GraphScale: A Framework to Enable Machine Learning over Billion-node Graphs

Vipul Gupta
Bytedance
San Jose, USA

Xin Chen
Bytedance
San Jose, USA

Ruoyun Huang
Bytedance, USA
Seattle, USA

Fanlong Meng
Bytedance
San Jose, USA

Jianjun Chen
Bytedance
San Jose, USA

Yujun Yan
Dartmouth College
Hanover, USA

ABSTRACT

Graph Neural Networks (GNNs) have emerged as powerful tools for supervised machine learning over graph-structured data, while sampling-based node representation learning is widely utilized in unsupervised learning. However, scalability remains a major challenge in both supervised and unsupervised learning for large graphs (e.g., those with over 1 billion nodes). The scalability bottleneck largely stems from the mini-batch sampling phase in GNNs and the random walk sampling phase in unsupervised methods. These processes often require storing features or embeddings in memory. In the context of distributed training, they require frequent, inefficient random access to data stored across different workers. Such repeated inter-worker communication for each mini-batch leads to high communication overhead and computational inefficiency.

We propose GraphScale, a unified framework for both supervised and unsupervised learning to store and process large graph data distributedly. The key insight in our design is the separation of workers who store data and those who perform the training. This separation allows us to decouple computing and storage in graph training, thus effectively building a pipeline where data fetching and data computation can overlap asynchronously. Our experiments show that GraphScale outperforms state-of-the-art methods for distributed training of both GNNs and node embeddings. We evaluate GraphScale both on public and proprietary graph datasets and observe a reduction of at least 40% in end-to-end training times compared to popular distributed frameworks, without any loss in performance. While most existing methods don't support billion-node graphs for training node embeddings, GraphScale is currently deployed in production at TikTok enabling efficient learning over such large graphs.

CCS CONCEPTS

• **Computing methodologies** → *Shared memory algorithms*; *Massively parallel algorithms*; **Semantic networks**; • **Computer systems organization** → Reliability.

KEYWORDS

Graph Learning, Node Embedding, Distributed Training, Billion-node Graphs

ACM Reference Format:
Vipul Gupta, Xin Chen, Ruoyun Huang, Fanlong Meng, Jianjun Chen, and Yujun Yan. 2024. GraphScale: A Framework to Enable Machine Learning over Billion-node Graphs. In *Proceedings of ACM CIKM 2024 (CIKM'24).* ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Graph learning is a powerful way to generalize traditional deep learning methods onto graph-structured data that contains interactions (edges) between individual units (nodes). Despite the promising performance of machine learning methods that learn directly from graph data [36, 45], their adaption at modern web companies has been limited due to the sheer scale [22]. For example, the social network graph at Facebook includes over two billion user nodes and over a trillion edges [4], the users-products graph at Alibaba consists of more than a billion users and two billion items [35], and the user-to-item graph at Pinterest includes at least 2 billion entities and over 17 billion edges [37]. At Bytedance, we frequently observe graphs with over a billion nodes on data obtained from the social media platform TikTok.

Machine learning on graphs can be broadly classified into two categories. The **first** is supervised learning, where the training process ingests the node features, edge features, graph structure, etc., and the labels for the supervised learning task at hand, for example, node classification and link prediction. Graph Neural Networks (GNNs) have become the de-facto way of supervised learning on graphs, popularized by methods such as Graph Convolution Network (GCN) [36] and GraphSAGE [12]. GNNs have been applied to a broad range of applications, such as recommendation systems [41] and drug discovery [9], to obtain state-of-the-art results.

One critical bottleneck for mini-batch training of GNNs is the sampling phase, where for each vertex in the mini-batch, the sampler samples and fetches features of a subset of its neighboring nodes (and/or corresponding edges). The sampling phase can take a significantly longer time than the training phase due to a large amount of random data access and remote feature fetching, especially during distributed training of large graphs [6, 18, 24, 43]. This is illustrated through an example in Fig. 1, where we distribute a 5-node graph across three workers and the fanout for GNN training is [2,2][1]. In some cases, the incurred communication cost may account for 80% or more of the training time [1, 8, 27].

GraphScale addresses this issue by decoupling feature fetching from the graph sampling process, which involves separating the computation and storage elements by using different workers to do the computation and storage. This separation allows overlapping feature fetching with computation. Also, each computation worker (referred to as a trainer) uses the full topology information to streamline the feature fetching process. This reduces the number

[1]A fanout of $[F_1, F_2, ..., F_h]$ in GNN training specifies that in the i-th layer of aggregation, each node samples information from up to F_i neighbors for all $i \in [1,h]$. This hyperparameter controls the breadth of neighborhood sampling at each layer.

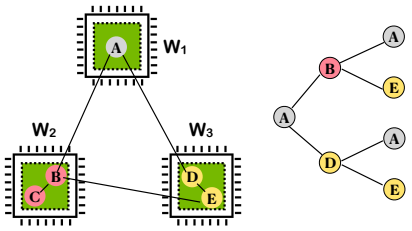


Figure 1: Traditional distributed GNN training of a five-node graph with 3 workers (W_1, W_2 and W_3) in frameworks like GraphLearn and DistDGL. On the right, we show the computation graph to update node A with a fanout of [2,2], which requires 5 cross-device visits to fetch neighbor data for aggregation (2 and 1 visits required at nodes B and D, respectively, in the second hop, and 2 visits required at node A in the first hop), causing extensive communication overhead.

