DGS: Communication-Efficient Graph Sampling for Distributed GNN Training

Xinchen Wan[1] Kai Chen[1] Yiming Zhang[2]
[1]*iSING Lab, Hong Kong University of Science and Technology*
[2]*NICEX Lab, Xiamen University*

***Abstract*—Distributed GNN training tends to generate huge volumes of communication. To reduce communication cost, the state-of-the-art sampling-based techniques sample and retrieve only a subset of the nodes. However, our analysis shows that current sampling algorithms are still inefficient in network communication for distributed GNN training, which is mainly because of three problems: first, they overlook the locality of the sampled neighbor nodes in the cluster; second, they sample data only at the coarse-grained *graph node* level; and third, some mechanisms they adopted fall short in distributed scenarios.**

This paper presents a graph sampling framework (*DGS*) for distributed GNN training, which effectively reduces network communication cost while preserving the final GNN model accuracy. To achieve this, DGS samples neighborhood information based on the locality of the neighbor nodes in the cluster, and samples data at the levels of not only graph nodes but also node features based on *explanation*. Specifically, DGS constructs an explanation graph which preserves the relationship between the local graph and remote nodes, and leverages the recently-proposed model explanation technique to design an online explanation scheme that interprets the importance of nodes and features. Evaluation results show that DGS achieves up to 1.25× throughput speedup over the state-of-the-art FastGCN and reduces the communication cost by up to 28.3%, while preserving the final model accuracy almost the same as that of full-batch training.

I. Introduction

Graphs are pervasive in many real-world applications, such as social network analysis [10], molecular structure generation [33], and recommendation systems [8]. Recent years have witnessed a surge of works that extend deep neural networks (DNNs) to extract structural information from graphs. These new methods, known as Graph Neural Networks (GNNs), have achieved satisfactory performance in various graph-related tasks including node classification, link prediction, and graph classification [7], [16], [24], [36].

GNNs express structural information by combining classical NN operations (*e.g.*, convolution and matrix multiplication) with iterative neighborhood aggregation. Figure 1 illustrates the computation process upon one node: a GNN layer (i) aggregates the embeddings of the node's one-hop neighbors, and (ii) transforms the aggregated embedding through NN operations to update the node's embedding. The two operations iterate across each of the n GNN layers, and the node's last-layer embedding contains information from all its n-hop neighbors.

978-1-6654-8234-9/22/$31.00 ©2022 IEEE

Training GNNs in large graphs is challenging because the neighborhood aggregation procedure can involve a large number of multi-hop neighbor nodes, quickly exceeding the memory limitation of a single device. Therefore, large GNN training tasks are usually processed in a distributed manner, where the graph is first partitioned into subgraphs and then trained in parallel (each subgraph on one worker) with necessary communication. However, while the memory constraint is relieved, iterative neighborhood aggregation makes inter-worker communication a bottleneck for GNN training. As the number of neighbor nodes can be exponentially large when increasing the number of GNN layers, the total communication volume increases drastically and affects the training efficiency significantly.

To reduce communication overhead in distributed GNN training, researchers have proposed the sampling-based techniques [21], which sample and retrieve only a subset of the features and thus generate less communication than obtaining the whole set of features. However, our analysis (§II) finds that current sampling algorithms [3], [4], [7], [30], [34], [40] are still far from being communication-efficient for distributed GNN training, mainly because of the following problems.

First, existing sampling algorithms overlook the locality of the neighbor nodes in the cluster, because they are initially designed for low memory footprint in *mini-batch training* on a single worker but not for distributed scenarios. Simply migrating these algorithms to distributed GNN training overlooks the difference between accessing local and remote nodes, and thus leads to extra communication traffic.

Second, existing sampling algorithms sample data at the relatively coarse-grained *node* level to guarantee high training accuracy. However, our experiments (§II-D) show that for popular sampling algorithms like GraphSAGE random neighbor sampling [7], as many as 30% features can be reduced with no more than 0.5% accuracy loss.

Third, some mechanisms adopted in sampling algorithms can severely lower the performance of distributed training. For example, LADIES [40] requires layer-wise adaptive compute and update of global sampling probability during training. While the overhead of such update is tolerable in a single worker, it becomes overwhelming in distributed scenario because of the frequent and heavy synchronization of probability tensors among workers.

Based on the above analysis, in this paper we study the graph sampling problem for efficient distributed GNN training.

The basic idea is to minimize the communication overhead of each GNN training iteration without affecting the accuracy. To this end, we propose *DGS*, a distributed graph sampling framework that samples neighborhood information at the levels of both graph nodes and node features while preserving the final GNN model accuracy. DGS first constructs an explanation graph which preserves the relationship between the local graph and remote nodes to facilitate the explanation process. It then designs an online explanation scheme that interprets the importance of nodes and features leveraging the state-of-the-art explanation technique [31] with minor system overhead.

We implement and evaluate DGS over 4 real-world datasets with popular GNN models [7], [16], [24]. Our results demonstrate that DGS can speed up distributed GNN training process by 1.25×-4.01×, while preserving the final model accuracy almost the same as that of full-batch training.

We summarize our contributions in this paper as follows:

- We analyze and identify the factors that determine the communication overhead in distributed GNN training, and illustrate the limitations of existing sampling algorithms when applied in a distributed scenario.
- We propose a novel sampling framework (DGS) which takes the locality of nodes in the cluster into consideration and applies sampling at both node and feature levels.
- We implement DGS and validate its effectiveness with extensive experiments. Experimental results demonstrate the effectiveness of DGS: it achieves up to 1.25× speedup over the state-of-the-art FastGCN and reduces the communication cost by up to 28.3%, while preserving the final accuracy almost the same as full-batch training.

II. Background and Motivation

A. Graph Neural Networks (GNNs)

GNNs emerge as a family of neural networks that perform *representation learning*: they take a graph as input and map each node into a d-dimensional vector, *a.k.a.*, an *embedding*. The embeddings are then used as inputs for downstream machine learning tasks, such as node classification, link prediction, and graph classification [7], [16], [24].

The computation process of GNN at one layer is illustrated in Figure 1. The GNN layer first aggregates the embeddings of the neighbor nodes calculated from the previous GNN layer, then applies classical NN operations such as matrix multiplication or convolution upon the aggregated result, and finally updates the embedding of that node. Formally, the neighborhood aggregation and NN operations are expressed as follows:

$$a_v^{(k)} = Aggregate^{(k)}(\{h_u^{(k-1)} \mid u \in \mathcal{N}(v)\}) \quad (1)$$

$$h_v^{(k)} = Update^{(k)}(h_v^{(k-1)}, a_v^{(k)}) \quad (2)$$

where $h_v^{(k)}$ is the embedding of node v at the k-th GNN layer, $a_v^{(k)}$ denotes the activation output of the aggregated results, and $\mathcal{N}(v)$ represents the neighbors of v in the graph. For each node v at layer k, *Aggregate* first outputs $a_v^{(k)}$ by gathering the embeddings of its neighbors with an accumulation function, and then *Update* computes the node's new embedding from its previous embedding $h_v^{(k-1)}$ and aggregation result $a_v^{(k)}$. The above operations iterate from layer 1 to K.

