with BLAD-Even. Different from BLAD-NM, BLAD-Even disables both the topology manager and the load balanced strategy, and evenly allocates snapshots to the training processes on each GPU according to the time intervals.

Figure 15 also shows the performance of BLAD-Even. As observed, the time of training an epoch further increases by 7.7% on average (up to 12.8%) with BLAD-Even, compared with BLAD-NM. The load balancing strategy helps in reducing the synchronization overhead.

In more detail, Figure 18 shows the average percentage of synchronization time between processes with BLAD-NM and BLAD-Even. As observed, the synchronization takes 16.7% and 31.4% respectively with BLAD-NM and BLAD-Even. The performance loss happens because BLAD-Even enables a suboptimal snapshot partition. On the contrary, an adaptive load balanced pipeline deploys snapshots based on the load of each snapshot and eliminates unnecessary synchronization.

## 7.6 Adapting to Distributed Clusters

We use two DGX-2 machines, each with four Nvidia V100 GPUs, to evaluate the performance of BLAD on distributed clusters. Figure 19 reports the normalized time of training an epoch on the two nodes with Aligraph, ESDG, and BLAD. All the time is normalized to the time of training an epoch with Aligraph. As observed, BLAD reduces the time of training an epoch by 41.9% and 21.3% compared with Aligraph and ESDG.

Benefiting from the snapshot group partition, BLAD still performs best on more nodes and more GPUs. Figure 20 summarizes the communication overhead in one training batch with BLAD, ESDG, and Aligraph. Because each worker only needs to aggregate the parameters of the training model, without neighbor aggregation and transmission of intermediate features/gradients. This makes the communication overhead in BLAD performs better in the scalable cluster.

## 7.7 Overhead of BLAD

In BLAD, the *scheduling training dataset*, *predicting the duration of the operators* and the *intra-GPU shared memory communication implement* introduce extra overhead.

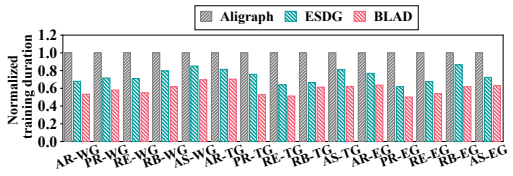



**Figure 19: The normalized duration of training one epoch with Aligraph, ESDG and BLAD on DGX-2 machines.**

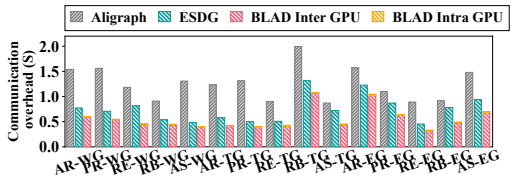



**Figure 20: The communication overhead in one training batch on the DGX-2 machines with Aligraph, ESDG and BLAD.**

**Scheduling overhead:** As stated in Section 5.2, BLAD needs to find the balanced partition for dynamic graph datasets and determine the execution order of the training model. The time consumption of the above two operations for scheduling all snapshots in one batch is less than 8ms and 2.3ms, respectively. It's far less than hundreds of milliseconds for one batch of training. **Predicting overhead:** BLAD maintains a predictor to estimate the duration of model operators to support dataset partition. Predicting all snapshots in one batch completes in less than 1 ms, which is much shorter than the training duration. **Memory overhead:** The memory overhead comes from an extra shared memory pool for intermediate data and gradient communication between training processes. For our benchmarks, the overhead used in one GPU has at most 8.70MB global memory in total. To conclude, the overhead of BLAD is acceptable for real-system deployment.

## 8 CONCLUSION AND FUTURE WORK

We propose BLAD, an effective dynamic GNN training system, to improve the utilization of GPU resources in multi-GPU clusters. BLAD comprises a two-level load scheduler and an overlap-aware topology manager. The two-level load scheduler deploys the graph dataset on multiple GPUs with low communication overhead. On each node, the GPU hosts multiple processes pipeline to improve the utilization of GPU resources. The load scheduler maintains an intra-GPU load balancing strategy to minimize the synchronization overhead between training processes. The topology manager adjusts the operator execution order of the training model to improve the compute-memory operators' overlap without hurting the training convergence. Experiments show that BLAD can improve overall training efficiency by 27.2% on average, compared with state-of-the-art work. Partitioning the dynamic graph datasets with long-term temporal dependencies would be a limitation of the proposed methods. As a single GPU may not hold the entire training group with a large number of related snapshots, the global memory may become a new bottleneck. A hybrid scheme that combines vertex partition, snapshot partition, and group partition would be interesting future work.