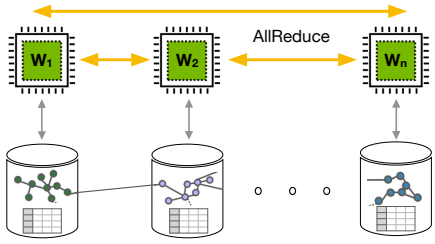


Figure 2: Traditional data parallel training of node embedding matrix where each trainer stores one graph partition and the entire embedding matrix, which can lead to memory bottlenecks. Further, high communication cost is incurred during allreduce while averaging gradients in each iteration.

of required feature requests, typically needing just one request per trainer for each mini-batch.

The **second** broad category is unsupervised learning to train graph embedding vectors. Working with graph data directly, especially for the purposes of machine learning, is complex, and thus, a common technique is to use graph embedding methods to create vector representations for each node. Node embedding methods are a way to convert graph data into a more computationally tractable form to allow them as an input to a variety of machine learning tasks [13]. The representations can then serve as useful features for downstream tasks such as recommender systems [35], link prediction [28], predicting drug interactions [46], community detection [2], etc. Hence, such methods are getting significant traction in both academia and industry [16].

One limitation of training node embeddings for large graphs is the size of the model[2]. For example, the node embedding matrix for a graph with one billion nodes and an embedding dimension of 256 would require 512 GB of memory in 16-bit precision floats[3]. Traditional distributed training methods that use data-parallelism hold the entire model in memory. In this case, the data (that is, the graph that contains node and edge connections) is partitioned into subgraphs using partitioning schemes, e.g., Metis [19], and each worker does mini-batch training from its own subgraph (as illustrated in Fig. 2). Such data-parallel training of node embeddings has two primary bottlenecks: 1) Requires large memory to store the model (as well as gradient and momentum) data during training, and 2) High communication cost while averaging gradients after every iteration due to the size of the model. Such methods are infeasible or highly inefficient for large graphs like the ones we observe at TikTok.

To ameliorate these bottlenecks, we propose hybrid (that is, both data and model) parallelism for training node embeddings as a part of the GraphScale framework. Here, we divide the graph into disjoint subgraphs across multiple trainers, as well as the model (that is, the embedding matrix) across storage workers. This allows us to scale both computation and storage independently with the size of the graph. Further, model parallelism enables each worker to handle only a subset of the model, reducing the need for full gradient communication per iteration and offering significant communication savings over data parallelism alone.

This paper proposes a framework called GraphScale to address and mitigate key bottlenecks for graph learning. The key features of GraphScale are:

- **Fast feature-fetching in GNN training**: GraphScale optimizes distributed GNN training by decoupling feature fetching from graph sampling, using distinct workers for computation and storage. This reduces feature requests and allows overlapping fetching with computation to alleviate communication bottlenecks.
- **Efficient node embedding training**: GraphScale uses hybrid parallelism for training node embeddings at scale. It uses data parallelism to divide computation between workers and model parallelism to reduce communication and storage requirements of the large node embedding matrix.
- **Serverless**: GraphScale, leveraging the capabilities of Ray–a framework designed for distributed serverless computation–offers significant advantages such as elasticity in resource allocation and failure tolerance [26]. With Ray, resources can be provisioned in real-time, and the complexities of managing them are abstracted away by Ray's user-friendly API. This approach ensures that users of GraphScale can focus on their training tasks, while Ray efficiently manages the underlying computational resources.
- **General Framework**: GraphScale is a unified framework that works on both supervised and unsupervised learning methods and is agnostic to the type of models and algorithms used (such as GraphSage and GCN for supervised learning and DeepWalk, Node2Vec and LINE for unsupervised learning). It is also backend agnostic and can work with multiple graph frameworks such as DGL, PyTorch-geometric, and GraphLearn.

2 PRELIMINARIES

In this section, we review the basics of GNN and node embedding training. We use the following notation throughout the paper. We want to perform learning over a graph $\mathcal{G} = (V,E)$ with $N = |V|$ nodes and $|E|$ edges. Each node has a feature vector of dimension f, and during supervised learning, the objective is to train a GNN based on the graph structure and the feature matrix $\mathbf{F} \in \mathbb{R}^{|V| \times f}$. During unsupervised learning, the objective is to learn the node embedding matrix $\mathbf{M} \in \mathbb{R}^{|V| \times d}$, where the embedding dimension for each node is d.

[2]In this paper, we use the term "model" interchangeably with the embeddings matrix in the context of unsupervised learning.

[3]Note that learning the node embeddings happens only for transductive methods. The embedding vectors are not trained for inductive methods, such as unsupervised Graphsage [12]. Instead, a neural network is trained to output embedding vectors for graph nodes based on their features and neighborhood subgraphs. However, such methods are not very good at capturing the structural information of the graph compared to transductive methods [20].

2.1 GNN Training

Mini-batch processing in GNN training can typically be divided into three main steps: (1) Sampling a subgraph from the original graph. The result is the vertex and edge IDs of the sampled nodes and edges, respectively. (2) Feature fetching corresponding to vertex/edge IDs of the sampled nodes/edges in the sub-graph, and (3) Training the GNN for that mini-batch using the sub-graph topology, and the features and labels for corresponding nodes and edges.

Current distributed training frameworks for GNNs suffer from data communication bottlenecks. Both GraphLearn (GL) [4] [38] and DistDGL [44], which are prominent distributed GNN training systems, exhibit a shared architectural approach in feature fetching. The process involves issuing requests from the graph backend to retrieve features for sampled vertices in each mini-batch. This methodology is inefficient for two primary reasons. Firstly, it generates a high volume of feature-fetching requests for each batch, potentially causing high network traffic. Secondly, due to the graph's power-law distribution, there is a high probability of duplicated vertices in neighboring hops. However, the system often processes features for these repeated vertices multiple times, resulting in increased network load. This is illustrated in Fig. 1.

