

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

Da Zheng
AWS AI

Xiang Song
AWS AI

Chengru Yang
AWS AI

Qidong Su
AWS AI

Minjie Wang
AWS AI

Chao Ma
AWS AI

George Karypis
AWS AI

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, containing hundreds of millions or billions of nodes. To tackle this challenge, we develop DistDGLv2, a system that extends DistDGL for training GNNs in a mini-batch fashion, using distributed *hybrid CPU/GPU* training to scale to large graphs. DistDGLv2 places graph data in distributed CPU memory and performs mini-batch computation in GPUs. DistDGLv2 distributes the graph and its associated data (initial features) across the machines and uses this distribution to derive a computational decomposition by following an owner-compute rule. DistDGLv2 follows a synchronous training approach and allows ego-networks forming mini-batches to include non-local nodes. To minimize the overheads associated with distributed computations, DistDGLv2 uses a multi-level graph partitioning algorithm with min-edge cut along with multiple balancing constraints. This localizes computation in both machine level and GPU level and statically balance the computations. DistDGLv2 deploys an asynchronous mini-batch generation pipeline that makes all computation and data access asynchronous to fully utilize all hardware (CPU, GPU, network, PCIe). The combination allows DistDGLv2 to train high-quality models while achieving high parallel efficiency and memory scalability. 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 100s millions of nodes on a cluster with 64 GPUs.

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, in the order of many millions of nodes or even billions of nodes. For instance, Facebook’s social network graph contains billions of nodes. Amazon is selling billions of items and has billions of users, which forms a giant bipartite graph for its recommendation task. Natural language processing tasks take advantage of knowledge graphs, such as Freebase [12] with 1.9 billion triples.

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). Distributed frameworks that perform full-graph training have been developed for both CPU- and hybrid CPU/GPU-based clusters [17, 24, 26, 28, 29], whereas distributed frameworks that perform mini-batch training have been developed/optimized only for CPU-based clusters [1, 32–34]. 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. The main challenges of GNN training on GPUs lie in two aspects. First, GNN models have much lower computation density than traditional neural network models, such as CNNs and Transformers. 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 ac-

curacy, 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 nodes 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.

In this work, we develop DistDGLv2 on top of DGL [31] to optimize distributed GNN training for hybrid CPU/GPU clusters. To maintain API compatibility with DGL’s mini-batch training, DistDGLv2 provides a programming interface that requires almost no code modification to DGL’s training scripts. DistDGLv2 adopts data parallelism for better training GNN models with large hidden sizes [9] and parallelize computations with both multithreading and multiprocessing. To scale to large graphs, DistDGLv2 stores the graph structure and node/edge attributes in CPU memory and to take advantage of GPU’s floating-point computation power, performs mini-batch computation in GPUs. The design of DistDGLv2 follows three principles: (i) increase data locality to reduce data copy overheads, (ii) balance the computations to ensure that all computing resources are used effectively, (iii) hide the latency of CPU computation and data communication and balance CPU/GPU computation. To reduce data copy in the cluster, DistDGLv2 deploys a hierarchical graph partitioning algorithm to localize data access in the level of machines and GPUs. DistDGLv2 balances synchronous SGD training in three stages: (i) at the stage of preprocessing, it formulates multi-constraint objectives for graph partitioning to generate balanced partitions; (ii) at the stage of launching distributed training jobs, it runs a distributed algorithm to split the training set to co-locate training data points with graph partitions and balance remote training data evenly across all trainers. (iii) at the stage of mini-batch computation, it uses an asynchronous mini-batch sampling pipeline to sample mini-batches ahead of time to hide the imbalance in mini-batch generation. To hide latency of CPU computation and data communication, DistDGLv2 breaks up the mini-batch sampling into many steps and turns the computation in every step asynchronous to not only hide the latency of network data access and PCIe data transfer, but also the local CPU data copy and computation. In the mini-batch sampling pipeline, DistDGLv2 moves large part of sampling computation to GPUs to balance CPU/GPU computation.

