Distributed Hybrid CPU and GPU training for Graph Neural Networks on Billion-Scale Heterogeneous Graphs

Da Zheng
AWS AI
USA
dzzhen@amazon.com

Xiang Song
AWS AI
USA
xiangsx@amazon.com

Chengru Yang
AWS AI
China
ychengru@amazon.com

Dominique LaSalle
NVIDIA Corporation
USA
dlasalle@nvidia.com

George Karypis
AWS AI
USA
gkarypis@amazon.com

ABSTRACT

Graph neural networks (GNN) have shown great success in learning from graph-structured data. They are widely used in various applications, such as recommendation, fraud detection, and search. In these domains, the graphs are typically large and heterogeneous, containing many millions or billions of vertices and edges of different types. To tackle this challenge, we develop DistDGLv2, a system that extends DistDGL for training GNNs on massive heterogeneous graphs in a mini-batch fashion, using distributed *hybrid CPU/GPU* training. DistDGLv2 places graph data in distributed CPU memory and performs mini-batch computation in GPUs. For ease of use, DistDGLv2 adopts API compatible with Deep Graph Library (DGL)'s mini-batch training and heterogeneous graph API, which enables distributed training with almost no code modification. To ensure model accuracy, DistDGLv2 follows a synchronous training approach and allows ego-networks forming mini-batches to include non-local vertices. To ensure data locality and load balancing, DistDGLv2 partitions heterogeneous graphs by using a multi-level partitioning algorithm with min-edge cut and multiple balancing constraints. DistDGLv2 deploys an asynchronous mini-batch generation pipeline that makes computation and data access asynchronous to fully utilize all hardware (CPU, GPU, network, PCIe). We demonstrate DistDGLv2 on various GNN workloads. Our results show that DistDGLv2 achieves $2-3\times$ speedup over DistDGL and $18\times$ speedup over Euler. It takes only $5-10$ seconds to complete an epoch on graphs with hundreds of millions of vertices on a cluster with 64 GPUs.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

ACM Reference Format:
Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. 2022. Distributed Hybrid CPU and GPU training for Graph Neural Networks on Billion-Scale Heterogeneous Graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22), August 14–18, 2022, Washington, DC, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3534678.3539177

1 INTRODUCTION

Graph Neural Networks (GNNs) have shown success in learning from graph-structured data and have been applied to many graph applications in social networks, recommendation, knowledge graphs, etc. In these applications, graphs are usually huge and heterogeneous, in the order of many millions or even billions of vertices and edges of different types. Examples include Facebook's social network graph, Amazon's buyer-product graph and knowledge graphs such as Freebase.

A number of GNN frameworks have been introduced that take advantage of distributed processing to scale GNN model training to large graphs. These frameworks differ on the type of training they perform (full-graph training vs mini-batch training) and on the type of computing cluster that they are optimized for (CPU-only vs hybrid CPU/GPU). So far, few frameworks are designed to handle heterogeneous graphs with more than one vertex type and edge type. Distributed frameworks that perform full-graph training have been developed for both CPU- and GPU-based clusters [10, 17, 18, 20, 21], whereas distributed frameworks that perform mini-batch training are mainly developed/optimized for CPU-based clusters [1, 25–27]. Unfortunately, for large graphs, full-graph training is inferior to mini-batch training because it requires many epochs to converge and converges to a lower accuracy (cf., Sec 3.2). This makes approaches based on distributed mini-batch training the only viable solution for large graphs. However, before such mini-batch-based approaches can fully realize their potential in training GNN models for large graphs, they need to be extended to take advantage of GPUs' higher computational capabilities.

It is natural to ask whether GPUs have advantage of training GNN models on large graphs in a cluster of machines. The main challenges of GNN training on GPUs lie in two aspects. First, GNN mini-batch computations have much lower computation density in GPUs than traditional neural network models, such as CNNs and Transformers. In addition, GNN mini-batch sampling requires a large amount of computations in CPUs if the graph data is stored in CPU. Consequently, for very large graphs, since we cannot store the entire graph and all of its features in GPU memory, it is critical to devise efficient strategies for moving data from slower memory (e.g., CPU, remote memory, disks) to GPUs during training. The second challenge is load imbalance among mini-batches. Typically, neural network models are trained with synchronous stochastic gradient descent (SGD) to achieve good model accuracy, which requires a synchronization barrier at the end of every mini-batch iteration. To ensure good load balance, mini-batches have to contain the same number of vertices and edges as well as reading the same amount of data from slower memory. Due to the complex subgraph structures in natural graphs, it is difficult to generate such balanced mini-batches. The load balancing problem becomes even more severe on heterogeneous graphs because vertices of different types may be associated with different feature sizes.

In this work, we develop DistDGLv2 on top of DGL [23] to optimize distributed GNN training on heterogeneous graphs for hybrid CPU/GPU clusters, where it stores the graph structure and vertex/edge features in CPU memory and performs mini-batch computation in GPUs. To provide good user experience and to minimize accuracy differences between development and deployment of a model, DistDGLv2 provides Python API compatible with DGL's mini-batch sampling and heterogeneous graph API. Thus, it requires almost no code modification to DGL's training scripts to enable distributed training. To ensure the quality of GNN models, DistDGLv2 uses synchronized SGD and generates mini-batches with non-local vertices. It extends the design principles of DistDGL [26], a CPU-only distributed GNN training framework, to increase data locality and balance computation among trainers on heterogeneous graphs. To move data efficiently from the CPU memory to GPUs, it deploys a sophisticated asynchronous mini-batch sampling pipeline that sample mini-batches ahead of time to overlap CPU and GPU computation and data communication and utilize all hardware resources (CPU, GPU, network, PCIe) simultaneously. To further speed up the sampling process, we move some of the mini-batch sampling computations to GPUs.