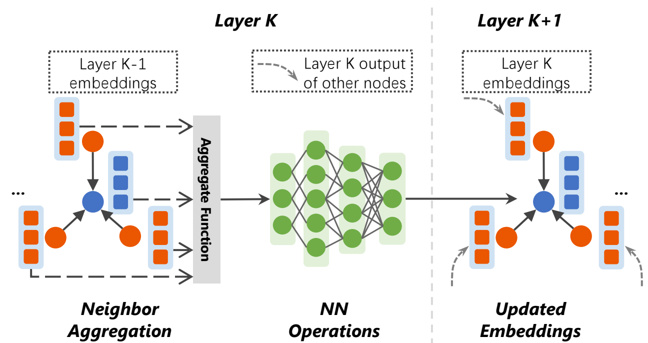


Fig. 1: Computation process upon one node in GNN layer K by first aggregating its neighbors' embeddings, and then applying NN operations for its layer K output embedding.

B. Sampling Algorithms

Sampling algorithms are originally designed for mini-batch training to reduce the memory requirement [3], [7]. Mini-batch training allows for a small memory footprint during training and strikes a balance between small memory footprint and fast model convergence [23]. For GNNs, however, the iterative neighborhood aggregation tends to explode the GPU memory during training. To alleviate the memory issue, sampling algorithms are proposed to down-sample multi-hop neighborhood features of each mini-batch. For example, GraphSAGE [7] and VR-GCN [30] applies neighbor sampling upon each mini-batch. FastGCN [3], AS-GCN [11], and LADIES [40] adopts layer-wise sampling strategies which generate sampling decision layer-by-layer. ClusterGCN [4] and GraphSAINT [34] use subgraph sampling to extract subgraphs as training samples.

C. Distributed GNN Training

As graphs are becoming too large to fit into a single device, training tasks over them are usually processed in a distributed manner, *i.e.*, distributed GNN training (DGT).

Following the experience of distributed DNN training [14], DGT typically applies data parallelism among workers to parallelize the training process. In such paradigm, the input graph is partitioned into subgraphs with classical partition strategies [15], and the subgraphs are then sent to different workers. When the training process starts, each worker simultaneously trains GNN over its assigned graph partition and synchronizes model parameters at each iteration.

Due to the neighborhood aggregation introduced in §II-A, communication between workers not only contains information of conventional parameters/gradients, but also information of neighbor embeddings for aggregation. From the view of a worker in DGT, to compute all its nodes' embeddings at layer K, it requires all its neighbors' embeddings at layer K-1, which again requires their neighbors' embeddings at layer K-2 recursively until layer 1. The amount of data needed grows

exponentially with the number of GNN layers, resulting in drastically increased communication volume and making DGT a network-intensive workload [12], [23].

Several techniques [12], [13], [18], [23], [27] have been proposed to alleviate the problem, among them the adoption of sampling in distributed scenario is a more promising one [21]. The idea of down-sampling neighbors at each subgraph helps decrease the communication volume across workers in DGT with minor model accuracy degradation [21], [37], thus effectively accelerating the global training process. As a matter of fact, sampling techniques have already been widely adopted in multiple distributed GNN systems [17], [37], [38].

Although the application of sampling to the distributed scenario seems natural, our analysis shows that current sampling algorithms are still far from communication-efficient for DGT.

D. Problems of Existing Sampling Algorithms

DGT Communication Cost Analysis. Given an input graph, DGT system divides it into n subgraphs and distributes them to workers for K-layer GNN training. The feature size at each node is $\mathcal{F}$, and $\mathcal{N}^j_{\mathcal{G}_i}$ denotes subgraph $\mathcal{G}_i$'s jth-hop neighbor nodeset. Following the iterative aggregation scheme in Equation 1,we formalize the communication cost of worker W_i as follows:

$$Comm(W_i) = (|\bigcup_{j=1}^{K} \mathcal{N}^j_{\mathcal{G}_i} - \mathcal{G}_i|) \times |\mathcal{F}| \quad (3)$$

Note that here we omit the communication cost caused by model synchronization because the network traffic generated by GNN models is trivial compared with the traffic generated by iterative neighborhood aggregation [37].

By summing up the communication cost of all workers, we obtain the total communication cost:

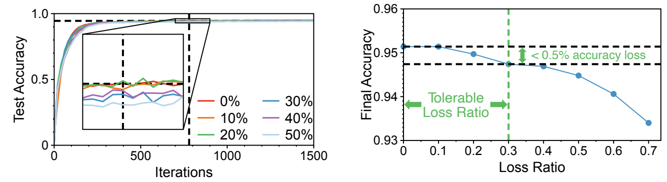
$$Comm_{total} = \sum_{i=1}^{n} Comm(W_i) = \sum_{i=1}^{n} n^i_{rmt} \times |\mathcal{F}| \quad (4)$$

where n^i_{rmt} denotes the size of $\mathcal{G}_i$'s K-hop neighborhood from remote workers.

Based on Equation 4, we find out that the size of remote neighbor nodes and the size of features are the root cause of the high communication cost in DGT. Below we elaborate our three observations that prevent existing sampling algorithms from being communication-efficient.

Observation 1: *Existing sampling algorithms overlook the locality of the neighbor nodes in the cluster.* All sampling algorithms can be categorized as an optimization strategy that decreases the number of neighbor nodes at each worker for lower network traffic. These algorithms, however, consider all neighbor nodes the same but overlook the different overhead of accessing local and remote nodes. Such overlook may lead to extra communication overhead caused by accessing the remote nodes which contribute trivial to model convergence. Therefore, the locality of neighbor nodes in the cluster indicates a promising communication optimization opportunity.

Observation 2: *Existing algorithms only sample data at the coarse-grained node level rather than the fine-grained feature level.* Our second observation originates from the second factor in Equation 4, *i.e.*, the feature size $\mathcal{F}$. As all existing algorithms sample data at node-level, we consider the question of whether sampling data can be done at a finer granularity (*i.e.*, feature-level) with minor model accuracy compromise? As the features in many popular graph datasets [7], [9] are pre-processed and expressed as sparse vectors, feature-level sampling may have no effect on the final accuracy. To verify this property, we run an experiment that trains a 2-layer GraphSAGE over Reddit [7], a popular graph dataset. We use the classical sample strategy of GraphSAGE, *i.e.*, 25 for the first layer and 10 for the second layer. Note that as the average node degree of Reddit is 984, our setting is an aggressive sampling strategy at node-level. During training, we randomly lose some values of the sampled nodes' features (*i.e.*, set those values to 0) with varying loss probabilities.



(a) Time-to-accuracy result. (b) Final accuracy result.

Fig. 2: Impact of random feature loss on model convergence: the 2-layer GraphSAGE converges to a tolerable final accuracy (no more than 0.5% accuracy loss) within the same iterations when the loss ratio is below 40%.