We conduct comprehensive experiments to evaluate the efficiency of DistDGLv2 and effectiveness of the optimizations. Overall, DistDGLv2 achieves $18\times$ speedup for training GraphSage over Euler and $2-3\times$ speedup over DistDGL [33] for various datasets and tasks on a cluster of 32 GPUs when all frameworks uses GPUs for mini-batch computation. Furthermore, DistDGLv2 takes advantage of GPU computation power and achieves up to $27\times$ speedup over distributed CPU training by DistDGL in a cluster of the same size, which indicates that GPUs can be effective for GNN training than CPUs. When scaling to 64 GPUs, DistDGLv2 achieves up

to $36\times$ speedup without compromising model accuracy. It takes about 5 seconds per epoch to train GraphSage and GAT models on a graph with 100 million nodes and 10 seconds per epoch to train RGCN on a graph with 240 million nodes with 64 GPUs.

2 Related Work

2.1 Distributed graph processing

There are many works on distributed graph processing frameworks. Pregel [25] is one of the first frameworks that adopt message passing and vertex-centric interface to perform basic graph analytics algorithms such as breadth-first search and triangle counting. PowerGraph [11] adopts vertex cut for graph partitioning and gather-and-scatter interface for computation. PowerGraph had significant performance improvement over Pregel. Gemini [35] shows that previous distributed graph processing framework has significant overhead in a single machine. It adopts the approach to improve graph computation in a single machine first before optimizing for distributed computation. Even though the computation pattern of distributed mini-batch training of GNN is very different from traditional graph analytics algorithms, the evolution of graph processing frameworks provide valuable lessons for us and many of the general ideas, such as locality-aware graph partitioning and co-locating data and computation, are borrowed to optimize distributed GNN training.

2.2 Distributed GNN Training

Many works have been developed to scale GNN training on large graph data for distributed CPU- and GPU-based clusters. Many of them [17, 24, 26, 28, 29] are designed for distributed full-graph training on multiple GPUs or distributed memory whose aggregated memory fit the graph data. As Figure 2 illustrates, full-graph training not only takes a long time to converge on a large graph, but may also converge to a lower accuracy.

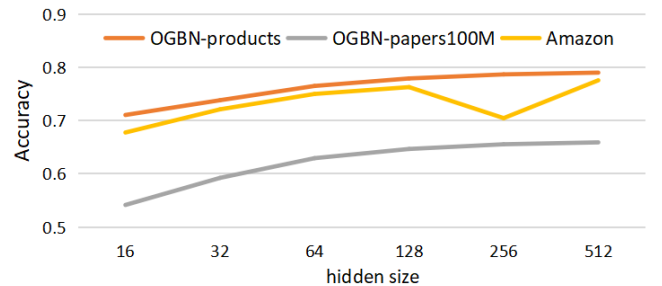


Figure 1: The model accuracy of GraphSage with different hidden sizes on datasets in Section 6.

Some GNN frameworks [1, 32, 34] built by industry adopt

distributed mini-batch training. However, none of these frameworks adopt locality-aware graph partitioning and co-locate data and communication. Their system is optimized for distributed training on a CPU cluster and some of their design choices (e.g., only using multiprocessing) are not suitable for GPU training. DGL [31, 33], support distributed training with locality-aware graph partitioning, but are still designed for distributed training in a CPU cluster. It cannot fetch data across the network to GPUs efficiently. Frameworks from academia, such as PyTorch-Geometric [8] and PaGraph [23], support multi-GPU training but cannot scale to graphs beyond the memory capacity of a single machine. P3 [9] 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. However, a large hidden size is required to achieve good model accuracy (Figure 1).

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, 10] formulate GNN models with *message passing*, in which vertices broadcast messages to their neighbors and compute their own representation by aggregating received 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:

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

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

There are potentially two types of model parameters in graph neural networks. f , \bigoplus and g can contain model parameters, which are shared among all vertices. These model parameters are updated in every mini-batch and we refer to these parameters as *dense* parameters. Some GNN models may additionally learn an *embedding* for each vertex. Embeddings are part of the model parameters and only a subset of vertex embeddings are updated in a mini-batch. We refer to these model parameters as *sparse* parameters.