We conduct comprehensive experiments to evaluate the efficiency of DistDGLv2. Overall, DistDGLv2 achieves 18× speedup over Euler-GPU and $2-3\times$ speedup over DistDGL-GPU, the modified version of Euler [1] and DistDGL [26] for GPU training, on a cluster of 32 GPUs. DistDGLv2 achieves up to 15× speedup over distributed CPU training by DistDGL in a cluster of the same size. This indicates that GPUs can be effective for GNN mini-batch training on massive graphs than CPUs. It takes 5 seconds per epoch to train GraphSage and GAT on a homogeneous graph with 100 million vertices and 13 seconds per epoch to train RGCN on a heterogeneous graph with 240 million vertices with 64 GPUs.

The main contributions of the work are listed below:

- We design an asynchronous mini-batch sampling pipeline with extensible Python sampling API and speed up distributed GNN training on hybrid CPU/GPU by a factor of $2-3\times$ over DistDGL-GPU without changing the training algorithm.
- DistDGLv2 is a distributed GNN framework that explicitly supports distributed heterogeneous graphs with very diverse vertex/edge features.

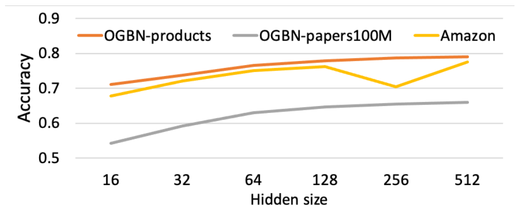


Figure 1: The model accuracy of GraphSage with different hidden sizes on datasets in Section 5.

- DistDGLv2 realizes all optimizations under DGL's API for ease of use and minimizing accuracy differences between development and deployment of a model.

2 RELATED WORK

Many works have been developed to scale GNN training on large graph data for distributed CPU- and GPU-based clusters. Many of them [10, 17, 18, 20, 21] are designed for distributed full-graph training on multiple GPUs or distributed memory whose aggregated memory fit the graph data. Even though full-graph training is easier to parallelize, it actually takes a longer time to converge on a large graph and may converge to a lower accuracy than mini-batch training (Section 3.2). Therefore, the focus of our work is to optimize mini-batch training.

Multiple frameworks have been developed for distributed GNN mini-batch training. Some of them [1, 25, 27] adopt distributed mini-batch training but does not use graph partitioning algorithms that minimize edge cut to reduce network communication. Their system is optimized for distributed training on a CPU cluster and many of their design choices (e.g., only using multiprocessing) are not suitable for GPU training. DistDGL [26] adopts METIS graph partitioning [12] to reduce network communication, but is not designed for GPU training. Frameworks, such as PyTorch-Geometric [4] and PaGraph [15], support multi-GPU training but cannot scale to graphs beyond the memory capacity of a single machine. P3 [5] is a distributed GNN framework designed for distributed training in a GPU cluster. It adopts model parallelism, which works better when the hidden size is small. In contrast, DistDGLv2 uses data parallel and works better for larger hidden sizes. As shown in Figure 1, a large hidden size is required to achieve good model accuracy. BGL [16] builds on top of DGL for distributed training. It heavily relies on changing the mini-batch sampling algorithm to increase data locality and GPU cache hits. This is orthogonal to DistDGLv2's design. DistDGLv2 focuses on optimizations that are agnostic to models and training algorithms to provide robust model training. It offers very flexible mini-batch sampling pipeline to adopt more advanced sampling algorithms.

3 BACKGROUND

3.1 Graph neural networks

GNNs emerge as a family of neural networks capable of learning a joint representation from both the graph structure and vertex/edge features. Recent studies [2, 6] formulate GNN models with *message passing*, in which vertices broadcast messages to their neighbors and compute their own representation by aggregating messages.

More formally, given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, we denote the input feature of vertex v as $\mathbf{h}_v^{(0)}$, and the feature of the edge between vertex u and v as $\mathbf{e}_{uv}$. To get the representation of a vertex at layer l, a GNN model performs the computations below:

$$\mathbf{h}_v^{(l+1)} = g(\mathbf{h}_v^{(l)}, \bigoplus_{u \in \mathcal{N}(v)} f(\mathbf{h}_u^{(l)}, \mathbf{h}_v^{(l)}, \mathbf{e}_{uv})) \quad (1)$$

f, $\bigoplus$ and g are customizable or parameterized functions for generating messages, aggregating messages and updating vertex representations, respectively. Similar to convolutional neural networks (CNNs), a GNN model iteratively applies Equations (1) to generate representations with multiple layers.

3.2 Mini-batch training

Even though GNN models can be trained in full-batch fashion, mini-batch training is more practical for GNN models on large graphs. It has been established that training neural networks with SGD using small mini-batches converges faster and to a lower minimal than passing the whole dataset through the network [14, 24]. Figure 2 shows the time of full-graph and mini-batch training to converge on graphs of medium scale and large scale (Table 1) on the same CPU machine. On these graphs, full-batch training of GraphSage is one or two orders of magnitude slower than mini-batch training. In addition, full-graph training cannot converge to the same accuracy as mini-batch training on some graphs. For example, full-graph training on the Amazon dataset has the test accuracy of 0.68 while mini-batch training gets test accuracy of 0.77.

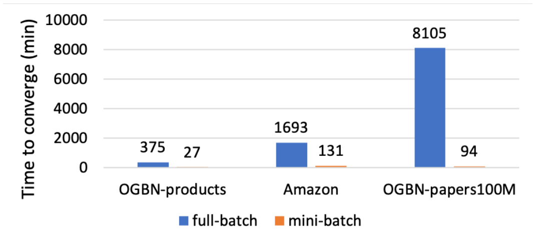


Figure 2: Train GraphSage with full-graph and mini-batch training on medium-size and large graphs on the same CPU machine using DGL.

GNN mini-batch training is different from other neural networks due to the data dependency between vertices. We need to carefully sample subgraphs to capture the dependencies in the original graph. A typical strategy of mini-batch sampling for GNN [7] follows three steps: (i) sample a set of N vertices, called *target vertices*, uniformly at random from the training set; (ii) randomly pick at most K (called *fanout*) neighbor vertices for each target vertex; and (iii) reduce the NK neighbors to a unique set. When the GNN has multiple layers, neighbor sampling repeats recursively. That is, from a sampled neighbor vertex, it continues sampling its neighbors. The number of recursions is determined by the number of layers in a GNN model. This sampling algorithm results in mini-batches with 100s times more vertices than the number of target vertices and causes large amount of communication in distributed training.