The result is shown in Figure 2, from which we can see that even after we apply an aggressive sampling strategy at node-level, as much as about 20% random loss of features has little impact on the model performance: the model can still achieve the same final accuracy within the same iterations. Besides, when the ratio is below 30%, the model can converge to approximately the same final accuracy with no more than 0.5% accuracy loss, within the same iterations. Therefore, the idea of sampling data at feature-level also indicates a promising optimization opportunity for DGT.

Note that in the above experiment we lose the features randomly without any intelligent selection. If we identify and select those features that may provide major contributions to model convergence, we can filter out more "unimportant" features from neighboring nodes to achieve much less communication traffic while still preserving the model accuracy.

Observation 3: *Some mechanisms adopted in existing algorithms may degrade the training performance.* Most sampling algorithms are designed by AI researchers with the assumption of running algorithms in a single host. However, when the scenario goes distributedly, some mechanisms may become a poor fit. For example, LADIES [40] requires layer-wise adaptive computation and update upon the global sampling probability during training. While the update overhead is trivial in a single worker, it becomes severe in distributed scenario due to the heavy synchronization of probability tensor among workers. Our experimental results (§V-A) also demonstrate that such

overhead caused by synchronization can severely degrade the overall performance (up to 2.94× worse performance).

E. Opportunity and Challenges

Opportunity: The above three observations inspire us to design a new sampling algorithm for DGT which targets reducing the *less important* communication cost for each worker. That is, for a given subgraph, we need to identify a subset of remote nodes and features from its multi-hop neighborhood that contributes little to prediction, while keeping minimizing the synchronization frequency as much as possible in mind.

A recent study, GNNExplainer [31], is a natural fit for our problem. Its goal is to identify a subgraph G_s from a complete graph G_c and the associated features X_s from X_c that are important for the induced GNN's prediction $\hat{y}$. To achieve this, the model formulates the importance of node and feature by quantifying the change in the probability of prediction $\hat{y} = \Phi(G_c, X_c)$. This metric, known as mutual information (MI), is defined as:

$$max_{G_s} MI(Y, (G_s, X_s)) = H(Y) - H(Y|G = G_s, X = X_s) \quad (5)$$

where $H(Y)$ is the entropy term of the prediction. The above equation can be explained as: if masking one node from G_c strongly decreases the probability of prediction $\hat{y}$, MI will become large, indicating that the removed node is important for the prediction. The feature mask can be explained in a similar way. The correctness of this model has been thoroughly verified in [31].

Though the explanation technique can help identify the important nodes and features for training, unfortunately, it is infeasible to run the explanation technique directly for DGT, mainly because of the following two challenges:

Challenge 1: *The original explanation is not designed for a subgraph.* We seek to find out a technique that can help us interpret the importance of remote nodes and sub-features upon a local subgraph, so that we can fetch the more important information and reduce the "unimportant" communication in DGT. However, the entity for which the explanation technique works is one single node in the graph or all nodes that have the same labels, rather than all nodes in the subgraph that are attached with different labels as expected. Directly using such technique is not feasible for our problem. A proper transformation of the subgraph is needed such that the transformed graph both preserves the original subgraph structure information and can be applied with explanation.

Challenge 2: *The original explanation training process is offline.* The original explanation technique operates offline: by inputting a trained GNN model and the prediction upon the node to be explained, the explanation process trains an explanation module iteratively and finally outputs the importance of the nodes and features. Clearly, DGT requires online explanation during training to guide its sample strategy. Further, even with online explanation the training process upon the explanation module may contend resources and hence affect the original GNN training performance. Therefore, it is necessary to parallelize the explanation process with GNN training while incurring minimal system overhead. Besides, the online explanation may require tensor synchronization among workers. As mentioned in observation 3, we should reduce such heavy synchronization as much as possible.

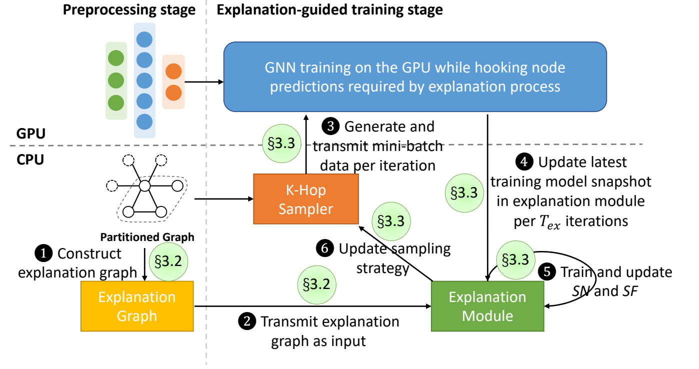


Fig. 3: DGS overview. DGS covers two stages (preprocessing and explanation-guided training stages) and has six operations (❶-❻) for training.

III. Design

A. DGS Overview

DGS consists of two stages, namely preprocessing stage and explanation-guided training stage. Figure 3 shows an overview of DGS from the perspective of a worker.

(1) *Preprocessing stage*: When a job is submitted, a DGS worker manages to construct an explanation graph (❶) based on its assigned partitioned graph and generates the explanation module in CPU. The generated explanation graph is then transmitted into the explanation module for the latter online explanation (❷). Besides, the worker is also responsible for some conventional DGT preparations such as communication setup and model construction.

(2) *Explanation-guided training stage*: During training, the k-hop sampler initiates with a pre-defined neighbor sampling strategy (*e.g.*, random sampling [7] or importance sampling [3]). The sampler requests and gathers sampled node features. It then attaches them to the generated computation graph, and transmits all required minibatch data to GPU for GNN training in each iteration (❸). After every T_{ex} iterations, DGS updates the GNN model duplicated in the explanation module using the latest snapshot of the training model, and infers the prediction of all nodes within the subgraph based on the model (❹). Note that we do not directly infer the predictions in GPU and then transfer out to CPU, as the size of GNN models is far smaller than the subgraph's node features, thereby reducing the transfer between CPU and GPU. Besides, the inference process in CPU is quick enough to satisfy our expectations. The explanation module in CPU then trains two masks, *i.e.*, sampled node mask (*SN*) and sampled feature mask (*SF*), given the node predictions and the loss function it predefined (in order to maximize MI between nodes) (❺). After the explanation module training is finished, *SN* and *SF*

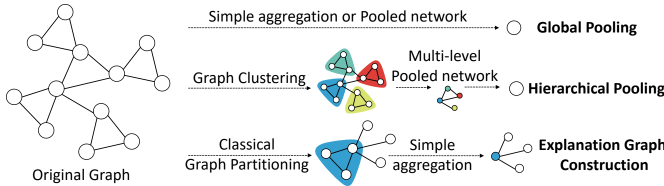


Fig. 4: Graph pooling and explanation graph construction. Explanation graph construction is based on hierarchical pooling but leverages classical graph partitioning and simplified aggregation for efficiency.

are transmitted to the k-hop sampler and interpreted to update sampling strategy (❻). The above operations repeat until the end of the training process.