3.2 Mini-batch training

Even though GNN models can be trained in full-batch fashion, as many GNN frameworks [17, 24, 26, 28] are designed for,

mini-batch training is still a more practical training method for GNN models, especially on large graphs. 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 medium-size graphs, full-batch training of GraphSage is about an order of magnitude slower than mini-batch training while the gap is even larger for large-scale graphs. 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 on the same dataset has test accuracy of 0.77.

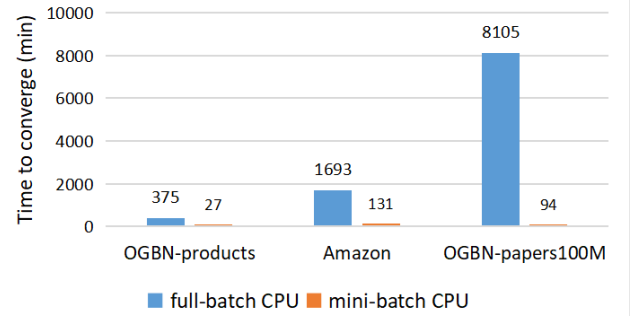


Figure 2: Train GraphSage with full-graph and mini-batch training on medium-size and large graphs on the same CPU machine using Deep Graph Library [31].

GNN mini-batch training is different from other neural networks due to the data dependency between vertices. Therefore, we need to carefully sample subgraphs that capture the data dependencies in the original graph to train GNN models.

A typical strategy of sampling a mini-batch for a GNN model [13] 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; (iii) compute the target vertex representations by gathering messages from the sampled neighbors. When the GNN has multiple layers, the sampling is repeated 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 strategy forms a computation graph for passing messages on. Figure 3 depicts such a graph for computing representation of one target vertex when the GNN has two layers. The sampled graph and together with the extracted features are called a mini-batch in GNN training.

There have been many works regarding to the different strategies to sample graphs for mini-batch training [4–6, 16, 36]. Therefore, a GNN framework needs to be flexible as well as scalable to giant graphs.

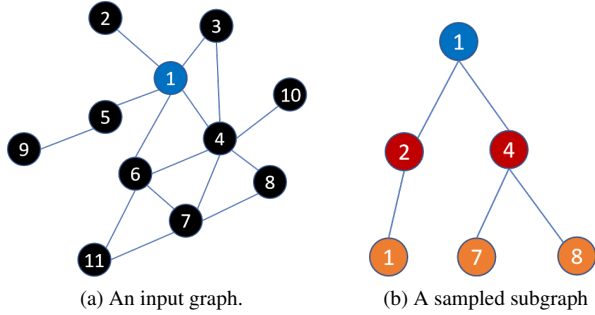


Figure 3: One sampled mini-batch in GNN training. The sampled subgraph for computing one target vertex representation with a two-layer GNN model. Node 1 is the target node of the mini-batch and node 1, 7, 8 are input nodes.

4 Motivation

GPUs have been commonly used for training neural network models because of their massive floating-point computation power and high memory bandwidth, which can be an order of magnitude more than CPUs [3, 7]. If all the data required by a mini-batch is inside of a GPU, training can achieve substantial throughput. Unfortunately, GPUs may have insufficient memory to store large graphs. We have to store majority of data in CPU memory or remote memory and move data to GPUs for mini-batch computation. However, GPU’s performance advantage diminishes once we need to move a large amount of data to GPUs, especially from remote machines.

In this work, we adopt *hybrid CPU/GPU* training to scale to large graphs in a cluster. This approach has been used by state-of-the-art GNN framework (e.g., DGL [31] and Pytorch Geometric [8]) train GNN models on a large graph in a single machine. This approach stores the graph structure as well as node attributes and edge attributes in CPU memory and perform mini-batch computation in GPUs. We sample subgraphs (mini-batches) from the graph structure in CPU, read node attributes and edge attributes involved in the mini-batches from the CPU memory and move them to GPUs for mini-batch computation. This allows users to take advantage of large CPU memory and use GPUs to accelerate GNN training.