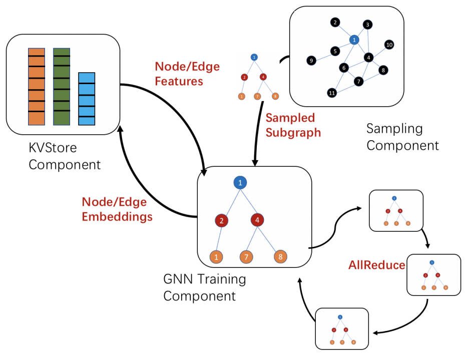


Figure 3: Distributed training components in DistDGL.

3.3 Distributed training

DGL/DistDGL [26] provides the distributed training capability on homogeneous graphs. It uses the existing programming interface of mini-batch training in DGL and divides distributed training into three components (Figure 3):

- A *mini-batch sampler* samples mini-batches from the input graph. Users invoke DistDGL samplers in the trainer process. Internally, the sampling requests are handled by multiple sampling processes, which generates remote process calls (RPC) to perform distributed sampling.
- A *KVStore* that stores all vertex features and edge features across machines. It provides the *pull* and *push* interfaces for pulling data from or pushing data to the distributed store.
- A *trainer* fetches mini-batch graphs from the sampler and corresponding vertex/edge features from the KVStore and runs the forward and backward computation to compute the gradients of the model parameters.

When DistDGL deploys these logical components to actual hardware, it is mainly optimized for distributed training in CPU, in which the main optimization is to reduce the network traffic among machines. DistDGL partitions the input graph with METIS algorithm [12] and partitions the vertex/edge features according to graph partitions. DistDGL launches sampler servers, KVStore servers and trainers on the same cluster of machines and dispatches computation to the data owner to reduce network communication.

4 SYSTEM DESIGN

DistDGLv2 preserves the programming interface of DGL/DistDGL and extends DistDGL in two major ways. First, It implements distributed heterogeneous graphs with guarantees in load balancing and data locality and expose DGL's heterogeneous graph interface for ease of use. It optimizes distributed hybrid CPU/GPU training, where graph data are in distributed CPU memory and mini-batch computation in GPU, with API compatible to DGL's mini-batch training. As such, DistDGLv2 enables distributed training with almost no code modification to DGL's training scripts. To construct an efficient system for distributed hybrid CPU/GPU training, we optimize the system in three aspects.

Data locality: Reducing data movement from slower memory (e.g., remote memory) to GPUs is essential to the training speed. To reduce data movement from the distributed CPU memory to GPUs, DistDGLv2 partitions a heterogeneous graph with the METIS algorithm and co-locates trainers with graph partitions (Section 4.1.1). To minimize data copy in CPU inside a trainer, it only uses multi-threading for parallelization in the mini-batch sampling pipeline (Section 4.2.1). To reduce data copy to GPUs in a mini-batch, DistDGLv2 deploys two-level graph partitioning to reduce the number of vertices in a mini-batch (Section 4.1.2).

Load balancing: Due to neighbor sampling, GNN mini-batches may vary significantly in the number of vertices and edges. In a heterogeneous graph, vertices of different types may have different features, which makes data access more imbalanced if vertex features are not evenly distributed among machines. DistDGLv2 balances the distributed training workloads in two levels. In the data preprocessing, it ensures roughly the same number of vertices and edges of different types in each partition (Section 4.1.3). During training, it removes global synchronization barrier in mini-batch generation to hide the impact of any imbalance in mini-batch sampling from the training process (Section 4.2.1).

Use all hardware resources simultaneously: Distributed hybrid CPU/GPU training involves in different hardware resources: CPU, GPU, network, PCIe, etc. Different hardware has different computation speeds or data transfer speeds. To use all hardware resources effectively, DistDGLv2 adopts two separate strategies: 1) split mini-batch generation into many stages in a pipeline and turn all computations into asynchronous operations (Section 4.2.1) to overlap computation and communication, 2) move more computation to GPUs to reduce the burden in CPU (Section 4.2.2).

4.1 Distributing heterogeneous graphs

DistDGLv2 is designed to support heterogeneous graphs with diverse vertex and edge features while providing the same user-friendly heterogeneous graph API of DGL. Figure 4 (b) shows a heterogeneous graph whose schema is shown in Figure 4 (a). After partitioning a heterogeneous graph and storing data in a cluster of machines (Figure 4 (d)), DistDGLv2's API allows users to access data in the distributed graph as if accessing a graph in a single machine. When partitioning a heterogeneous graph, we ensure minimal edge cut and balanced partitions.

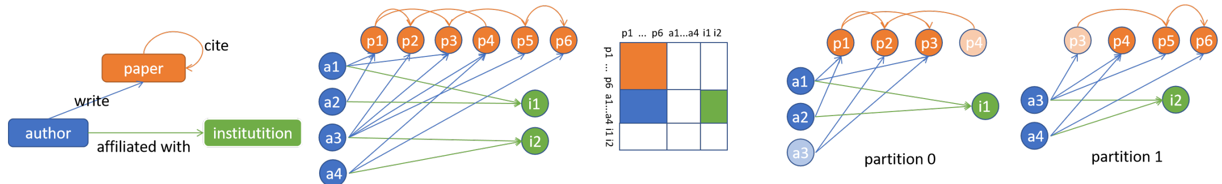
4.1.1 Partition heterogeneous graphs. To reduce data communication in distributed training, DistDGLv2 deploys METIS [12] to partition a heterogeneous graph with a minimal number of edge cuts across partitions. Because graph partitioning is a preprocessing step, the partitioning overhead can be amortized. We usually partition a graph once and use it for many training runs (e.g., during hyperparameter tuning).