## ACKNOWLEDGEMENTS

We thank all the reviewers for their constructive comments on our work. This work is partially sponsored by the National Natural Science Foundation of China (62022057, 62232011, 61832006), and Shanghai international science and technology collaboration project (21510713600), and Open Project of Hubei Province Key Laboratory of Intelligent Information Processing and Real-time Industrial System (ZNXX2022002).

## REFERENCES

[1] [n. d.]. Nvidia Nsight System. https://developer.nvidia.com/nsight-systems.
[2] [n. d.]. NVIDIA System Management Interface. https://developer.nvidia.com/.
[3] [n. d.]. Pytorch. https://developer.nvidia.com/.
[4] Jiandong Bai, Jiawei Zhu, Yujiao Song, Ling Zhao, Zhixiang Hou, Ronghua Du, and Haifeng Li. 2021. A3t-gcn: Attention temporal graph convolutional network for traffic forecasting. *ISPRS International Journal of Geo-Information* 10, 7 (2021), 485.
[5] Lei Bai, Lina Yao, Can Li, Xianzhi Wang, and Can Wang. 2020. Adaptive graph convolutional recurrent network for traffic forecasting. *Advances in neural information processing systems* 33 (2020), 17804–17815.
[6] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: An Efficient Communication Library for Distributed GNN Training. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 130–144. https://doi.org/10.1145/3447786.3456233
[7] Venkatesan T. Chakaravarthy, Shivmaran S. Pandian, Saurabh Raje, Yogish Sabharwal, Toyotaro Suzumura, and Shashanka Ubaru. 2021. Efficient Scaling of Dynamic Graph Neural Networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 77, 15 pages. https://doi.org/10.1145/3458817.3480858
[8] Bo Chen, Wei Guo, Ruiming Tang, Xin Xin, Yue Ding, Xiuqiang He, and Dong Wang. 2020. TGCN: Tag graph convolutional network for tag-aware recommendation. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 155–164.
[9] Jinyin Chen, Xueke Wang, and Xuanheng Xu. 2022. GC-LSTM: Graph Convolution Embedded LSTM for Dynamic Network Link Prediction. *Applied Intelligence* 52, 7 (may 2022), 7513–7528. https://doi.org/10.1007/s10489-021-02518-9
[10] Weihao Cui, Zhenhua Han, Lingji Ouyang, Yichuan Wang, Ningxin Zheng, Lingxiao Ma, Yuqing Yang, Fan Yang, Jilong Xue, Lili Qiu, Lidong Zhou, Quan Chen, Haisheng Tan, and Minyi Guo. 2023. Optimizing Dynamic Neural Networks with Brainstorm. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 797–815. https://www.usenix.org/conference/osdi23/presentation/cui
[11] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 2021. Enable simultaneous DNN services based on deterministic operator overlap and precise latency prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.
[12] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Eds.), Vol. 29. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2016/file/04df4d434d481c5bb723be1b6df1ee65-Paper.pdf
[13] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.
[14] Kaihua Fu, Jiuchen Shi, Quan Chen, Ningxin Zheng, Wei Zhang, Deze Zeng, and Minyi Guo. 2022. QoS-Aware Irregular Collaborative Inference for Improving Throughput of DNN Services. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1109/SC41404.2022.00074
[15] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 551–568. https://www.usenix.org/conference/osdi21/presentation/gandhi
[16] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. 2022. DynaGraph: Dynamic Graph Neural Networks at Scale. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences &*

*Systems (GRADES) and Network Data Analytics (NDA)* (Philadelphia, Pennsylvania) *(GRADES-NDA '22)*. Association for Computing Machinery, New York, NY, USA, 10 pages. https://doi.org/10.1145/3534540.3534691

[17] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687* (2020).

[18] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 187–198. https://proceedings.mlsys.org/paper/2020/file/fe9fc289c3ff0af142b6d3bead98a923-Paper.pdf

[19] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E. Leiserson, and Jie Chen. 2022. Accelerating Training and Inference of Graph Neural Networks with Fast Sampling and Pipelining. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 172–189. https://proceedings.mlsys.org/paper/2022/file/35f4a8d465e6e1edc05f3d8ab658c551-Paper.pdf

[20] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. *SIGARCH Comput. Archit. News* 45, 1 (apr 2017), 615–629. https://doi.org/10.1145/3093337.3037698

[21] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392. https://doi.org/10.1137/S1064827595287997 arXiv:https://doi.org/10.1137/S1064827595287997

[22] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. 2020. Representation learning for dynamic graphs: A survey. *The Journal of Machine Learning Research* 21, 1 (2020), 2648–2720.