However, distributed hybrid CPU/GPU training brings multiple challenges to the training system. First, GPUs have much higher computation power and memory bandwidth than CPUs. This raises two questions: 1) how to effectively move data to GPUs for mini-batch computation from CPU memory and from remote machines; 2) how to divide GNN mini-batch training to balance the workloads in CPUs and GPUs. In addition, it is difficult to generate balanced GNN mini-batches (the same number of nodes and edges in a mini-batch) due to the power-law distribution in vertex degrees and graphs’ internal clustering structure. Due to the synchronization barrier of synchronous SGD, some trainers may finish mini-batch computation earlier than others. Thus, we need to tackle the

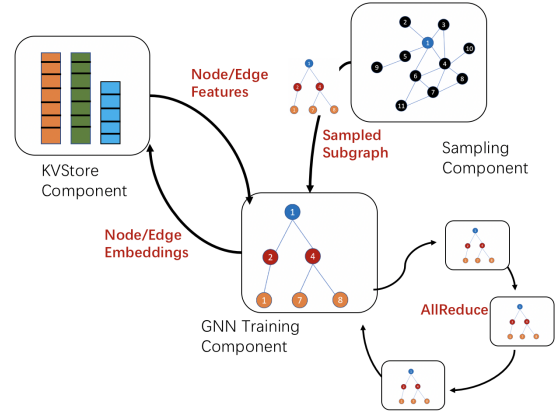


Figure 4: DistDGLv2’s logical components.

load balancing issue in synchronous SGD training to enable efficient distributed GNN training.

5 System Design

5.1 Distributed Training Architecture

DistDGLv2 builds on top of DGL to optimizes distributed GNN training for heterogeneous CPU/GPU clusters. It maintains API compatibility with DGL’s mini-batch training and requires almost no code modification to DGL’s training scripts. Thus, DistDGLv2 adopts data parallelism and use both multiprocessing and multithreading to parallelize computation in GNN training. We design DistDGLv2 to address the challenges in hybrid CPU/GPU training. At a high level, DistDGLv2 consists of the following logical components (Figure 4):

- A *mini-batch sampler* samples mini-batch graph structures from the input graph. It breaks the mini-batch sampling computation into many steps and performs the computation in every step asynchronously. It runs inside the *trainer*’s process and uses multithreading to parallelize sampling computation and data copy.
- A *KVStore* that stores all vertex data and edge data distributed across machines. It provides the *pull* and *push* interfaces for pulling data from or pushing data to the distributed store.
- A *trainer* computes the gradients of the model parameters over a mini-batch. At each iteration, it first fetches mini-batch graphs from the sampler and corresponding vertex/edge features from the KVStore. It then runs the forward and backward computation on the mini-batches in parallel to compute the gradients. The gradients of dense parameters are dispatched to the *dense model update component* for synchronization, while the gradients

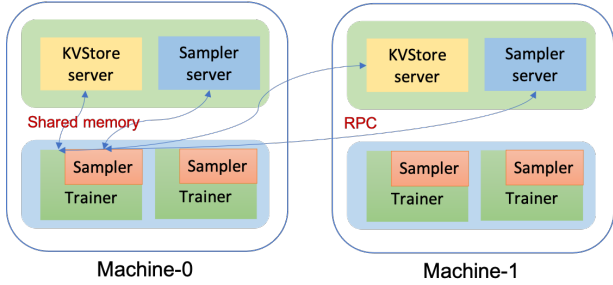


Figure 5: The deployment of DistDGLv2’s logical components on a cluster of two machines.

of sparse embeddings are sent back to the KVStore for update.