Because METIS can only partition a homogeneous graph, DistDGLv2 homogenizes a heterogeneous graph and stores the entire graph in a single adjacency matrix, in which all vertices, regardless of their vertex types, are assigned with unique vertex IDs (Figure 4 (c)). The edges are located in the colored blocks in the adjacency matrix. In this format, the vertices and edges of the same type are assigned with contiguous IDs and vertex types and edge types are stored as metadata. We pass the adjacency matrix to METIS for partitioning, which results in partitions in Figure 4 (d). After assigning vertices to a partition, DistDGLv2 follows the same strategy in DistDGL to assign edges to partitions and split a graph into physical subgraphs. This results in partitions shown in Figure 4 (d).

4.1.2 Hierarchical partitioning. DistDGLv2 deploys two-level partitioning to reduce data transfer to GPUs. In the first level, we deploy METIS to split a graph into physical subgraphs and assign one first-level partition to a machine. Due to the min-edge cut by METIS, the first-level partitions reduce data communication across the network. Inside each partition, we run METIS again to generate second-level partitions and assign one second-level partition to a GPU. Instead of generating physical subgraphs for the second level, we simply assign vertices to second-level partitions and split the training set accordingly. As such, a trainer samples target vertices or edges from the local second-level partition. This increases locality in neighbor sampling. That is, two vertices is more likely to sample the same neighbor vertex, which reduces the number of vertices in a mini-batch. Our ablation study (Section 5.5) shows that introducing the second-level partitions can effectively reduce the number of vertices in a mini-batch and improves the training speed by roughly 20% on the benchmark datasets.
Because DistDGLv2 uses synchronous SGD to train the model, the estimation of the model gradients is unbiased. As such, distributed training in DistDGLv2 in theory does not affect the convergence rate or the model accuracy.

4.1.3 Load balancing on graph partitions. Minimizing edge cut reduces data communication in distributed training, but may result in imbalanced partitions and imbalanced data storage in the cluster. In a heterogeneous graph, different vertex types and edge types may be associated with different data sizes. It is essential to distribute graph partitions, vertex data and edge data of different types evenly across all machines so that CPU memory storage in each machine is fully utilized and data access during the training can be evenly distributed among machines. By default, METIS only balances the number of vertices in a graph. This is insufficient for a heterogeneous graph. We formulate this load balancing problem as a multi-constraint partitioning problem [13]. DistDGLv2 takes advantage of the multi-constraint mechanism in METIS to balance training/validation/test vertices/edges in each partition as well as balancing the vertices of different types and the edges incident to the vertices.

4.1.4 Heterogeneous graph vertex/edge data. To support flexible storage of diverse features on different vertex types and edge types, DistDGLv2 extends the distributed KVStore to store features on each vertex type and edge type separately. The extended KVStore supports an arbitrary number of ID spaces. When DistDGLv2 loads a distributed heterogeneous graph, it creates an ID space in KVStore for each vertex type and edge type. Each ID space is also associated with a partition policy that maps vertex/edge data to physical machines. The partition policy is derived from the first-level graph partitioning (Section 4.1.1). To simplify the access to vertex/edge data in the KVStore, DistDGLv2 uses type-specific vertex/edge ID, which is only unique within a particular vertex/edge type.



(a) The schema of an academic graph. (b) A small academic graph. (c) The adjacency matrix of the homogenized academic graph. (d) Partition the academic graph into two parts.

Figure 4: An example of a heterogeneous graph and its distributed storage.

4.2 Asynchronous mini-batch generation

The key of efficient hybrid CPU/GPU training is to bring mini-batch data to GPU efficiently. This requires optimizations in multiple aspects. First, we need to overlap mini-batch generation with mini-batch computation as well as overlapping computation and communication to simultaneously utilize all computation resources (e.g., CPU and GPU) and communication channels (e.g., network, CPU memory and PCIe). We need to parallelize computation in CPU and avoid any unnecessary data copy in CPU. In addition, we need to hide the impact of imbalance of GNN mini-batch sampling among different trainers.

4.2.1 Asynchronous mini-batch pipeline. DistDGLv2 deploys an asynchronous pipeline that generates mini-batches from the distributed graph. It provides the sampling API compatible with DGL and delivers mini-batches from a distributed graph as if sampled from a graph in a single machine. It allows customization of sampling algorithm in Python while deploying heavy optimizations to speed up computation. DistDGLv2 divides mini-batch training into many stages (Figure 5 (a)):

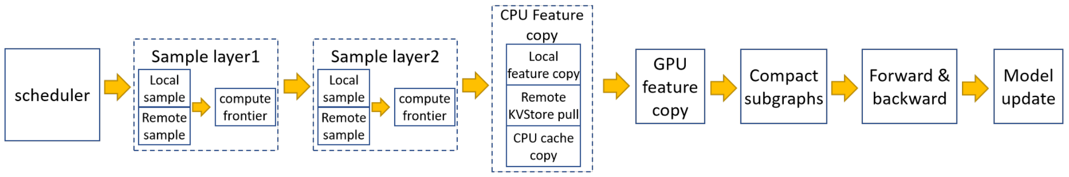
- a scheduler that determines target vertices or target edges in each mini-batch to support various learning tasks (e.g., node classification, link prediction) for GNN models,
- neighbor sampling that samples multi-hop neighbors of the target vertices for GNN computation,
- CPU feature copy that fetches data from both local machines and remote machines for each mini-batch and stores data in contiguous CPU memory,
- GPU feature copy that loads data from CPU to GPU,
- post-sampling GPU processing for mini-batches (in vertex-wise neighbor sampling, we performs subgraph compaction that remaps vertex IDs and edge IDs in the subgraph in GPU),
- forward and backward computation on mini-batches,
- model parameter updates.

There are dependencies between operations in different stages, but in some stages there are multiple operations that can run in parallel. For example, the stage of *CPU feature copy* requires the frontier of the input layer to be complete in the sampling stage; on the other hand, copying features in CPU includes data copy from the local partition, from remote KVStore and from local CPU cache, which can run independently. A neighbor sampling stage can be further divided into two substages: sample neighbors and compute the frontier. The two substages also have dependencies: we have to wait for neighbor sampling to complete before computing the frontier of the layer.