2.2 Node embedding training

Methods such as skip-gram and negative sampling are popular for training graph node embeddings due to their ability to effectively capture complex graph structures and scale to large networks [3, 10, 28, 31]. These methods translate graph topologies into a learnable format by simulating random walks, enabling them to handle various types of relationships and interactions between nodes in a computationally efficient manner. A few notable algorithms for training node embedding algorithms are DeepWalk [28], Node2vec [10] and LINE [34]. DeepWalk trains node embeddings by generating random walks from each node in the graph and applying the skip-gram model from NLP for context-based training. Node2vec is an extension of DeepWalk that introduces a flexible, biased random walk to balance between breadth-first and depth-first traversals, allowing for more nuanced exploration and representation of graph structures. In LINE, the objective function utilizes information from both the local and global graph structures around a node to learn its embedding. In this paper, we use DeepWalk and LINE as representative node embedding algorithms to illustrate training with GraphScale.

As illustrated in Fig. 2, there are two bottlenecks to data-parallel training of node embeddings for large graphs, namely storage (to store the model, that is, the embeddings matrix $\mathbf{M} \in \mathbb{R}^{|V| \times d}$ in memory) and communication (to average gradients during each mini-batch update to the model).

3 GRAPHSCALE FOR LEARNING ON GRAPHS

GraphScale system design. In GraphScale, our reimagined system architecture for GNN training capitalizes on the separation of storage and computation. Throughout this paper, computation workers are referred to as trainers, and workers who store the data are referred to as actors (following Ray's terminology). Note that a single machine

[4]ByteGNN [43] is a further optimized version of GraphLearn, where sampling and feature fetching are phases of GNN training are parallelized, and the k-hop neighbor fetching is pipelined.

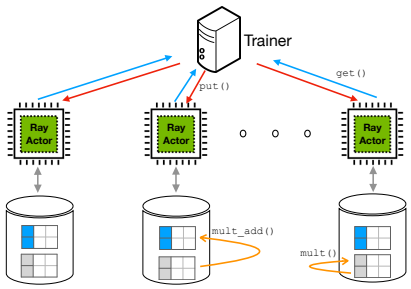


Figure 3: An illustration of the GraphScale system, where each Ray actor stores one shard each of two large matrices (such as the embedding and momentum data) in two separate GraphScale data structures. Trainers can use `get()` (`put()`) to read (write) data from (to) the actors. Operations `mult()`, and `mult_add()` are performed locally in actors' memory and in a parallel fashion.

can have tens or hundreds of CPUs/cores. Within the Ray framework, each of these cores is capable of functioning either as an actor or a trainer. Despite actors and trainers operating on the same machine, Ray's abstraction permits us to treat storage and computation as distinct entities. This approach not only maximizes efficient usage of resources (such as CPU memory and computational power) but also simplifies resource management for the user by abstracting away the intricacies of actual resource allocation. GraphScale is currently implemented in CPU-only environments, where we emphasize scalability in settings where GPUs are not readily available.

GraphScale system is illustrated in Fig. 3. Separating storage actors and trainers drastically cuts down communication overhead, as it necessitates only a single feature request for the sub-graph from each trainer. This efficiency is largely enabled by Ray's capacity to facilitate an easy abstraction of storage and computation in serverless systems. GraphScale storage actors also enable effective data updates by allowing operations like read, update, addition, and multiplication as illustrated in Fig. 3.

Implementation. GraphScale is implemented using the Ray actor model[5]. The system comprises 2,800 lines of Python code and 500 lines of C++ code. This implementation takes advantage of Ray's flexible scheduling features, allowing for straightforward integration of asynchronous operations and pipeline processing. The C++ component is specifically utilized to enhance the efficiency of indexing and constructing large data arrays.

3.1 GNN Training

Reducing feature requests by trainers. In GraphScale, the trainers keep the entire graph topology[6], but the heavy feature matrix is stored at the storage actors. To use network bandwidth efficiently during GNN training, the trainers combine multiple feature requests into a single batch at the end of each mini-batch training session. By keeping the entire graph topology, trainers can merge the same vertex IDs from subgraphs before making (a single) feature request to the actors. The actors then respond with one data block per request.

[5]https://docs.ray.io/en/latest/ray-core/actors.html

[6]Note that storing the topology locally in GraphScale can quickly become a bottleneck for large graphs. However, graph compression methods like GBBS [5] can be used to mitigate this bottleneck, as shown by the authors in [30], where they store a 74 billion node graph's topology in just 107 GB of memory.

PyTorch DDP model

```
class DDPModel(torch.nn.Module):
    def __init__(self, N, d):
        super().__init__()
        self._emb = torch.nn.Embedding(N, d)



    def forward(self, batch):

        # Computing model loss for nodes in "batch"
        return compute_loss(self._emb, batch)
```

GS-based model

```
1  class  GSModel(torch.nn.Module):
2      def __init__(self, N, n, d):
3          super().__init__()
4          self._emb = torch.nn.Embedding(n, d)
5          self.emb_data =  GSData("embedding", N, d)
6          self.mmt_data =  GSData("momentum", N, d)
7
8      def forward(self, batch):
9          # Copy embeddings from  GS  to the PyTorch model
10         self._emb.weight.data = self.emb_data.mget(batch)
11         return compute_loss(self._emb, batch)
```

Figure 4: Comparison of Python codes for PyTorch DDP versus GraphScale (GS) model definitions using PyTorch.