- A *dense model update component* for aggregating dense GNN parameters to perform synchronous SGD. DistDGLv2 reuses the existing components depending on DGL’s backend deep learning frameworks (e.g., PyTorch, MXNet and TensorFlow). For example, DistDGLv2 calls the all-reduce primitive when the backend framework is PyTorch [22], or resorts to parameter servers [21] for MXNet and TensorFlow backends.

When deploying these logical components to actual GPU hardware, the first consideration is to reduce the network traffic among machines. DistDGLv2 adopts the owner-compute rule (Figure 5). The general principle is to dispatch computation to the data owner to reduce network communication. DistDGLv2 first partitions the input graph with a light-weight min-cut graph partitioning algorithm. It then partitions the vertex/edge features and co-locates them with graph partitions. DistDGLv2 launches the sampler and KVStore servers on each machine to serve the local partition data. Trainers also run on the same cluster of machines and each trainer is responsible for the training samples from the local partition. This design leverages data locality to its maximum. Each trainer works on samples from the local partition so the mini-batch graphs will contain mostly local vertices and edges. Most of the mini-batch features are locally available via shared memory to reduce data copy in the local machine. We parallelize the computation with both multiprocessing and multithreading, where we run a trainer process for mini-batch computation in GPU and in each trainer we use multithreading to parallelize sampling computation in CPU. In the following sections, we will elaborate more on the design of each components.

5.2 Design principles

To construct an efficient system for distributed *hybrid CPU/GPU* training, we need to optimize the system in three aspects:

- *data locality*: we need to reduce data movement from the distributed CPU memory to GPUs,
- *load balancing*: we need to distribute mini-batch computation evenly across GPUs to keep all GPUs busy with the presence of mini-batch iteration barriers,
- *balance the workloads in CPUs and GPUs*: we need to saturate CPUs and GPUs simultaneously as well as communication channels, such as networks and PCIe, and avoid any components in the system from being the bottleneck.

There are two types of *data locality* when constructing mini-batches: *intra-batch locality* and *inter-batch locality*. Higher intra-batch locality requires target nodes within a mini-batch to be topologically close to each other. This data locality has two effects: 1) all nodes and their neighbors are close to each other, which allows better co-location of data and computation to reduce data copy; 2) it increases collisions among sampled neighbors, which results in a smaller number of input nodes and a smaller amount of data read for a mini-batch. Inter-batch locality indicates the overlap of input nodes of mini-batches that are processed subsequently. DistDGLv2 adopts hierarchical graph partitioning (Section 5.3) and always sample target nodes/edges from local partition. To take advantage of the data locality, DistDGLv2 splits the training workloads (Section 5.6.1) to co-locate mini-batch computation with graph data.

Load balancing has two scales: *partition level* and *batch level*. For the partition level, each partition needs to have roughly the same number of vertices and edges to ensure balanced workloads in an epoch. DistDGLv2 ensure this level of balancing during graph partitioning (Section 5.3.2) and data loading time (Section 5.6.1). Batch-level balancing refers to the balancing between all trainers within a mini-batch iteration. DistDGLv2 hides the imbalance of mini-batch sampling with ahead-of-time mini-batch sampling (Section 5.5).

To balance CPU/GPU computation as well as using all hardware resources simultaneously, we adopt two strategies: 1) split mini-batch generation into a pipeline with many stages and turn all computations into asynchronous operations to utilize all hardware resources simultaneously, 2) move computation to GPUs whenever possible (Section 5.5).

5.3 Hierarchical Graph Partitioning

DistDGLv2 splits the input graph to multiple partitions with a minimal number of edges across partitions before training. Graph partitioning is a preprocessing step. To amortize its overhead, we partition a graph once and use it for many training runs. This is necessary because GNN training requires hyperparameter tuning, which usually requires many training runs. As such, graph partitioning has to be model agnostic.

DistDGLv2 deploys hierarchical partitioning to handle multiple levels of data locality. The first level of partitions is assigned to machines to reduce data access across network. The second level of partitions is assigned to GPUs to reduce the number of nodes in a mini-batch. We only split the graph into physical subgraphs for the first-level partitions, as shown in Figure 6. In this level, we split the vertex/edge features along with graph partitioning; no vertex/edge features are duplicated. All trainers in a machine share the same partition subgraph and use node/edge split (Section 5.6.1) to take advantage of the second-level partitions.