[23] Srijan Kumar, William L. Hamilton, Jure Leskovec, and Dan Jurafsky. 2018. Community Interaction and Conflict on the Web. In *Proceedings of the 2018 World Wide Web Conference* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 933–943. https://doi.org/10.1145/3178876.3186141

[24] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations *(KDD '05)*. Association for Computing Machinery, New York, NY, USA, 177–187. https://doi.org/10.1145/1081870.1081893

[25] Jia Li, Zhichao Han, Hong Cheng, Jiao Su, Pengyun Wang, Jianfeng Zhang, and Lujia Pan. 2019. Predicting path failure in time-evolving graphs. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1279–1289.

[26] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. 2018. Diffusion Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting. In *International Conference on Learning Representations (ICLR '18)*.

[27] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN Training on Large Graphs via Computation-Aware Caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 401–415. https://doi.org/10.1145/3419111.3421281

[28] 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 *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 443–458. https://www.usenix.org/conference/atc19/presentation/ma

[29] Franco Manessi, Alessandro Rozza, and Mario Manzo. 2020. Dynamic graph convolutional networks. *Pattern Recognition* 97 (2020), 107000.

[30] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 76, 14 pages. https://doi.org/10.1145/3458817.3480856

[31] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3341301.3359646

[32] Nvidia. [n. d.]. Nvidia DGX-2 system user guild. https://docs.nvidia.com/dgx/dgx2-user-guide/index.html/.

[33] George Panagopoulos, Giannis Nikolentzos, and Michalis Vazirgiannis. 2021. Transfer graph neural networks for pandemic forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 4838–4845.

[34] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. 2020. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 5363–5370.

[35] S. Potluri, H. Wang, D. Bureddy, A.K. Singh, C. Rosales, and Dhabaleswar K. Panda. 2012. Optimizing MPI Communication on Multi-GPU Systems Using CUDA Inter-Process Communication. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 1848–1857. https://doi.org/10.1109/IPDPSW.2012.228

[36] Reddit. [n. d.]. Reddit Dataset. https://www.reddit.com/.

[37] Benedek Rozemberczki, Paul Scherer, Yixuan He, George Panagopoulos, Alexander Riedel, Maria Astefanoaei, Oliver Kiss, Ferenc Beres, Guzman Lopez, Nicolas Collignon, and Rik Sarkar. 2021. PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*. 4564–4573.

[38] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *European semantic web conference*. Springer, 593–607.

[39] Alex Sherstinsky. 2020. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *Physica D: Nonlinear Phenomena* 404 (2020), 132306.

[40] Xinkai Song, Tian Zhi, Zhe Fan, Zhenxing Zhang, Xi Zeng, Wei Li, Xing Hu, Zidong Du, Qi Guo, and Yunji Chen. 2022. Cambricon-G: A Polyvalent Energy-Efficient Accelerator for Dynamic Graph Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 1 (2022), 116–128. https://doi.org/10.1109/TCAD.2021.3052138

[41] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, 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 *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 495–514. https://www.usenix.org/conference/osdi21/presentation/thorpe

[42] Minjie Yu Wang. [n. d.]. Deep Graph Library: towards efficient and scalable deep learning on graphs. *ICLR Workshop on Representation Learning on Graphs and Manifolds* ([n. d.]). https://par.nsf.gov/biblio/10311680

[43] Yufeng Wang and Charith Mendis. 2023. TGOpt: Redundancy-Aware Optimizations for Temporal Graph Attention Networks. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 354–368. https://doi.org/10.1145/3572848.3577490

[44] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (jan 2021), 4–24. https://doi.org/10.1109/tnnls.2020.2978386

[45] Hongxia Yang. 2019. AliGraph: A Comprehensive Graph Neural Network Platform *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 3165–3166. https://doi.org/10.1145/3292500.3340404

[46] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: A Factored System for Sample-Based GNN Training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 417–434. https://doi.org/10.1145/3492321.3519557

[47] Jiaxuan You, Tianyu Du, and Jure Leskovec. 2022. ROLAND: Graph Learning Framework for Dynamic Graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 2358–2366. https://doi.org/10.1145/3534678.3539300

[48] 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. *Proc. VLDB Endow.* 13, 12 (sep 2020), 3125–3137. https://doi.org/10.14778/3415478.3415539

[49] Wei Zhang, Quan Chen, Kaihua Fu, Ningxin Zheng, Zhiyi Huang, Jingwen Leng, and Minyi Guo. 2022. Astraea: Towards QoS-Aware and Resource-Efficient Multi-Stage GPU Services. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 570–582. https://doi.org/10.1145/3503222.3507721

[50] Yanping Zheng, Hanzhi Wang, Zhewei Wei, Jiajun Liu, and Sibo Wang. 2022. Instant Graph Neural Networks for Dynamic Graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 2605–2615. https://doi.org/10.1145/3534678.3539352