PyTorch DDP training

```
ddp_model = DDPModel(N=1000000000, d=128)
ddp_model = DistributedDataParallel(ddp_model)
for _ in range(num_epochs):
    for batch in enumerate(train_loader):
        loss = ddp_model(batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

GS-based training

```
1    gs_model =  GSModel(N=1000000000, n=5000, d=128)
2
3  for _ in range(num_epochs):
4      for batch in enumerate(train_loader):
5          loss =  gs_model(batch)
6          optimizer.zero_grad()
7          loss.backward()
8          gs_sgd_update( gs_model, lr, batch)
```

Figure 5: Comparison of Python codes for DDP versus GraphScale (GS) based graph node embedding training for PyTorch.

This method of consolidating requests and removing duplicate vertices greatly reduces the network bandwidth needed. It also reduces the dependence on the partitioning scheme, since regardless of the partition, only one communication request is needed per trainer. However, the size of the data to be communicated still depends on the partitioning scheme, but its effect on the overall training time is negligible. In Sec. 4.1, we show that training GNNs with GraphScale is at least 30% faster than distributed GNN training frameworks like DistDGL [44] and GraphLearn (also known as AliGraph) [38]. Next, we describe node embedding training with GraphScale.

3.2 Node Embedding Training

Like GNN training, GraphScale utilizes its storage actors to scale the embedding matrix with the size of the graph, mitigating the storage bottleneck in data-parallel training of the node embedding matrix. To mitigate the communication bottleneck, we exploit the fact that gradients are sparse, as observed below.

During training a mini-batch with B source nodes, let n be the maximum number of nodes obtained during graph sampling (including source nodes in mini-batch and their positively and negatively sampled nodes). The number n is typically easy to estimate, e.g., for DeepWalk, it is $B\times$ `walk_len`, where `walk_len` is the length of the random walk used for DeepWalk training, and for LINE, it is $B\times$(2+`num_neg`), where `num_neg` is the number of negatively sampled nodes per source node in the mini-batch. For example, a good batch-size and walk length for DeepWalk is $B=1000$ and `walk_len`=5, respectively. We generally have $N \gg n$, that is, the total number of nodes in the graph (e.g., $N = 1$ *billion*) is much greater than the number of nodes sampled in a mini-batch (e.g., $n=5000$). Thus, the gradients during node embedding training are typically sparse. This allows each trainer in GraphScale to communicate only an update of size n instead of size N in data-parallel training to update the model in each iteration as we describe next.

PyTorch's DistributedDataParallel (DDP) is a widely used method for distributed data-parallel training of large models and is also used in graph frameworks like GraphLearn (GL) [38]. We use DDP training as a baseline to illustrate node embedding training with GraphScale. Figures 4 and 5 detail the Python codes for model definition and training loops, highlighting implementation differences in yellow. Both methods are implemented on PyTorch.

Fig. 4 shows the common aspects of model definitions shared by DDP and GraphScale. In line 4, the PyTorch embeddings are initialized (however, with different sizes). In lines 5 and 6 in the GraphScale case, we initialize the GraphScale storage for node embeddings and momentum vector, respectively, where the dimension of GraphScale matrices is $N\times d$. This initializes the embeddings and momentum matrices distributedly using Ray. In line 11, `compute_loss` is a loss function that computes the loss using the current batch (which includes the source and positively and negatively sampled nodes).

Fig. 5 describes the pseudo-code for DDP and GraphScale-based PyTorch training loop. Note that the implementation for GraphScale-based training is as straightforward as only a few lines of code changes. The most notable points are:

- GraphScale requires a significantly smaller memory at each worker to hold the model. The size of the embedding matrix in `ddp_model` is $N\times d$ (Line 4, Fig. 4), where $N=1$ billion (Line 1, Fig. 5). In comparison, the `gs_model` in GraphScale has an embedding matrix of dimension $n\times d$, where $N \gg n$, e.g., $n=5000$ when the mini-batch size, $B=1000$, and the walk length is 5 in DeepWalk.
- In GraphScale, workers communicate smaller gradients after each iteration. DDP uses `ddp_model` to wrap PyTorch's DistributedDataParallel class for data-parallel training (Line 2, Fig. 5). This means

gradients are averaged before model updates, communicating an $N \times d$ matrix in DDP, compared to $n \times d$ in GraphScale.
- In GraphScale, each worker fetches only a subset of the model for each iteration. The loss (and the gradient) is computed using the sampled node embeddings for the local mini-batch at each worker and not all the nodes (Line 11, Fig. 4). Thus, each worker needs embeddings of n nodes instead of N nodes from the storage to compute loss on its local subgraph.
- GraphScale doesn't rely on a central parameter server for momentum and model updates by distributing them within its storage. Hence, instead of using PyTorch's optimizer, we implement custom optimizers like SGD with momentum [29] and Adam [21] within GraphScale. This is illustrated in Fig. 5 (Line 8) and Fig. 6.

In Fig. 6, we describe our implementation of model update in GraphScale for SGD with momentum. In Line 3, we obtain the gradient of the embedding vectors for nodes that are a part of the subgraph created by the corresponding mini-batch. In Line 7, we multiply the momentum data in GraphScale by the momentum value using the GraphScale `mult()` operation. In Line 11, we add the gradient corresponding to the nodes in the batch to update the momentum vector for that iteration using the GraphScale `put()` operation. Finally, in Line 18, the Ray actors perform an SGD update of the model by first multiplying the momentum data by the learning rate and then subtracting it from the current model data using the `mult_add()` operation. Note that operations such as `mult()`, `mult_add()`, `get()` and `put()` operations are completed by multiple Ray storage actors parallelly that store data like model and momentum and do not require any raw data communication (see Fig. 3 for an illustration).

Synchronization. There are two types of synchronizations required: across iterations and across trainers. To achieve synchronization across iterations, we add a barrier after the write operation at the end of each iteration (that is, after the trainers update embedding and momentum). This barrier allows trainers to read updated values for the next iteration. During the write operation, synchronization is also required among parallel trainers (as they might be updating embeddings for the same nodes). However, we do not apply strict synchronization between the trainers and allow them to write incremental updates to embeddings of the same nodes. This is because such asynchronous methods are faster in distributed settings due to no locking and have the same theoretical rate of convergence as synchronous SGD [32, 42]. Further, we evaluate the convergence of training with GraphScale in Section 4.2 and show that it performs on par with synchronized SGD while enjoying at least 70% speedup.

4 EXPERIMENTAL RESULTS

To evaluate GraphScale's performance, we divide our experiments into two parts: GNN algorithms and node embedding algorithms, respectively, in Section 4.1 and Section 4.2. For GNN training, we compare GraphScale against two popular distributed frameworks, GraphLearn (GL) [38] and DistDGL [44], in terms of throughput and latency with GraphSage [12] as a representative algorithm. For node embedding training, we compare GraphScale with PyTorch DDP and PyTorch BigGraph [22] with DeepWalk [28] and LINE [28] as representative algorithms.

Experimental setup: Each machine has 48 physical cores and 700 Gigabytes of memory. As a result, we get 48 actors per machine.