After assigning vertices to a partition, DistDGLv2 assigns all incident edges of these vertices to the same partition to generate a physical graph partition. This ensures that all the neighbors of a vertex are accessible on the local partition so that samplers can compute locally without communicating to each other. With this partitioning strategy, some vertices and edges are duplicated (Figure 6). We refer to the vertices assigned by METIS to a partition as *core vertices* and the vertices duplicated by our edge assignment strategy as *HALO vertices*. All the core vertices also have unique partition assignments.

DistDGLv2 manages two sets of vertex IDs and edge IDs. DistDGLv2 exposes global vertex IDs and edge IDs for model developers to identify vertices and edges. Internally, DistDGLv2 uses local vertex IDs and edge IDs to locate vertices and edges in a partition efficiently, which is essential to achieve high system speed as demonstrated by previous works [35]. To save memory for maintaining the mapping between global IDs and local IDs, DistDGLv2 relabels vertex IDs and edge IDs of the input graph during graph partitioning to ensure that all IDs of core vertices and edges in a partition fall into a contiguous ID range. In this way, mapping a global ID to a partition is binary lookup in a very small array and mapping a global ID to a local ID is a simple subtraction operation.

5.3.1 METIS graph partitioning

DistDGLv2 adopts METIS [19] to partition a graph. METIS is a multi-level partitioning algorithm. It assigns densely connected vertices to the same partition to reduce the number of edge cuts between partitions. The partitions with IDs numerically close to each other are more densely connected than the ones whose IDs are far away. As such, METIS algorithm is a good fit with DistDGLv2’s multi-level partitioning.

METIS’ partitioning algorithms are based on the multilevel paradigm, which has been shown to produce high-quality partitions. However, for many types of graphs involved in learning on graphs tasks (e.g., graphs with power-law degree distribution), the successively coarser graphs become progressively denser, which considerably increases the memory and computational complexity of multilevel algorithms. To address this problem, we extended METIS to only retain a subset of

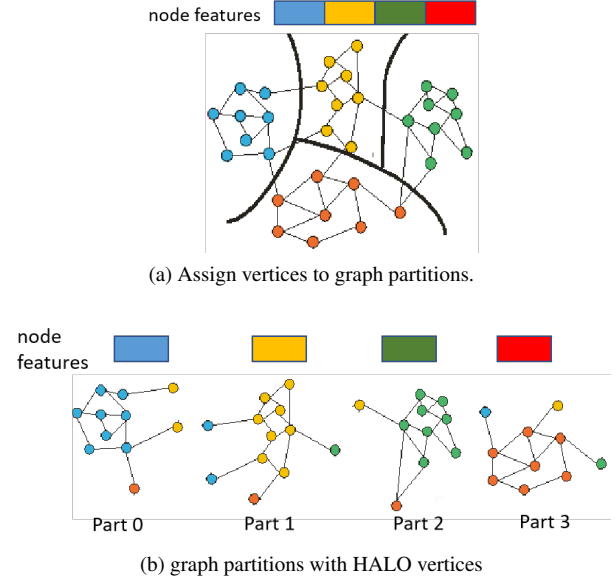


Figure 6: The first-level graph partitioning with METIS in DistDGLv2.

the edges in each successive graph so that the degree of each coarse vertex is the average degree of its constituent vertices. This ensures that as the number of vertices in the graph reduces by approximately a factor of two, so do the edges. To ensure that the partitioning solutions obtained in the coarser graphs represent high-quality solutions in the finer graphs, we only retain the edges with the highest weights in the coarser graph. In addition, to further reduce the memory requirements, we use an out-of-core strategy for the coarser/finer graphs that are not being processed currently. Finally, we run METIS by performing a single initial partitioning (default is 5) and a single refinement iteration (default is 10) during each level. For power-law degree graphs, this optimization leads to a small increase in the edge-cut (2%-10%) but considerably reduces its runtime. Overall, the set of optimizations above compute high-quality partitionings requiring $5\times$ less memory and $8\times$ less time than METIS’ default algorithms.