[51] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81. https://doi.org/10.1016/j.aiopen.2021.01.001

# Appendix: Artifact Description/Artifact Evaluation

## ARTIFACT DOI

https://doi.org/10.5281/zenodo.8086109

## ARTIFACT IDENTIFICATION

The computation artifacts include a dynamic GNN training scheme named BLAD that achieves a high throughput and scalable dynamic GNN training. BLAD comprises a two-level load scheduler and an overlap-aware topology manager. The major part of our experiments is the evaluation of the training efficiency. For all evaluations, we use 3 dynamic GNN models, including WDGCN, TGCN, and EvolveGCN. Also, we use five datasets, including Arxiv, Products, Reddit, Reddit-body and AS. The three models and the five datasets are free combinations to form 15 benchmarks. For measuring training efficiency, we test the 15 benchmarks and record the duration of training one training set. The training set is defined as training a vertex set with 3000 randomly selected graph vertices. In all experiments, we run those benchmarks according to the experiment setup with a machine with 2 Nvidia RTX 2080 Ti GPUs, and two DGX-2 machines with Nvidia V100 GPUs.

## REPRODUCIBILITY OF EXPERIMENTS

### 0.1 Description

(1) How much time to prepare the environments: about 4 hours.
(2) How much time to execute the experiment workflow: about 96 hours.

### 0.2 Experiment workflow

(1) Profile the execution time of operations under different dynamic datasets and GNN models.
(2) Evaluate the training efficiency with BLAD, ESDG, and Aligraph using 3 involved dynamic models under 5 datasets.
(3) Evaluate the training accuracy and loss with BLAD and the origin scheme with 3 dynamic GNN models under Arxiv.
(4) Evaluate the communication overhead with BLAD, ESDG and Aligraph using 3 involved dynamic models under 5 datasets.
(5) Evaluate the training efficiency of 15 benchmarks with BALD, BALD-NM, and BALD-Even, two variants of BLAD disabling the topology manager and the load-balanced strategy, respectively.
(6) Evaluate the training efficiency on scalable DGX-2 machines with BLAD, ESDG and Aligraph using 15 benchmarks.

### 0.3 Evaluation and expected results

- Expected results: Improving training efficiency compared to Aligraph and ESDG. And an almost identical final training accuracy compared to the baseline.
- How the expected results from the evaluation relate to the results in the paper: The results presented in the paper can be obtained by plotting the output data into figures using the scripts provided in the code.

## ARTIFACT DEPENDENCIES REQUIREMENTS

### 0.4 Artifact check-list (meta-information)

- **State transition diagram**
- **Program: python**
- **Compilation: gcc, nvcc**
- **Model: WDGCN, TGCN, EvolveGCN**
- **Data set: Arxiv, Products, Reddit, Reddit-body, AS**
- **Run-time environment:**
- **Hardware: NVIDIA RTX 2080 Ti GPUs, Nvidia Tesla V100 GPUs**
- **Metrics: Training efficiency**
- **How much disk space required (approximately): 400G**
- **How much time is needed to prepare workflow (approximately): 4 hours**
- **How much time is needed to complete experiments (approximately): 96 hours**
- **Publicly available: https://github.com/fkh12345/BLAD.git**
- **DOIs to the artifact: https://doi.org/10.5281/zenodo.8086109**
- **Code licenses (if publicly available): MIT**

*0.4.1 Hardware dependencies.*

- CPU1: Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
- GPU1: Nvidia RTX 2080 Ti GPUs
- DGX-2 CPU: Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz
- DGX-2 GPU: Nvidia Tesla V100s-SXM3

*0.4.2 Software dependencies.*

- Ubuntu: 18.04.6 (kernel 4.15.0)
- GPU Driver: 470.57
- CUDA: 11.1
- CUDNN: 8.0
- PyTorch: 1.9.0
- DGL: 0.8.0

*0.4.3 Data sets.* We use 5 datasets (Arxiv, Products, Reddit, Reddit-body, AS) to evaluate the artifact. Reddit-body and AS are the real dynamic graph workloads that are widely used in public research. Meanwhile, we also use 3 widely-used static graph datasets (Arxiv, Products, Reddit) to represent more various graph topologies and input features on larger graphs.

## ARTIFACT INSTALLATION DEPLOYMENT PROCESS

We provide our reproduce runtime in a docker container and can enable an easy reproducibility process. And our artifact can be used in the following steps: 1. Download the container image and start the runtime environment. How to access: docker pull midway2018/blad_runtime. 2. Switch the python version to the preinstalled runtime environment and install the essential dependencies by executing pip install -r requirements.txt. 3. The code of the artifact can be deployed by cloning the code on the github repository into the container.