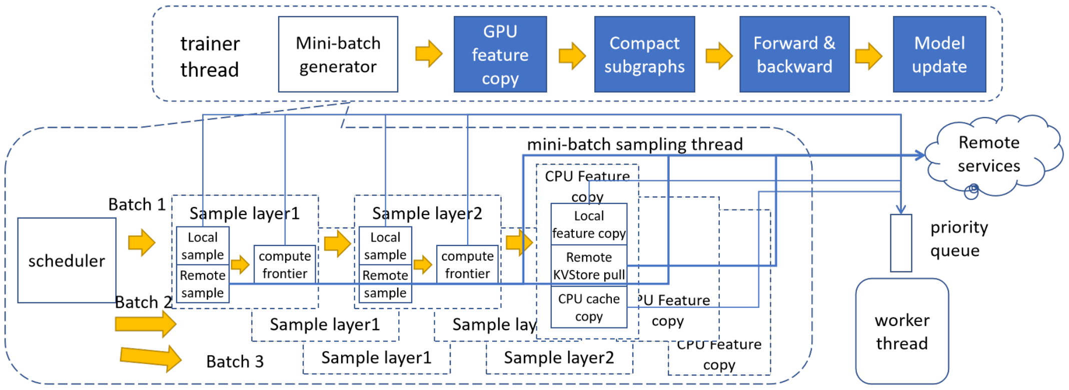
DistDGLv2 implements a flexible and efficient asynchronous mini-batch pipeline to overlap computation with network communication (Figure 5 (b)). Because the target vertices or edges are sampled from the training set randomly, there are no dependencies between mini-batches. This allows us to sample mini-batches ahead of time and process multiple mini-batches in a pipelining fashion. DistDGLv2 divides the mini-batch pipeline into two parts: 1) mini-batch sampling in CPU, which includes mini-batch scheduling, distributed neighbor sampling and CPU feature copy, and 2) post-sampling computation in GPU, which includes data loading to GPUs, compacting subgraphs, and mini-batch computation in GPU. To avoid sampling computation from blocking mini-batch training in GPU, DistDGLv2 creates a dedicated Python thread for the sampling computation in CPU, which allows us to run customized Python code for sampling. We refer to this thread as a *sampling thread* and the original thread as a *training thread*. The CPU and GPU division reduces the interference between the two threads: The GPU computation in the training thread has a global synchronization barrier among trainers due to synchronized SGD but is not blocked by any computation in the sampling thread; the sampling thread is not blocked by the global synchronization barrier caused by SGD. Even though Python threads have a global lock to guard the data access to Python objects, the lock is released whenever we jump to C code. Thus, the Python global lock does not interfere the computation of the two threads by much.

Inside the sampling thread, DistDGLv2 performs all computation asynchronously to ensure that network operations are not blocked by local CPU computations. Whenever DistDGLv2 performs a local operation in CPU, it creates a job for this operation, sends it to a worker thread and returns immediately. The job is placed in a priority queue of the worker thread. A job created for a later stage gets a higher priority. Whenever DistDGLv2 performs a remote operation, it issues an RPC request and returns immediately. The sampling thread processes multiple mini-batches simultaneously. After it processes the operations of a mini-batch at a certain stage, instead of waiting for the operations to be complete, it proceeds to some operations of another mini-batch at a different stage. After issuing a sufficient number of pending operations, the thread sleeps and waits for some operations to be complete.

This aggressive ahead-of-time mini-batch generation can potentially lead to data staleness and consume much memory. Currently, we only apply this ahead-of-time mini-batch generation on immutable data (i.e., sampling from the graph structure and read vertex/edge features). If a model has learnable embedding table on vertices, we read the learnable embeddings synchronously. Thus, our asynchronous mini-batch pipeline does not affect model convergence at all. To reduce memory consumption, the pipeline sets



(a) The stages of the mini-batch pipeline. The arrows indicate the dependencies of computation.



(b) Realize the asynchronized mini-batch pipeline with three threads.

Figure 5: DistDGLv2 deploys an asynchronous mini-batch pipeline for hybrid CPU/GPU training. The pipeline are divided into multiple stages. Some of the stages run in GPUs, indicated by the blue boxes, while others run in CPU, indicated by the white boxes. The computations in the pipeline run in three threads. All GPU computations are invoked in the trainer thread; sampling computation and CPU feature copy are invoked in the sampling thread but their actual computation happens in the worker thread. In the sampling thread, computations in multiple mini-batches are invoked simultaneously in a pipelining fashion to overlap computation of different stages.

different capacities for different stages. The capacity is defined with the number of pending operations issued in a stage. The memory consumption by the operations at different stages is different. At the beginning of the pipeline (mini-batch scheduling and neighbor sampling), a pending operation only needs to store vertex IDs and edge IDs, which does not require too much memory, so we can work on many mini-batches simultaneously. In the middle of the pipeline, we prefetch vertex/edge features from remote machines and collect features from local partitions, which may require hundreds of megabytes of CPU memory, so we allows a relatively small number of mini-batches. At the end of the pipeline, we only move one mini-batch ahead of time to GPUs because of the scarceness of GPU memory. As such, we use a relatively large capacity for scheduling and neighbor sampling (e.g., 25); a relatively small capacity (e.g., 5) for CPU feature copy; a capacity of 1 for GPU feature copy.

The main reason of using multithreading for parallelizing sampling in DistDGLv2 is to minimize data copy in the pipeline. This is different from many other distributed GNN training frameworks, such as DistDGL [26] and Euler [1], which uses multiprocessing for parallelization. Even though multiprocessing parallelizes sampling computation well, it requires to copy mini-batch data between processes, which results in additional data copy and data serialization and deserialization. In contrast, multithreading completely avoids these overheads. To further reduce data copy in the pipeline, DistDGLv2 carefully manage data buffers for network communication and data copy between CPUs and GPUs. It allocates a pinned memory buffer to collect data from the network and from the local partition before sending them to GPUs, which results in only one data copy for each byte.

The asynchronous sampling pipeline introduces a startup overhead when filling the pipeline at the beginning of every epoch. This hurts the performance especially when the training set is small. To remove the startup overhead, we run the asynchronous sampling pipeline throughout the entire training without stopping in the sampling thread. The trainer thread only needs to fetch mini-batches from the sampling thread.