5.3.2 Load balancing

Minimizing edge cut reduces data communication in distributed training, but may result in imbalanced partitions, which leads to imbalanced training workloads. DistDGLv2 adopts two levels of load balancing strategies to balance the training workloads.

DistDGLv2 deploys multiple strategies to balance the physical graph partitions. By default, METIS only balances the number of vertices in a graph. This is insufficient to generate balanced partitions for synchronous mini-batch training, which requires the same number of batches from each partition per epoch and all batches to have roughly the same

size. We formulate this load balancing problem as a multi-constraint partitioning problem, which balances the partitions based on user-defined constraints [20]. 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 of different types.

5.4 Distributed Key-Value Store

The features of vertices and edges are partitioned and stored in multiple machines. Even though DistDGLv2 partitions a graph and assigns densely connected vertices to the same partition, we still need to read data from remote partitions. To simplify the data access on other machines, DistDGLv2 develops a distributed in-memory key-value store (KVStore) to manage the vertex and edge features as well as vertex embeddings.

DistDGLv2’s KVStore supports flexible partition policies to map data to different machines. For example, vertex data and edge data are usually partitioned and mapped to machines differently as shown in Section 5.3; in a heterogeneous graph, vertices of different types are partitioned and mapped to machines separately. DistDGLv2 defines separate partition policies for vertex data of different vertex types and edge data of different edge types, which aligns with the graph partitions in each machine.

Accessing vertex and edge features can easily become the bottleneck in GNN distributed training, especially in a cluster that does not have high-speed network. DistDGLv2 splits the training workloads to co-locate mini-batch computation with data to reduce data copy across the network. To keep up with the GPU computation, it is essential to reduce CPU memory copy. Thus, DistDGLv2 uses shared memory to access data in the local KVStore server to minimize data copy in preparing mini-batches for GPU training.

5.5 Asynchronous distributed mini-batch generation

DistDGLv2 provides a flexible and efficient pipeline for sampling mini-batches, illustrated by Figure 7. In this pipeline, every operation is asynchronous, including the computations in the local machine, such as sampling computation and CPU data copy. This pipeline effectively overlaps the computation and communication of different stages to simultaneously utilize all computation resources (e.g., CPU and GPU) and communication channels (e.g., network, CPU memory and PCIe) as well as hiding the imbalance of GNN mini-batch sampling in different trainers.

This pipeline has multiple stages (Figure 7): 1) a mini-batch scheduling 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, 2) neighbor sampling that samples multi-hop neighbors of the target nodes for GNN computation, 3) CPU prefetching that collects data from both local machines and remote machines for each mini-batch and store data in contiguous memory, 4) GPU prefetching that moves data from CPU to GPU, 5) subgraph compaction that remaps node IDs and edge IDs in the subgraph for mini-batch computation. We try to move heavier computations, such as subgraph compaction, to GPUs to reduce the workloads in CPU. Each of the components provides Python API for customization.

Because mini-batches are sampled independently from the training set, we can sample mini-batches ahead of time and process multiple mini-batches in a pipelining fashion. This allows us to take advantage of asynchronous computation in the pipeline to overlap the computation of different stages of multiple mini-batches to hide I/O latency (e.g., network communication and PCIe data transfer). This also allows better parallelization in mini-batch computation to speed up sampling. However, aggressive ahead-of-time mini-batch generation can potentially consume too much memory. To mitigate this problem, the pipeline allows different degrees of aggressiveness in different stages. For example, at the beginning of the pipeline (mini-batch scheduling and multi-layer neighbor sampling), we can work on many mini-batches simultaneously; in the middle of the pipeline, we prefetch node/edge data from remote machines for a relatively small number of mini-batches; at the end of the pipeline, we only move one mini-batch ahead of time to GPUs.