```
1  def  gs_sgd_update(model, lr, batch, momentum=0.9):
2      """Getting gradient after the backward pass"""
3      gradient = model._emb.weight.grad.data.numpy()
4
5      """Multiply momentum vector in  GS  by momentum
6      value: mmt_data = mmt_data * momentum"""
7      model.mmt_data.mult(momentum)
8
9      """Apply momentum vector update in  GS  storage:
10     mmt_data = mmt_data + gradient"""
11     model.mmt_data.put(batch, gradient, add=True)
12
13     """
14     Update the embeddings inside  GS  storage using
15     the updated momentum vector (also inside  GS ):
16     emb_data = emb_data - lr * mmt_data
17     """
18     model.emb_data.mult_add(model.mmt_data, -1 * lr)
```

Figure 6: Python code for SGD with momentum update when both the embeddings and momentum vectors are stored in GraphScale (GS) storage.

The distributed cluster consists of 4 such machines. The ethernet provides a communication bandwidth of 25 gigabits per second.

Datasets: From publicly available datasets, we use Reddit [12], ogbn-products [14] and ogbn-papers100M [14]. Reddit is a graph dataset extracted from Reddit posts made in the month of September 2014 with 232,965 nodes as posts and a feature dimension of 602. ogbn-products is an undirected graph that has approximately 2.5 million nodes and 62 million edges and represents an Amazon product co-purchasing network, while ogbn-papers100M has 111 million nodes and 1.6 billion edges and is a directed graph representing a paper citation network. Finally, we also implement GraphScale-based node embedding on a proprietary dataset from the TikTok social media platform with 1 billion nodes and 92 billion edges.

Configuration: Unless mentioned otherwise, we use the following as the default configurations. The batch size is 512. The number of epochs is 4. The number of trainers per machine is 2. The learning rate is (0.003) for GraphSage, (0.01) for DeepWalk, and (1.0) for LINE. The default sampling fanout for GraphSage is [15, 10]. We use a walk-length of 5 and a window-size of 3 for DeepWalk; and 5 negative nodes per source node for LINE.

4.1 Training GNNs with GraphScale

Experiments in this section compare the throughput of feature fetching, assessing its impact on latency, and comparing the overall performance of GraphScale with DistDGL and GL across all three stages–sampling, feature fetching, and training on the Reddit graph dataset.

We define throughput as the average number of features fetched per second. We vary two variables to evaluate GraphScale's scalability and throughput: the number of trainers and sampling fanout. Trainers are the computation workers doing feature fetching in parallel, while fanout represents the aggregation workload per batch. High fanout leads to higher accuracy but increased workload since more neighbors need to be aggregated.

Fig. 7 presents a comparison with GL on a cluster of 4 machines. With the number of trainers per machine increasing from 1 to 8, GraphScale has a steady growth while GL has a small throughput increase. With 8 trainers per machine, GraphScale's throughput is 3.73× of GL's. Next, we vary the sampling fanout from [10, 5] to [20, 15] (that is, now we aggregate 20 and 15 neighbors in the first and

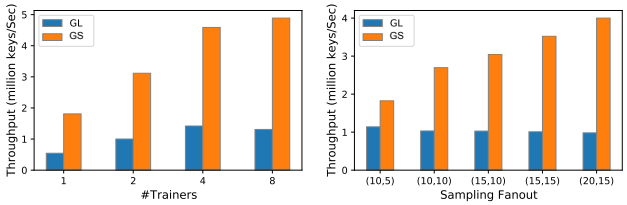


Figure 7: Throughput (in million vertices per second) for GraphSage algorithm with the Reddit dataset. GraphScale (GS) increases the throughput up to 4× when compared to GL.

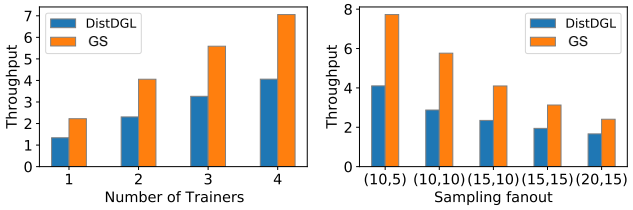


Figure 8: Throughput comparison between DistDGL and GraphScale (GS) on GraphSage with the Reddit dataset. GraphScale has significantly better throughput than DistDGL in all scenarios.

second hop, respectively). Unlike GL, the throughput increases commensurately with fanout (that is, computation load) for GraphScale. For the fanout of [20, 15], GraphScale has a throughput of 4× the throughput of GL. This is because GraphScale reduces communication time during feature fetching and mitigates the I/O bottlenecks to utilize computational resources more efficiently.

Similarly, Fig. 8 compares the performance of GraphScale with DGL. When the number of trainers increases, the throughput of both DistDGL and GraphScale grows proportionally. On average, GraphScale increases the training throughput by 71.7%. When the sampling fanout increases, both DistDGL and GraphScale have reduced throughput because both feature-fetching and training time increase due to the growing workload. On average, GraphScale improves DistDGL's throughput by 73.8%.

We also evaluated GraphScale on the ogbn-papers100M dataset and observed similar trends. Specifically, GraphScale achieved at least a 1.4× higher throughput compared to GL and DistDGL. Detailed results are omitted due to space limitations.

4.2 Node Embedding Training with GraphScale

In this section, we evaluate the performance of GraphScale on training node embeddings of graphs through experiments on several real-world datasets. Throughout the node embedding experiments, we use a Ray cluster of 4 machines and eight trainers while training. We perform our experiments on two popular node embedding algorithms, DeepWalk and LINE, to show the efficacy of GraphScale-based node embedding training.