4.2.2 *Distributed hybrid CPU/GPU sampling.*

In hybrid CPU/GPU training, the distributed graph is placed in CPU memory. We have to sample neighbors of target vertices on CPU from the distributed graph. To take advantage of GPU's computation power, we move some computation to GPUs. As such, DistDGLv2 divides the computation into multiple components. In this section, we use the vertex-wise neighbor sampling algorithm [7] for vertex classification for illustration. The same computation decomposition applies to other neighbor sampling algorithms, such as layer-wise sampling [28], and to other training tasks, such as link prediction.

Figure 5 shows the mini-batch sampling pipeline of vertex-wise neighbor sampling [7] for vertex classification. It starts from the seed vertices and samples their neighbor vertices in the ego-network. The sampling computation proceeds with one hop of neighborhood at a time. After sampling all neighbors within a hop, it computes

Table 1: Dataset statistics.

Dataset	# Vertices	# Edges	Vertex features	# train vertices	# train links
OGBN-PRODUCT[9]	2.4M	61.9M	100	197K	61.9M
AMAZON [3]	1.6M	264M	200	1.3M	264M
OGBN-PAPERS100M[9]	111M	3.2B	128	1.2M	3.2B
OGBN-MAG[9]	1.9M	21M	128	629K	21M
MAG-LSC[8]	240M	7B	756	1.1M	7B

the frontier (i.e., the unique set of vertices) as the seed vertices for the next-hop neighbor sampling.

DistDGLv2 divides neighbor sampling into two components. In CPU, it samples vertices and edges from the distributed graph for each hop of neighborhood. The sampled subgraphs are small enough to fit in GPU memory. It then moves the subgraphs to GPU and performs graph compaction to remove empty vertices and relabel vertices and edges for mini-batch computation. This algorithm samples neighbors on each vertex independently and, thus, we can further decompose the sampling computation within a hop into local sampling and remote sampling. DistDGLv2 dispatches the remote sampling requests to the sampler servers and issues a job for local sampling simultaneously. After local and remote sampling complete, DistDGLv2 collects the results from different partitions, stitches them together and issues another job to compute the frontier vertices for the next hop. All sampling computation within a hop runs in CPU. To better parallelize the frontier computation, DistDGLv2 bundles the sampling computation of multiple mini-batches and use OpenMP to parallelize them.

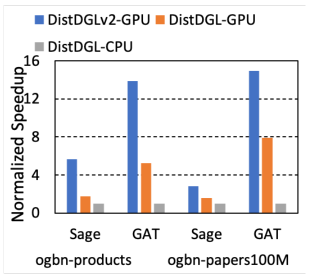
5 EVALUATION

In this section, we evaluate DistDGLv2 with multiple GNN models on large graph datasets. We benchmark three commonly used GNN models (GraphSAGE [7], Graph Attention Networks (GAT) [22] and Relational Graph Convolution Networks (RGCN) [19]) to evaluate the performance of DistDGLv2.

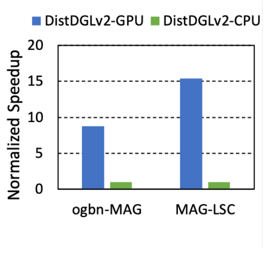
Our benchmarks use three medium-size graphs (OGBN-PRODUCT [9], AMAZON [3] and OGBN-MAG) and two large graphs (OGBN-PAPERS100M [9] and MAG-LSC [8]) (see Table 1 for various statistics). Note that even though all datasets contain labels for vertex classification, the number of labeled vertices in all but the smaller datasets is similar. As a result, the cost to train vertex classification models for the larger graphs is not as high as the size of the graphs suggests. However this is not the case for the link-prediction task, for which we use all the edges to train the GNN models, leading to training sets with billions of data points.

5.1 DistDGLv2 vs. other frameworks

We compare the training speed of DistDGLv2 with DistDGL [26] and Euler [1], the state-of-the-art distributed GNN mini-batch training frameworks, on four g4dn.metal instances, each of which is equipped with eight NVIDIA T4 GPUs and two Intel Xeon Platinum 8259CL CPUs, for a total of 32 GPUs and 192 CPU cores. Both DistDGL and Euler are designed for distributed CPU training, so we run them on four r5dn.24xlarge instances to collect their CPU training speed, each of which have two Intel Xeon Platinum 8259CL CPUs, for a total of 192 CPU cores across the four instances.



(a) GraphSage and GAT on homogeneous graphs



(b) RGCN on heterogeneous graphs

Figure 6: Training speed of DistDGLv2 vs. DistDGL.

To have a fair comparison with DistDGLv2, we change DistDGL and Euler to perform GNN training on GPUs by moving sampled mini-batches to GPUs. We refer to their CPU versions as DistDGL-CPU and Euler-CPU and their GPU versions as DistDGL-GPU and Euler-GPU. We run all experiments with the same global batch size (the total size of the batches of all trainers in an iteration) to get the same convergence.

Figure 6 (a) shows that DistDGLv2 gets $2-3\times$ speedup over DistDGL-GPU on various datasets. DistDGLv2 has higher speedup over DistDGL-GPU on simpler GNN models (e.g., GraphSage). The main bottleneck of GraphSage training is mini-batch sampling in CPU and data copy to GPUs. Even though both DistDGLv2 and DistDGL use METIS to partiton a graph and co-locate data with computation, this alone cannot fully take advantage of GPU's computation. Asynchronous mini-batch generation, parallelization strategies and load balancing deployed in DistDGLv2 further improve the performance of GNN training.

To verify the benefit of distributed GNN training with GPUs, we compare DistDGLv2-GPU with DistDGL-CPU on GraphSage and GAT and DistDGLv2-CPU on RGCN. Figure 6 shows DistDGLv2-GPU has up to $15\times$ speedup over DistDGL-CPU and DistDGLv2-CPU, which indicates that high floating-point computation and fast memory in GPU are beneficial to train GNN models on large graphs especially for more complex GNN models, such as GAT and RGCN. Even for GraphSage, using GPUs still gets $3-6\times$ speedup.