B. Explanation Graph

As described in §II-E, current explanation technique [31] only supports explanation upon one node or a set of nodes that have the same labels, rather than all nodes in the subgraph as expected. Therefore, we should construct an explanation graph based on the given partitioned graph such that it can preserve the original graph structure information, especially the relationship between the local graph and remote nodes. We expect the explanation graph to have the following properties:

- Expressive graph structure: The explanation graph should be expressive to represent the relationship between local subgraph and remote nodes.
- Fast embedding interpretation: The operation for interpreting the predictions from the original graph toward the explanation graph should be fast. This is because such interpretation should be operated frequently (per T_{ex} iterations), hence a heavy interpretation may become a severe burden and degrade the performance.

We leverage graph pooling techniques [29], [32] and knowledge of graph partitioning [15] to achieve the above properties. Graph pooling is a popular technique used in GNN tasks, especially in graph classification tasks [32], to predict the label associated with the entire graph. It uses the generated node embeddings to encode the coarse-grained graph structure with a pre-defined differentiable pooling approach. As shown in Figure 4, there are two conventional types of pooling manners, namely, (i) global pooling all the node embeddings together, *e.g.*, using a simple summation/average or neural network that operates over the whole graph; and (ii) hierarchical pooling node embeddings to a set of clusters, with each level training a dedicated pooling neural network.

The hierarchical approach usually demonstrates better accuracy improvement [32], but at the cost of complex cluster assignment and sophisticated neural network layer. Leveraging graph partitioning, we adopt hierarchical pooling in an efficient way. We utilize the principle of balanced edge-cut for graph partitioning to cluster nodes, and simply employ the average pooling approach to encode the subgraph with local nodes' embeddings. Since classical partition algorithms such as METIS [15] are already capable of capturing community structure from graphs, we explicitly utilize the partition result so as not to introduce extra clustering overhead. Specifically, we identify the node type (local nodes and remote nodes), shrink all local nodes (*i.e.*, the community recognized by partition algorithm) into one subgraph node v_G, and construct links between it and all remote nodes. Besides, we add a self-loop edge toward v_G to guarantee that the node can help explain itself during explanation. While the average pooling operation upon embeddings is simple, recent works [29] and our evaluation results (see §V-B) demonstrate that the approach is sufficient for our expectation.

C. Online Explanation

In this subsection, we introduce the online explanation which helps us process the offline explanation technique in an online mode. We first describe the module plug-in designed to integrate the explanation module in DGS framework, and then elaborate the online explanation process in detail.

Module Plug-in. We directly employ the explanation module presented in [31], and use the output masks, *i.e.*, node mask (*SN*) and feature mask (*SF*), to update our sampling strategy. As introduced in §II-E, these masks represent the importance of nodes and features upon the subgraph node v_G induced in the explanation graph. DGS interprets *SN* and *SF* into two probability tensors, $\mathcal{SN}$ (sampled nodes) and $\mathcal{SF}$ (sampled features), based on the node mapping between the original graph and explanation graph. The value of each tensor in one dimension ranges between 0 and 1 and represents the importance of a remote node or a feature dimension. Thereafter, DGS is able to identify the important remote nodes and features leveraging these tensors, and sample them with built-in GNN system APIs and user-defined sampling values. As it may incur bias because we only sample remote nodes but remain all local nodes for GNN computation, we add ϵ (empirically ranging from 0 to 1/10 of the total sampled number) as a deviation towards the number of local sampled nodes and remote sampled nodes while remaining the number of total sampled nodes unchanged.

Online Explanation. The online explanation algorithm of DGS is described in Algorithm 1. During training, the main training process operates in GPU, and the explanation process operates in CPU to avoid GPU contention.

In the training process, the sampler first performs sampling to generate a computation graph of each mini-batch at the beginning of every iteration. Initially, it employs a pre-defined classical neighbor sampling strategy (line 5-6) which operates only in the first T_{ex} iterations, and switches to our distributed sampling if $\mathcal{SF}$ and $\mathcal{SN}$ are set by explanation process (line 7-8). During distributed sampling, it fetches top-N_f sub-features with respect to $\mathcal{SF}$, from the sampled nodeset regarding the probability tensor $\mathcal{SN}$ with deviation ϵ, and generates the sampled minibatch computation graph. DGS then transmits the computation graph from CPU to GPU to instantiate distributed GNN training process (line 9-12). The above operations iterate over time.

In the explanation process, the explanation module pulls the latest model from GPU every T_{ex} iterations, and updates its internal GNN model to infer node predictions (line 14). It then interprets the predictions and features based on the embedding interpretation described in §III-B (line 15-16). Note that the pulled model is the latest model that generates a minimal loss, indicating a more correct node and feature masks to be explained. To avoid GPU contention with the main training process, the explanation module $\mathcal{F}_E$ is trained in CPU and generates explanation loss epoch by epoch (line 17-21). If the loss is below a pre-defined threshold $\mathcal{L}$, the explanation process terminates and updates $\mathcal{SN}$ and $\mathcal{SF}$ to guide future distributed sampling (line 22). Otherwise, DGS updates $\mathcal{F}_E$ and $w_E^{t_e}$, and loops the explanation training process until the end of $\mathcal{T}_E$ epochs. Eventually, the explanation process terminates when the training process finishes.

IV. Implementation

We implement DGS on top of DGL [37], a popular open-source framework for DGT. We use DGL as a distributed framework for inter-worker message exchange and a graph propagation engine for graph-related operations, and use PyTorch [20] for neural network execution and model synchronization.

We extend DGL in multiple ways to support DGS design. First, we adopt DGL's probability-based sampling API and reuse its k-hop graph sampling service to achieve our sampling framework. We assign an appropriate tensor, *i.e.*, the tensor interpreted from $\mathcal{SN}$ (§III-C) to the service such that we can interpret the explanation toward DGS sampling strategy, which can then be executed in all corresponding workers.

Second, we add the feature selection function in DGL to support fine-grained feature sampling. We extend DGL's `KVStore` push/pull handler to support extracting the subtensors given the dimension and selected index in the pull requests. With this extension, the extracted subtensors can be replied via RPC library. We also extend the feature selection function in DGL's functional components, *e.g.*, `DistGraph` and `DistTensor`, for compatibility.

For system optimization, we use `sysv_ipc` [2] (a high-performance IPC library) to exchange the updated model and masks between processes. To minimize the IPC overhead, we use shared memory among processes to avoid extra memory copy. We pre-register shared memory to be used for training, and load/store updated objects when necessary during training.