GraphScale outperforms data-parallel training: First, we compare GraphScale with data-parallel training (that is, PyTorch DDP) which is employed in popular frameworks like GL and PyTorch Geometric. In Fig. 9, we plot the training loss for GraphScale and DDP with respect to the number of batches processed and the amount of time taken for the ogbn-papers100M datasets, with an embedding size of 32. The training statistics were averaged over three independent trials. We highlight two key observations below.

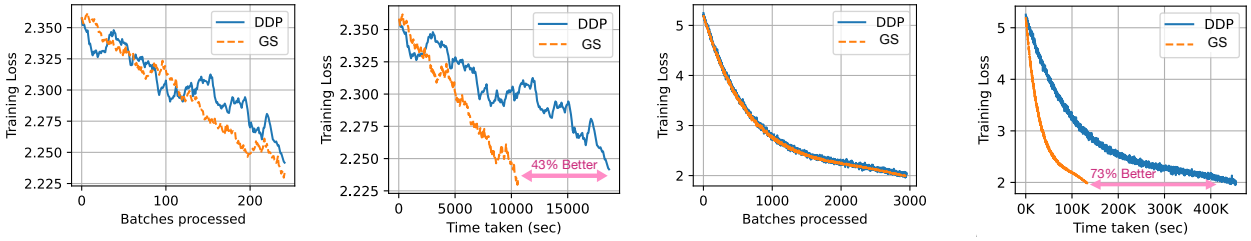
1. **Training Loss:** GraphScale has similar (if not better) training accuracy when compared with DDP. This is evident from figures 9a and 9c where we plot the smoothened training loss for DeepWalk and LINE, respectively.
2. **Training Runtime**: GraphScale is significantly faster than DDP. This is evident from figures 9b and 9d, where we observe savings of 43% for DeepWalk and 73% for LINE, respectively.

We also performed the same experiments on the ogbn-products datasets for both DeepWalk and LINE, and observed savings of 39% and 68%, respectively, without any loss in performance.

GraphScale saves both communication and computation time: In Fig. 10, we compare the average computation and communication time per iteration for GraphScale with DDP. Computation time primarily comprises of the forward and backward passes, and communication time comprises the optimizer and model updates (that is, allreduce in the case of DDP and GraphScale storage updates in the case of GraphScale). We note that GraphScale outperforms DDP significantly during both communication and computation. During communication, for large embedding sizes, the gradients become large too, and PyTorch DDP requires a lot of communication during allreduce. On the other hand, by virtue of only training nodes in the mini-batch and its sampled neighbors (that is, the subgraph for that mini-batch), GraphScale reduces the amount of communication required. During computation, GraphScale further takes less time since it performs operations (such as forward and backward passes) on a PyTorch tensor of the size of the subgraph, while DDP performs these operations on the entire embedding matrix. On a side note, we can also see that DeepWalk spends more time in communication, while LINE requires more time in computation.

GraphScale is scalable: In Fig. 11, we plot the total runtimes for the ogbn-products dataset with DDP and GraphScale for different embedding sizes and observe that GraphScale performs better as the embeddings get larger. GraphScale is slower for small embedding sizes since it requires two rounds of communication in each iteration, one for copying the current embedding values from GraphScale to memory and the second for updating the model in GraphScale after training. On the other hand, PyTorch DDP requires only one round of communication when the gradients are averaged. However, DDP is not scalable since it requires both higher communication and computation as it operates on the entire DDP matrix (as shown in Fig. 10 and explained earlier).

Comparison with PBG: PyTorch-BigGraph (PBG) [22] addresses the memory bottleneck in graph node embedding training by partitioning nodes into M buckets (where M is the number of machines). This divides the edges into M^2 buckets depending on the source and destination nodes. Thus, it enables large-scale embedding training when embeddings don't fit in memory by limiting training to disjoint edge buckets. PBG's architecture involves a lock server, parameter server, partition server, and shared filesystem, which can be complex to implement in production environments. In Figure 12, we compare PBG, DDP, and GraphScale in reaching a training loss of 4.45 with the same initial model values. PBG takes significantly longer than DDP and GraphScale due to several factors. PBG's positive and negative sampling is not independent and identically distributed as it samples only from the local bucket. Furthermore, PBG's restriction that only one machine can work on one node partition makes training not embarrassingly parallel, leading to underutilization of resources.



(a) Training loss for DeepWalk. (b) Training time for DeepWalk. (c) Training loss for LINE. (d) Training time for LINE.

Figure 9: Training stats with the ogbn-papers graph. GraphScale (GS) has the same accuracy as DDP but takes at least 43% less time.

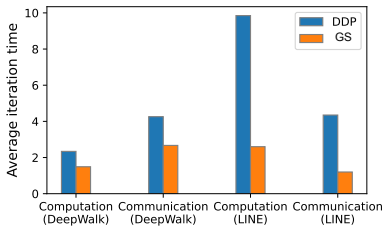


Figure 10: Runtime comparison for computation and communication phases with GraphScale and DDP for DeepWalk and LINE on the ogbn-products dataset (with embedding size 128). We observe commensurate savings with computation and communication.

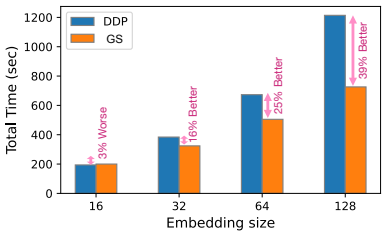


Figure 11: Total runtime comparison for different embedding sizes training DeepWalk on the ogbn-products dataset. Savings with GraphScale increase with embedding size due to less communication and computation when compared to DDP.

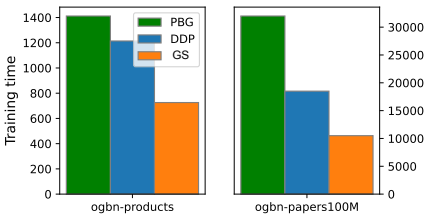


Figure 12: Total runtimes for PBG, DDP and GraphScale on the two datasets for DeepWalk to achieve the same training loss. PBG is worse than both DDP and GraphScale due to slower convergence and under-utilization of resources.