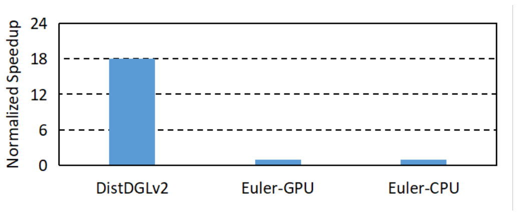
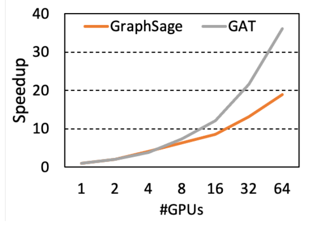
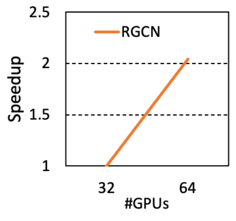


Figure 7: The speedup of DistDGLv2 and Euler-GPU over Euler-CPU for training GraphSage on OGBN-PRODUCT.

We further compare DistDGLv2 with Euler on CPUs and GPUs when training GraphSage on OGBN-PRODUCT (Figure 7). DistDGLv2 gets $18\times$ speedup over both Euler-CPU and Euler-GPU. Euler-GPU does not get speedup over Euler-CPU. Because Euler only uses multiprocessing to parallelize computation and run sampling inside the trainer process, it requires many trainer processes to achieve



(a) OGBN-PAPERS100M



(b) MAG-LSC

Figure 8: The speedup of DistDGLv2 with GPUs on large graphs.

good performance. This parallelization strategy works relatively well on CPU clusters but does not work well on GPUs because we usually launch one trainer process per GPU to save GPU memory and avoid interfering computation between trainer processes on the same GPU. This indicates that effective distributed GNN training on GPUs requires both multiprocessing and multithreading.

5.2 Scalability

We evaluate the scalability of DistDGLv2 in the EC2 cluster. In this experiment, we fix the mini-batch size in each trainer and increase the number of trainers when the number of GPUs increases.

Figure 8 shows that DistDGLv2 achieves 20× speedup in GraphSage and 36× speedup in GAT with 64 GPUs on OGBN-PAPERS100M. MAG-LSC is too large to fit in the CPU memory of one or two g4dn.metal instances. Its training speed doubles when scaling from four instances to eight instances. The sub-linear speedup of DistDGLv2 in GraphSage is due to CPU saturation caused by mini-batch generation and network saturation caused by data copy from remote machines. When a GNN model (e.g., GAT) has more computation, DistDGLv2 gets better speedup. In a cluster of 64 GPUs, one epoch takes only 5 seconds for GraphSage and 7 seconds for GAT on the OGBN-PAPERS100M graph and takes 13 seconds for RGCN on MAG-LSC in a cluster of 64 GPUs.

5.3 Training convergence

Each trainer of DistDGLv2 samples data points from its graph partition, but collectively, the data points in a global mini-batch are sampled uniformly at random from the entire training set. This training method is a little similar to ClusterGCN [3], which partitions a graph with METIS and sample partitions to form mini-batches. We compare DistDGLv2 with ClusterGCN on OGBN-papers100M. We partition the graph into 32 partitions for DistDGLv2 and 16,384 partitions for ClusterGCN.

Figure 9 shows that ClusterGCN has slower convergence than DistDGLv2 and it cannot converge to the same accuracy as DistDGLv2. The main difference between ClusterGCN and DistDGLv2 is that ClusterGCN drops the edges that do not belong to the partitions in a mini-batch, while DistDGLv2 always samples neighbors uniformly at random. Thus, DistDGLv2 estimates neighbor aggregation in an unbiased fashion, while ClusterGCN's estimation is based by graph partitioning results. This indicates that we have to sample neighbors across partitions to achieve good model accuracy.

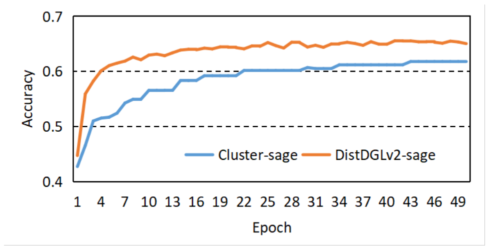


Figure 9: Convergence of DistDGLv2 vs. ClusterGCN on OGBN-PAPERS100M.

Table 2: Time breakdown of distributed training for different tasks on OGBN-PAPERS100M.

Task	ParMETIS	Load/save (partition)	Load data (training)	Train to converge
Vertex classification	12 min	23 min	8 min	4 min
Link prediction	12 min	23 min	8 min	305 min

5.4 Time breakdown

In DistDGLv2, training a GNN model requires to partition a graph and run a distributed training job on the partitions. We measure the time of different components in the training pipeline, including loading and saving data for partitioning, partitioning the graph, loading partition data for training and finally training a model to converge. We use ParMETIS [11] to partition large graphs. We benchmark ParMETIS on OGBN-PAPERS100M on a cluster of four r5dn.24xlarge instances and distributed training jobs on a cluster of g4dn.metal instances.

Table 2 shows the time breakdown in the training pipeline. It assumes that graph partition occurs for every distributed training job. In practice, we partition a graph for multiple training jobs (e.g., parameter searching and testing different models). Even in this setting, graph partitioning is not the most time-consuming component in the training pipeline. It takes only 12 minutes to partition OGBN-PAPERS100M into 512 partitions. In comparison, data loading and saving takes much more time. For vertex classification, the training time is short because OGBN-PAPERS100M has a very small training set (1% of vertices are in the training set). It is likely to get a large dataset with more labeled vertices. For link prediction, we may use all edges to train a model, which leads to a training set with billions of data points. Training a model for link prediction takes multiple hours even with one epoch.