Algorithm 1: Online explanation algorithm (a per-partition view)

```
Input:
partition id i, subgraph partition G_i, explanation graph
E_i, number of GNN layer K, feature dimension f,
node feature X_i, node predictions h_i, node sampling
size N_n, node sampling deviation ε, feature sampling
rate N_f, sampling update period T_ex, explanation
epoch T_E, loss threshold L, explanation module F_E,
initial model w^0
Output:
trained model w^T
1  begin
2    for t = 1...T do
3      do in parallel
         // Training process
4        for each batch do
5          if not set SN or SF then
6            Use conventional sampling strategy;
7          else
8            Fetch sampled nodeset with top-N_f
               sub-features according to SN, ε,
               and SF from remote workers;
9          Construct K-layer computation graph;
10         Process GNN computation;    ▷ in GPU
11         Process AllReduce to share gradients;
12         Update w^t;
         // Explanation process
13       for every T_ex batches do
14         Update latest model w^t as w_E^0;    ▷ in
             CPU
15         Inference node predictions h_{G_i} and
             interpret to h_{E_i} in E_i;
16         Update X_{E_i} in E_i;
17         for t_e = 1...T_E do
18           loss =
               F_E(E_i, X_{E_i}, h_{E_i}, p_n, p_f, w_E^{t_e});
19           if loss ≤ L then
20             break
21           Update F_E, w_E^{t_e};
22         Update SN and SF;
```

V. Evaluation

We evaluate DGS over several real-world graphs and compare it with state-of-the-art sampling algorithms. Overall, our results show that:

- DGS accelerates the training process by 1.25×-4.01×, and reduces the communication cost by up to 28.3%. The benefits DGS brings up increase with cluster size.
- DGS achieves almost the same final accuracy as full-batch training and without accuracy loss.
- DGS demonstrates its ability to explore complex GNN models against other sampling algorithms in DGT.

Experimental Setup: We evaluate DGS upon a GPU cluster with 4 physical servers, each equipped with 2 Tesla V100, 40 CPU cores (2.4GHz Intel Xeon Gold 5115), 128 GB RAM, and 2 Mellanox ConnectX5 NICs. The servers are interconnected via 4 Mellanox SN2100 switches running Onyx 3.7.1134 operating system. We virtualize an 8-node cluster by separating two docker containers at each server, where

	Reddit	Ogbn-products	Amazon	Ogbn-papers
Nodes	232.9K	2.45M	1.60M	111.1M
Edges	114.6M	61.86M	132.2M	1.616B
Features	602	100	200	128
Classes	41	47	107	172
Avg. Deg	984.0	50.5	82.7	29.1

TABLE I: Graph datasets used in our evaluation.

each container is equipped with one dedicated GPU, 20 CPU cores, 64GB RAM, and 10Gbps virtual Ethernet interface[1]. All servers run 64-bit Ubuntu 18.04 with CUDA library v11.0, DGL v0.6.1, and PyTorch v1.10.1.

Baseline: We choose several state-of-the-art sampling algorithms for comparison. Random sampling [7] performs random selection upon neighbors, which is the default sampling algorithm adopted in DGL. FastGCN [3] interprets graph convolutions as integral transforms and uses Monte Carlo approaches for importance sampling. LADIES [40] adopts a layer-wise sampling algorithm which considers the previously sampled nodes for calculating layer-dependent sampling probability. ClusterGCN [4] exploits the graph clustering structure and allows each worker to independently train GNN model upon its assigned subgraph.

Note that except Random, all other works are open-sourced for single machine only[234]. Therefore, we modify the three algorithms to process distributedly based on the example code of DGL[5]. Such modification only helps process the algorithm in a distributed manner and has no effect on the final accuracy. As ClusterGCN does not require feature exchange between workers and hence always achieves linear throughput, we only compare with it in terms of accuracy (§V-B).

Datasets: Table I lists four real-world graph datasets that we used in our evaluation, including Reddit [7], a dense online discussion forum dataset, Ogbn-products [9], an undirected product co-purchase graph, Amazon [34], a multi-label amazon co-purchasing network, and Ogbn-papers [9], a billion-edge directed citation graph. We run multi-class classification tasks on Reddit, Ogbn-products, and Ogbn-papers, and binary classification tasks on Amazon.

Models & Metrics: We evaluate DGS with three representative models: GCN (Graph Convolutional Network) [16], GraphSAGE [7], and GAT (Graph Attention Network) [24]. We use the default model hyper-parameters as [7], *i.e.*, 2 layers and 16 hidden dimension per layer.

Parameters Setup. We set the batch size to 1024 and the sampling strategy with fanout $\{64, 64\}$ (adopted in [40]) in all experiments. For DGS-related parameters, we set the node sampling size N_n at each layer the same to $\{64, 64\}$, the node sampling deviation ϵ to 0.1, the feature sampling rate N_f to 0.85, the sampling update period T_{ex} to 30, the explanation epoch $\mathcal{T}_E$ to 50, and the loss threshold $\mathcal{L}$ to 0.01 by default. The above settings ensure that the explanation module converges in our experiments.

[1]We use SR-IOV to separate the resource of physical NIC. [5] shows that it achieves nearly the same performance as the non-virtualized environments.

[2]https://github.com/acbull/LADIES/blob/master/pytorch_ladies.py#L95

[3]https://github.com/acbull/LADIES/blob/master/pytorch_ladies.py#L127

[4]https://github.com/dmlc/dgl/tree/master/examples/pytorch/cluster_gcn

[5]https://github.com/dmlc/dgl/blob/master/examples/pytorch/graphsage

A. Overall Performance

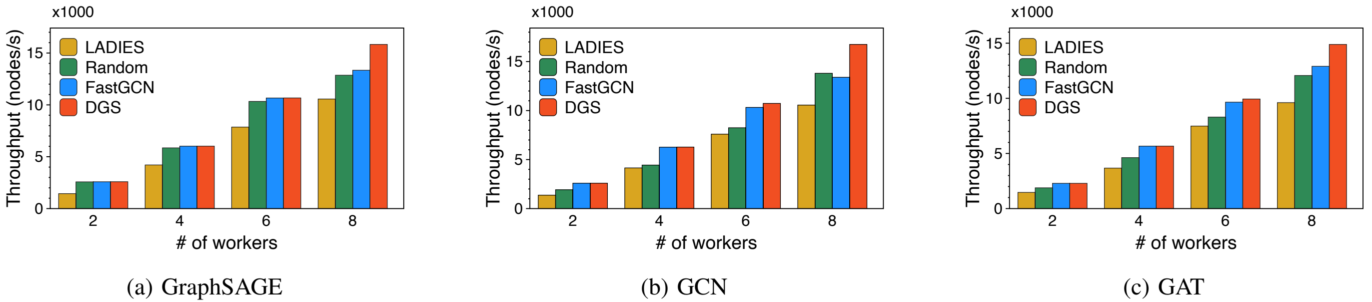
Due to the limited memory in each node (64 GB), we are not able to run large graphs like Ogbn-papers in small-scale clusters. Hence, we first present the scalability of each sampling algorithm over other 3 datasets. Then, we show the training performance over Ogbn-papers in an 8-node cluster. Besides, we also show the communication cost per epoch.