Industry-scale datasets: We trained DeepWalk on an industry-scale dataset featuring 1 billion nodes and 92 billion edges, using SGD with momentum (learning rate: 0.01, batch size: 2048, momentum: 0.9). Distributed training utilized a Ray cluster of 10 machines, each with 48 cores and 700 GB of memory, totaling 480 Ray actors serving both compute and storage. With an embedding size of 128, the size of the node embedding matrix becomes 256 GB in fp16 precision. Storing it along with the graph topology (and the momentum vector) at each node makes PyTorch DDP training infeasible. Thus, we employ GraphScale, which requires that each storage actor holds one-tenth of the embedding matrix (25.6 GB in this case). We observe that a mini-batch of size 512 takes 0.8 seconds on average, and training converges to a loss of 0.4 in slightly less than 3 hours.

GraphScale is currently deployed at TikTok and its Chinese counterpart Douyin, two of the world's largest social media platforms, for both supervised and unsupervised graph learning at scale.

5 RELATED WORK

Many single-machine systems have been proposed in the literature to tackle the sampling bottleneck in GNN training. PyTorch-Geometric [7] implements a message-passing API for GNN training. Using the Apache TVM compiler, FeatGraph [15] generates optimized kernels for GNN operators for both CPU and GPU. PaGraph [23] proposes a GPU caching policy to address the subgraph data loading bottleneck. NextDoor [17] proposes a graph-sampling approach called transit-parallelism for load balancing and caching of edges. [6] propose Global Neighborhood Sampling (GNS) for mixed CPU-GPU training, where a cache of nodes is kept in the GPU through importance sampling, allowing for the in-GPU formation of mini-batches. [18] uses a performance-engineered neighborhood sampler to mitigate mini-batch preparation and transfer bottlenecks in GNN training. We note that many of the above schemes for faster sampling for GNN training can be used complementarily with GraphScale.

The above single-machine systems are limited by CPU/GPU memory when processing large industrial-scale graphs. For GNN training on such large graphs, distributed training was utilized in DistDGL [44] and GraphLearn [38]. However, it introduces a key performance bottleneck: high sampling time due to random data access and remote feature fetching. As noted from our experiments in Sec 4.1, GraphScale is able to mitigate this bottleneck by separating the topology and feature storage in graphs.

Despite its industrial importance, research on node-embedding training for large graphs remains nascent. Authors in [40] propose tensor-train decomposition [39] to compress embeddings by enabling their compact parametrization. This method can easily be incorporated into node embedding training with GraphScale to further reduce memory and communication bottlenecks. Marius [25] was proposed for a single machine GPU training of node embeddings to reduce the data movement by leveraging partition-caching and buffer-aware data orderings. Moreover, techniques like gradient quantization and sparsification [11, 33] can be applied complementarily to GraphScale to reduce communication costs. PBG [22] proposed a distributed solution but complicated partition-aware distributed training that is not embarrassingly parallel. Also, it requires a large shared parameter server to store the node embeddings while training. As we observed in our experiments in Sec. 4.2, GraphScale outperforms PBG due to a better parallelization strategy that utilizes the subgraph structure while storing the node embeddings distributedly.

REFERENCES

[1] 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*. 130–144.

[2] Sandro Cavallari, Vincent W. Zheng, Hongyun Cai, Kevin Chen-Chuan Chang, and Erik Cambria. 2017. Learning Community Embedding with Community Detection and Node Embedding on Graphs. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management* (Singapore, Singapore) *(CIKM '17)*. Association for Computing Machinery, New York, NY, USA, 377–386. https://doi.org/10.1145/3132847.3132925

[3] Sudhanshu Chanpuriya and Cameron Musco. 2020. InfiniteWalk: Deep Network Embeddings as Laplacian Embeddings with a Nonlinearity. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2020). https://doi.org/10.1145/3394486.3403185

[4] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1804–1815.

[5] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2021. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)* 8, 1 (2021), 1–70.

[6] Jialin Dong, Da Zheng, Lin F Yang, and Geroge Karypis. 2021. Global neighbor sampling for mixed CPU-GPU training on giant graphs. *arXiv preprint arXiv:2106.06150* (2021).

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

[8] 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)*. 551–568.

[9] Thomas Gaudelet, Ben Day, Arian R Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy BR Hayter, Richard Vickers, Charles Roberts, Jian Tang, et al. 2021. Utilizing graph machine learning within drug discovery and development. *Briefings in bioinformatics* 22, 6 (2021), bbab159.

[10] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 855–864.