5.5 Ablation Study

DistDGLv2 deploys many optimizations. In this section, we study the effectiveness of the main optimizations introduced by DistDGLv2, excluding the ones introduced in DistDGL: 1) 2-level partition that splits the graph for the levels of machines and GPUs; 2) asynchronous pipeline that performs every operation in mini-batch generation asynchronously to overlap CPU and GPU computation and network I/O; 3) hybrid CPU/GPU sampling that moves some mini-batch sampling computation to GPUs. We study these optimizations by adding one optimization after another until we add all optimizations. The last one basically includes all optimizations in the study. The study uses the optimization of METIS partitioning

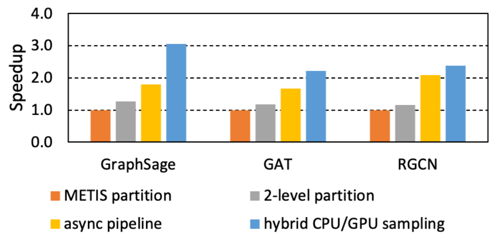


Figure 10: The effectiveness of the techniques for GraphSage and GAT on OGBN-PRODUCT and for RGCN on OGBN-MAG.

as the baseline because the benefit of METIS partitioning has been demonstrated by DistDGL [26]. We use a cluster of four g4dn.metal instances to run the experiments.

Figure 10 shows each optimization has impact in performance and we get overall 3× speedup for GraphSage and over 2× speedup for GAT and RGCN on top of METIS partitioning. Even though this cluster already has 100Gbps network, 2-level partitioning gets about 20% speedup because confining the training vertices in a smaller partition leads to better locality and a smaller number of neighbor vertices in a mini-batch. Asynchronous sampling pipeline gets significant boost because it overlaps the CPU and GPU computation and network I/O to hide network latency, PCIe data transfer and CPU data copy. Hybrid CPU/GPU sampling is another effective optimization. This indicates that moving more computation to GPU is beneficial to speed up training.

6 CONCLUSION

We develop DistDGLv2 for distributed GNN training in a GPU cluster. The hybrid CPU/GPU training allows to scale to very large graphs. We show that distributed hybrid CPU/GPU training can get speedup by a factor of 3 − 15 over distributed CPU training on a graph with hundreds of millions of vertices. DistDGLv2 adopts many optimizations to make GNN training more efficient in a cluster of GPUs. We show that only using METIS partitioning is insufficient to achieve good training speed for distributed hybrid CPU/GPU training. By deploying an asynchronous pipeline for generating mini-batches, we can effectively hide the latency of data communication and overlap CPU and GPU computation. Because asynchronous mini-batch sampling only applies to immutable data, it does not affect model convergence. By having all optimizations, DistDGLv2 gets 2 − 3× speedup over DistDGL and 18× speedup over Euler on GPUs. Currently, the heterogeneous graph support in DistDGLv2 has been released as part of DGL 0.7 and have been used in production. We plan to release the asynchronous mini-batch sampling pipeline in DGL's following release.

REFERENCES

[1] Euler github. https://github.com/alibaba/euler, 2020.
[2] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.
[3] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Jul 2019.
[4] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.
[5] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 551–568. USENIX Association, July 2021.
[6] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, 2017.
[7] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1025–1035, 2017.
[8] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. Ogb-lsc: A large-scale challenge for machine learning on graphs. *arXiv preprint arXiv:2103.09430*, 2021.
[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] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 187–198. 2020.
[11] G. Karypis and Kirk Schloegel. Parmetis 4.0: Parallel graph partitioning and sparse matrix ordering library. Technical report, Department of Computer Science, University of Minnesota, 2011. *http://www.cs.umn.edu/˜metis*.
[12] 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.
[13] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, page 1–13, USA, 1998.
[14] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.
[15] 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*, pages 401–415, 2020.
[16] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing. *arXiv preprint arXiv:2112.08541*, 2021.
[17] 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, Renton, WA, July 2019.
[18] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj D. Kalamkar, Nesreen K. Ahmed, and Sasikanth Avancha. Distgnn: Scalable distributed training for large-scale graph neural networks. *CoRR*, abs/2104.06700, 2021.
[19] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. *CoRR*, abs/1703.06103, 2017.
[20] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 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. USENIX Association, July 2021.
[21] Alok Tripathy, Katherine Yelick, and Aydin Buluc. Reducing communication in graph neural network training. *arXiv preprint arXiv:2005.03300*, 2020.
[22] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. *CoRR*, abs/1710.10903, 2018.
[23] 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. *arXiv preprint arXiv:1909.01315*, 2019.
[24] D Randall Wilson and Tony R Martinez. The general inefficiency of batch training for gradient descent learning. *Neural networks*, 16(10):1429–1451, 2003.
[25] Dalong Zhang, Xin Huang, Ziqi Liu, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Zhiqiang Zhang, Lin Wang, Jun Zhou, Yang Shuang, and Yuan Qi. AGL: a scalable system for industrial-purpose graph machine learning. *arXiv preprint arXiv:2003.02454*, 2020.
[26] 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*, 2021.
[27] 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.
[28] 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.

A APPENDIX

A.1 Hyperparameters and software

We perform hyperparameter search and select a set of hyperparameters that achieve good model accuracy on these datasets. For GraphSage and GAT in vertex classification, we use three GNN layers and the hidden size of 256; the fanout of each layer is 15, 10 and 5. GAT uses 2 attention heads. RGCN uses two layers with the hidden size of 1024 and the sampling fanout is 15 and 25. We use the batch size of 1000 per trainer for GraphSage and 500 for GAT and RGCN [1]. For link prediction, we run two GraphSage layers to generate embeddings and the sampling fanout is 25 and 15; the remaining configurations are the same. We use a cluster of eight AWS EC2 g4dn.metal instances (96 vCPU, 384GB RAM, 8 T4 GPUs each) for GPU experiments and a cluster of four AWS EC2 r5d.24xlarge instances (96 vCPU, 768GB RAM) for CPU experiments and data preprocessing. Both clusters connect machines with 100Gbps network.

In all experiments, we use DistDGL in DGL 0.6[2] and Pytorch 1.8. DistDGLv2 is implemented based on DGL 0.6. For Euler experiments, we use Euler v2.0 and TensorFlow 1.12.

[1]GAT and RGCN run out of memory with the batch size of 1000.
[2]Some of the features in DistDGLv2 have been implemented in DGL0.7 and newer releases