Scalability. The scalability results are shown in Figure 5, 6, and 7. We find that DGS outperforms all baselines in all settings with the speedup of 1.10×-3.24×, and the benefit increases with the cluster size. In general, LADIES performs worst among the three baselines due to the frequent probability synchronization of LADIES, as stated in II-D. Specifically, LADIES performs as much as 3.24× worse than DGS over Amazon, as the number of synchronizations increases with large graphs. Random and FastGCN do not perform such synchronization during training: Random just samples nodes uniformly at random; FastGCN pre-computes and synchronizes the probability tensor before training, and then remains the value unchanged throughout training. Hence, the two methods perform better and achieve comparable performance in all experiments. However, we still see that they perform up to 1.25× worse performance than DGS because of their communication-inefficiency: without the locality awareness of neighbor nodes and adopting coarse-grained node-level sampling only, the two methods bring up extra and redundant communication, which degrades the global training throughput. For DGS, though it needs to synchronize the probability tensor when necessary, its update period is much coarser: one-time update every $T_{ex} = 30$ iterations versus the layer-size× updates per iteration in LADIES. Overall, with the communication-efficient mechanisms adopted in DGS, specifically, both node- and feature-sampling, and the coarse-grained synchronization mechanisms, DGS gains more benefit for training than all other algorithms. We expect that DGS can benefit the training process with more workers involved in.

Ogbn-papers. The training performance is shown in Table II. Overall, DGS achieves 1.07×-4.02× speedup in 3 GNN models over Ogbn-papers. We find that DGS outperforms all baselines with the speedup of 1.07×-4.02×. DGS reduces more redundant communication costs than other algorithms with the help of the communication-efficient mechanisms. We expect that DGS can benefit training over even larger graphs.

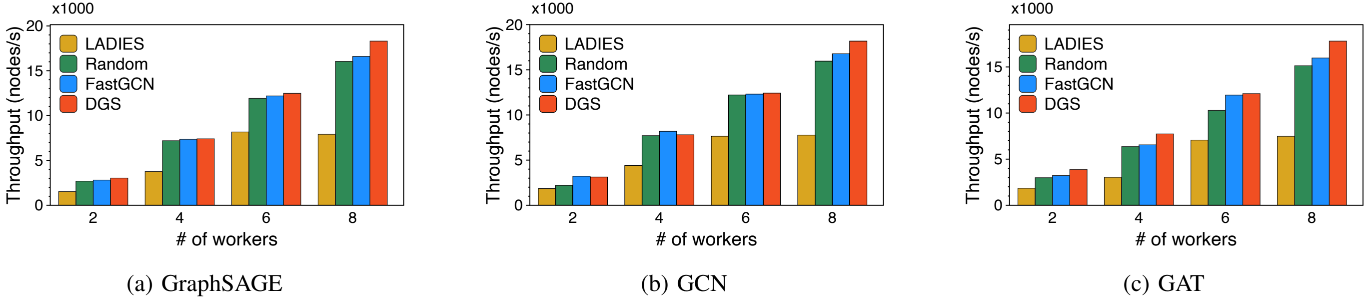
Communication Cost. We also present the communication cost during training. We log the counter of the NIC to obtain the communication cost/epoch result when training GraphSAGE over three datasets in an 8-node cluster. We also deep-dive the node-sampling (NS-only) and feature-sampling (FS-only) mechanisms.

The results are shown in Table III. DGS reduces the communication cost by 24.05%, 24.55%, 28.33%, and 24.89%



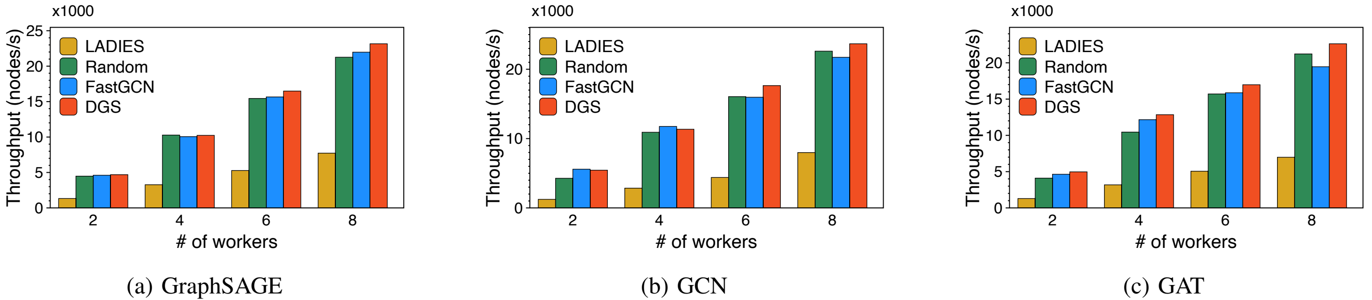
(a) GraphSAGE (b) GCN (c) GAT

Fig. 5: Throughput comparison with existing sampling-based methods in 3 GNN models over Reddit.



(a) GraphSAGE (b) GCN (c) GAT

Fig. 6: Throughput comparison with existing sampling-based methods in 3 GNN models over Ogbn-products.



(a) GraphSAGE (b) GCN (c) GAT

Fig. 7: Throughput comparison with existing sampling-based methods in 3 GNN models over Amazon.

	LADIES	Random	FastGCN	DGS	Speedup
GraphSAGE	10.43	31.42	36.72	41.57	1.13-3.99×
GCN	10.46	30.71	39.19	42.00	1.07-4.02×
GAT	9.580	28.38	34.76	38.09	1.09-3.98×

TABLE II: Overall throughput (Knodes/s) when training over Ogbn-papers.

Method	Reddit	Ogbn-products	Amazon	Ogbn-papers
Sampling-based	11.030	15.168	24.896	8.302
NS-only	9.312	12.962	20.390	7.057
FS-only	10.026	14.119	22.194	7.809
DGS	**8.377**	**11.445**	**17.843**	**6.236**

TABLE III: Communication cost (GB) per epoch. NS-only represents DGS adopts node-sampling only. FS-only represents DGS adopts feature-sampling only. Note that the communication cost of Ogbn-papers is small because in fact only a subset of Ogbn-papers is involved in training.

for Reddit, Ogbn-products, Amazon, and Ogbn-papers, respectively, which demonstrates the communication efficiency of DGS. We also show the communication cost with NS-only and FS-only. As expected, the result shows that node-sampling is more useful for communication reduction, as node-sampling may eliminate the whole node feature transmission while feature-sampling only reduces a subset (15%) of it.

B. Final Accuracy

Next we present the final accuracy. We compare DGS with baselines, including ClusterGCN, in terms of final accuracy when training GraphSAGE over 4 datasets. We add each baseline with a random feature sampling rate 0.85, noted as +RF, to show the performance of our feature-level sampling. For DGS, we show the results by varying the number of partitions[6] and T_{ex} as the two parameters may affect the final accuracy in DGT. We also add the full-batch training (mini-batch without sampling) accuracies as our target accuracies.

The results are shown in Table IV. The full-batch accuracies are: 95.17% in Reddit, 70.08% in Ogbn-products [7], 62.62% in Amazon, and 43.64% in Ogbn-papers. As expected, LADIES achieves the highest accuracies among baselines in almost all datasets due to its adaptive layer-wise importance sampling. FastGCN has lower accuracies than LADIES. ClusterGCN

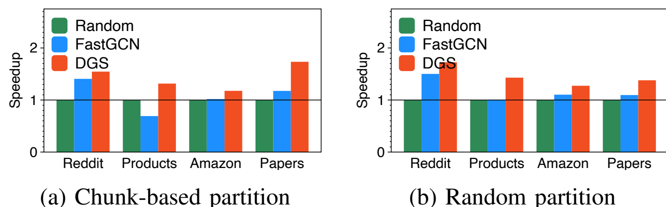
[6]For Ogbn-papers, we only train it over 8-node due to memory constraint.