[11] Vipul Gupta, Dhruv Choudhary, Peter Tang, Xiaohan Wei, Xing Wang, Yuzhen Huang, Arun Kejariwal, Kannan Ramchandran, and Michael W Mahoney. 2021. Training recommender systems at scale: Communication-efficient model and data parallelism. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2928–2936.

[12] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[13] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584* (2017).

[14] 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. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[15] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. Featgraph: A flexible and efficient backend for graph neural network systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[16] Zexi Huang, Arlei Silva, and Ambuj Singh. 2021. A Broader Picture of Random-Walk Based Graph Embedding. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Virtual Event, Singapore) *(KDD '21)*. Association for Computing Machinery, New York, NY, USA, 685–695. https://doi.org/10.1145/3447548.3467300

[17] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating graph sampling for graph machine learning using GPUs. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 311–326.

[18] 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. *Proceedings of Machine Learning and Systems* 4 (2022), 172–189.

[19] 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.

[20] Megha Khosla, Vinay Setty, and Avishek Anand. 2019. A comparative study for unsupervised network representation learning. *IEEE Transactions on Knowledge and Data Engineering* 33, 5 (2019), 1807–1818.

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

[22] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. Pytorch-biggraph: A large scale graph embedding system. *Proceedings of Machine Learning and Systems* 1 (2019), 120–131.

[23] 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*. 401–415.

[24] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. 2021. Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing. *arXiv preprint arXiv:2112.08541* (2021).

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

[26] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 561–577.

[27] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. Sancus: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1937–1950.

[28] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 701–710.

[29] Ning Qian. 1999. On the momentum term in gradient descent learning algorithms. *Neural networks* 12, 1 (1999), 145–151.

[30] Jiezhong Qiu, Laxman Dhulipala, Jie Tang, Richard Peng, and Chi Wang. 2021. Lightne: A lightweight graph processing system for network embedding. In *Proceedings of the 2021 international conference on management of data*. 2281–2289.

[31] J. Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. 2017. Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec. *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining* (2017). https://doi.org/10.1145/3159652.3159706

[32] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems* 24 (2011).

[33] S. Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. 2018. Sparsified SGD with Memory. *ArXiv* abs/1809.07599 (2018).

[34] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. LINE: Large-Scale Information Network Embedding. In *Proceedings of the 24th International Conference on World Wide Web* (Florence, Italy) *(WWW '15)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1067–1077. https://doi.org/10.1145/2736277.2741093

[35] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 839–848.

[36] Max Welling and Thomas N Kipf. 2016. Semi-supervised classification with graph convolutional networks. In *J. International Conference on Learning Representations (ICLR 2017)*.

[37] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. 2014. Embedding entities and relations for learning and inference in knowledge bases. *arXiv preprint arXiv:1412.6575* (2014).

[38] Hongxia Yang. 2019. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery and data mining*. 3165–3166.

[39] Chunxing Yin, Bilge Acun, Carole-Jean Wu, and Xing Liu. 2021. Tt-rec: Tensor train compression for deep learning recommendation models. *Proceedings of Machine Learning and Systems* 3 (2021), 448–462.

[40] Chunxing Yin, Da Zheng, Israt Nisa, Christos Faloutsos, George Karypis, and Richard Vuduc. 2022. Nimble GNN Embedding with Tensor-Train Decomposition. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2327–2335.

[41] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery and data mining*. 974–983.

[42] Shen-Yi Zhao and Wu-Jun Li. 2016. Fast asynchronous parallel stochastic gradient descent: A lock-free approach with convergence guarantee. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 30.

[43] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. 2022. ByteGNN: efficient graph neural network training at large scale. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1228–1242.

[44] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. 36–44. https://doi.org/10.1109/IA351965.2020.00011

[45] 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.

[46] Marinka Zitnik and Jure Leskovec. 2017. Predicting multicellular function through multi-layer tissue networks. *Bioinformatics* 33, 14 (2017), i190–i198.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009