[7]The accuracy is different from the result reported in [1] because we use 2-layer GraphSAGE and they use 3-layer.

Method	Reddit	Ogbn-products	Amazon	Ogbn-papers
Full-Batch	**95.17**	**70.08**	**62.62**	**43.64**
Random	93.97	69.07	61.16	41.24
Random+RF	93.69	68.02	58.06	39.41
FastGCN	95.00	68.30	61.28	41.41
FastGCN+RF	94.91	67.73	58.76	39.04
LADIES	95.05	69.37	62.28	41.62
LADIES+RF	94.70	68.77	59.24	40.86
ClusterGCN	92.49	67.86	62.35	26.19
ClusterGCN+RF	92.12	67.48	57.37	24.72

DGS	Reddit				Ogbn-products				Amazon				Ogbn-papers
# of workers	2	4	6	8	2	4	6	8	2	4	6	8	8
T_{ex} = 30	94.57	94.80	95.02	95.03	69.88	69.23	69.51	**69.77**	62.36	62.33	62.38	62.35	**41.86**
T_{ex} = 60	94.36	95.09	**95.09**	95.05	69.01	69.02	69.03	69.02	62.41	62.43	62.42	**62.44**	40.72
T_{ex} = 90	94.80	95.06	95.00	94.95	68.98	68.95	69.03	68.98	62.30	62.28	62.20	62.34	40.77

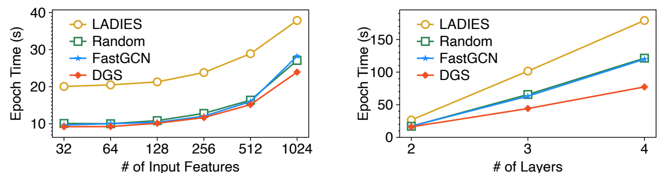
TABLE IV: Comparison of test accuracy (%) on 4 datasets, where DGS under different numbers of partitions are shown. **The accuracies of full-batch and the highest accuracy of DGS in terms of # of workers are in bold.**



(a) Chunk-based partition

(b) Random partition

Fig. 8: Impact of Partitioning Strategy.



(a) DGS's benefit increases as feature size increases.

(b) DGS's benefit increases as the number of layer increases.

Fig. 9: Impact of feature size and number of layers.

behaves differently in different datasets: in Amazon it achieves as much as 62.35% comparable accuracy as LADIES, but 92.49% and 26.19% low accuracy in Reddit and Ogbn-papers. It is because ClusterGCN depends heavily on the graph structure and the partition strategy may not be optimal. The accuracies of Random are the worst as it does not recognize the importance of each node. Besides, all baselines with RF face accuracy loss, because the random feature sampling cannot identify the important features during training and hence may sample the important ones and degrade the accuracy. DGS achieves similar accuracies as full-batch. As the table shows, DGS has no preference regarding the number of workers. This is because DGS adopts METIS partition which shows great ability in capturing graph community information regardless of the number of partitions. Besides, we find that DGS achieves the highest accuracy in most cases when T_{ex} is moderately 60, rather than what we expected $T_{ex} = 30$. We assume it is because the frequency of strategy may also affect the final accuracy, and a frequent update may adversely prevent the model from converging. Moreover, with infrequent updates upon sampling ($T_{ex} = 90$), DGS achieves lower accuracy compared to others, but still outperforms most baselines.

C. Impact of GNN Parameters

Next we present the effect of several GNN parameters, *i.e.*, partitioning strategy, feature size, and number of layers, to demonstrate the ability of DGS to explore complex GNN models in DGT. By default, we conduct all experiments when training GraphSAGE over Reddit in an 8-node cluster.

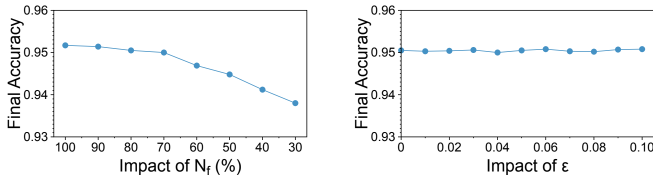
Impact of Partitioning Strategy. We first evaluate the effect of partitioning strategy. We adopt chunk-based strategy [39] and random strategy over 4 datasets and compare DGS with baselines. Note that we omit LADIES here because it performs far worse than others.

Figure 8 shows the throughput results (normalized by the throughputs of Random sampling). We see that DGS always achieves the best among all, with the average speedup of 1.44× and 1.45× in chunk-based and random settings, respectively. The results demonstrate that DGS is able to accelerate DGT via reducing communication, regardless of the choice of partition strategies.

Impact of Input Features. Next we evaluate the effect of input feature size. We vary the input features of Reddit from 32 to 1024-dimension and compare DGS with baselines. We use *epoch time* as the metric.

Figure 9a shows the throughput results. We see that DGS outperforms baselines in all settings, with the speedup ranging from 2.2% to 54.89%. Especially when feature size reaches 1024, DGS achieves 11.6%-51.03%× throughput speedup. We remind the readers that the communication cost is proportional to the size of input features, hence the communication ratio increases and shifts the training bottleneck from computation to communication. Without the awareness of nodes' locality in Random, FastGCN, and LADIES, the extra communication introduced by "unimportant" nodes from remote workers can result in significant communication overhead. DGS, however, takes into account the locality of nodes and reduces communication leveraging both node- and feature-level sampling.

Impact of Layers. We also evaluate the effect of the number of GNN layers. We create variants of GraphSAGE with 2, 3, and 4 layers and reuse the sampling strategy at each layer, *i.e.*, sample 64 nodes per layer. We use epoch time as the metric.



(a) Using a lower N_f may hurt the final accuracy.

(b) Using a varied ϵ has little effect on the final accuracy.

Fig. 10: Sensitivity analysis of DGS parameters.

Method	CPU Util.	GPU Util.	Memory
with OE	1380.2%	3337MB	20.4GB
without OE	1218.1%	3267MB	18.4GB

TABLE V: Resource Utilization of Online Explanation.

Figure 9b shows the throughput results. Compared to the increase of input features, the increase of layers affects GNN training more severely. It is because the increase of layers affects not only the communication volume among workers for more hop neighbor nodes' features, but also the computation workload of GNN, which ultimately limit the exploration for more complex but powerful GNN models. While DGS only optimizes the communication part, it achieves the lowest epoch time in all settings, with the speedup of 35.5% -56.8% in 4-layer experiments. Specifically, the epoch time of DGS increases gracefully compared to other baselines, indicating that DGS is capable of facilitating the exploration of deeper GNN model.

D. Sensitivity Analysis

We evaluate parameters of DGS, N_f and ϵ in terms of accuracy. We vary the value of N_f from 100% to 30%, the value of ϵ from 0 to 1/10, in GraphSAGE training over Reddit.

Impact of feature sampling rate N_f on accuracy. Figure 10a shows the results of N_f. We find that N_f has no effect in accuracy from 100% to 70%, but starts to degrade the accuracy starting from 60% (from 95.17% to 94.69% and lower). Such phenomenon echos our motivation results in §II-D: the graph data is sparse and has various redundant information (30% or more information can be reduced without hurting accuracy).

Impact of node sampling deviation ϵ on throughput. Figure 10b shows the results of ϵ. We find that in general, ϵ has no effect on accuracy from 0 to 1/10. This is because METIS partition results in a minimal relationship of local subgraph with remote nodes. Hence the bias which may occur in DGS that prefer local nodes to remote nodes has no effect upon the final accuracy.

E. Resource Utilization of Online Explanation

Finally, we measure several resource utilizations of DGS when training with and without online explanation (OE) with GraphSAGE over Reddit. Note that we set T_{ex} to a high value, *i.e.*, 2000, such that the explanation process will not be triggered during the epoch for the w/o explanation case.

As shown in Table V, the CPU utilization and memory consumption with OE are 162.1% and 10.9% higher than those without OE, while the GPU utilization almost remains the same. This is because the online explanation processes entirely in CPU and does not consume GPU resources.

VI. Related Work

GNN Framework. Various works have emerged to improve GNN training. For full-graph training, NeuGraph [18] and ROC [13] process full-graph embeddings synchronization layer-by-layer. As they do not use sampling during training, they may suffer from significant device memory constraints when performing large graph training. For mini-batch training, various frameworks [6], [12], [17], [26], [35], [37], [38] have been extensively used either in academia or industry. Note that DGS can benefit all mini-batch training frameworks by reducing the communication cost.

GNN Training Optimization. Several GNN training optimizations have emerged in recent years. GNNAdvisor [27] explores GNN input properties and proposes optimizations such as workload management and GPU memory customizations. Marius [19] optimizes the data movement during training with partition caching and buffer-aware data orderings. CAGNET [23] proposes distributed-memory parallel GNN training algorithms and reduces the communication costs by dividing the feature vectors into small sub-vectors. BNS-GCN [25] explores partition-parallelism and applies a random boundary sampling strategy for distributed GNN training. However, it applies sampling randomly at node-level, but DGS applies sampling with explanation guidance at fine-grained feature-level. Dorylus [22] brings serverless techniques to GNN training. It breaks down a single training iteration into 4 stages and pipelines each partition based on the workflow of each stage. The above optimizations are orthogonal to DGS.

VII. Conclusion

This paper proposes DGS, a communication-efficient graph sampling framework for distributed GNN training. Its key idea is to reduce network communication cost by sampling neighborhood information based on the locality of the neighbor nodes in the cluster, and sampling data at the levels of not only graph nodes but also node features. DGS constructs an explanation graph that preserves the relationship between the local graph and remote nodes, and leverages the recently-proposed model explanation technique to design an online explanation scheme that interprets the importance of nodes and features. Our evaluation results show that DGS outperforms existing state-of-the-art sampling algorithms by up to 1.25×, and reduces the communication cost by up to 28.3% while preserving the final accuracy.

Acknowledgements

This work is supported in part by the Key-Area Research and Development Program of Guangdong Province (2021B0101400001), the Hong Kong RGC TRS T41-603/20-R, GRF-16215119, GRF-16213621, the NSFC Grant 62062005, the NSFC Grant 61872376, and the Turing AI Computing Cloud (TACC) [28].

REFERENCES

[1] OGB Leaderboards. https://ogb.stanford.edu/docs/leader_nodeprop/, 2021.

[2] System V IPC for Python. https://semanchuk.com/philip/sysv_ipc, 2021.

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

[4] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery; Data Mining*, KDD '19, page 257–266, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.

[6] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[7] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[8] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182, 2017.

[9] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[10] Xia Hu, Lei Tang, Jiliang Tang, and Huan Liu. Exploiting social relations for sentiment analysis in microblogging. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 537–546, 2013.

[11] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. Adaptive sampling towards fast graph representation learning. *arXiv preprint arXiv:1809.05343*, 2018.

[12] Anand Jayarajan, Jinliang Wei, Garth A. Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. In *Proceedings of Systems and Machine Learning (SysML)*, 2019.

[13] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems (MLSys)*, pages 187–198, 2020.

[14] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed {DNN} training in heterogeneous gpu/cpu clusters. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 463–479, 2020.

[15] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

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

[17] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 401–415, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: parallel deep neural network computation on large graphs. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 443–458, 2019.

[19] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). USENIX Association*, pages 533–549, 2021.

[20] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[21] Marco Serafini. Scalable graph neural network training: The case for sampling. *SIGOPS Oper. Syst. Rev.*, 55(1):68–76, June 2021.

[22] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. 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)*, pages 495–514, 2021.

[23] Alok Tripathy, Katherine Yelick, and Ayd n Buluç. Reducing communication in graph neural network training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[24] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[25] Cheng Wan, Youjie Li, Ang Li, Nam Sung Kim, and Yingyan Lin. Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. *Proceedings of Machine Learning and Systems*, 4, 2022.

[26] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks, 2020.

[27] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An adaptive and efficient runtime system for {GNN} acceleration on gpus. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 515–531, 2021.

[28] Kaiqiang Xu, Xinchen Wan, Hao Wang, Zhenghang Ren, Xudong Liao, Decang Sun, Chaoliang Zeng, and Kai Chen. Tacc: A full-stack cloud computing infrastructure for machine learning tasks. *arXiv preprint arXiv:2110.01556*, 2021.

[29] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2019.

[30] Rui Ye, Xin Li, Yujie Fang, Hongyu Zang, and Mingzhong Wang. A vectorized relational graph convolutional network for multi-relational network alignment. In *IJCAI*, pages 4135–4141, 2019.

[31] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnnexplainer: Generating explanations for graph neural networks. *Advances in neural information processing systems*, 32:9240, 2019.

[32] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *arXiv preprint arXiv:1806.08804*, 2018.

[33] Jiaxuan You, Bowen Liu, Rex Ying, Vijay Pande, and Jure Leskovec. Graph convolutional policy network for goal-directed molecular graph generation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 6412–6422, Red Hook, NY, USA, 2018. Curran Associates Inc.

[34] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019.

[35] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. Agl: A scalable system for industrial-purpose graph machine learning. *Proc. VLDB Endow.*, 13(12):3125–3137, August 2020.

[36] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *Advances in Neural Information Processing Systems*, 31:5165–5175, 2018.

[37] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: Distributed graph neural network training for billion-scale graphs. *arXiv preprint arXiv:2010.05337*, 2020.

[38] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: A comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730*, 2019.

[39] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 301–316, USA, 2016. USENIX Association.

[40] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. Layer-dependent importance sampling for training deep and large graph convolutional networks. *arXiv preprint arXiv:1911.07323*, 2019.