To realize asynchronous mini-batch generation pipeline, DistDGLv2 creates a dedicated thread for mini-batch sampling to overlap the sampling computation with mini-batch computation. We refer to this thread as a *sampling thread* and the original thread as a *training thread*. The sampling thread generates mini-batches and parallelizes sampling computation and CPU data copy with multithreading. The two threads exchange data via a shared queue. The sampling thread is not blocked by any global synchronization barrier.

Due to the synchronization barrier at the end of a mini-batch iteration, we have to ensure balanced computation among training threads on all machines and little interference from other threads. One potential interference comes from GPU CUDA synchronization. To avoid such interference, we break the asynchronous pipeline into two parts: all CPU operations, which includes mini-batch scheduling, distributed neighbor sampling and data copy from remote machines and local machines, run in the sampling thread; all GPU computations, which includes compacting subgraphs and data loading to GPUs, run in the training thread.

The asynchronous sampling pipeline could introduce a startup overhead when filling the pipeline at the beginning of every epoch. This hurts the performance especially when the training node set is small and sampling task is heavy. To remove the startup overhead, we run the asynchronous sam-

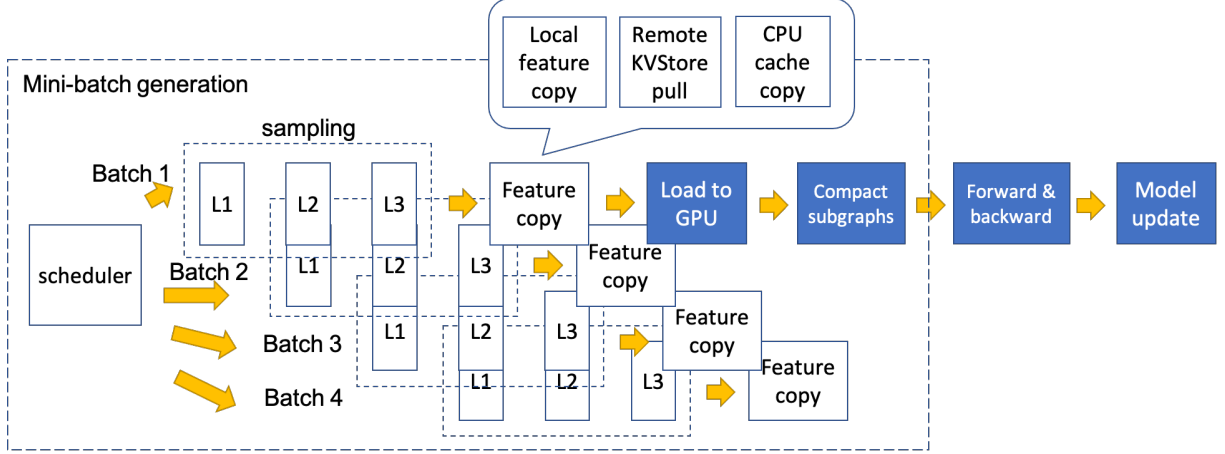


Figure 7: DistDGLv2 deploys an asynchronous mini-batch generation pipeline to deliver mini-batches to GPUs. The pipeline has five stages. The stages indicated by the blue boxes run on GPU in the training thread, while the stages in white boxes run in the sampling thread. The sampling thread performs sampling computation and feature copy for multiple mini-batches simultaneously and overlap the computations of different stages as well as with mini-batch computation. The feature copy can be decomposed into three operations that can run independently.

```
def sample_neighbors(g, seeds, fanouts):
    frontiers = []
    for i, fanout in enumerate(fanouts):
        # Sample neighbors from seed nodes
        frontier = dgl.sample_neighbors(g, seeds, fanout)
        # Seed nodes for the next layer.
        src, dst = frontier.edges()
        seeds = unique(src)
        frontiers.append(to_block(frontier))
    return blocks
```

Figure 8: Sample mini-batches for node classification.

pling pipeline throughout the entire training without stopping in the sampling thread. The trainer thread only needs to fetch mini-batches from the sampling thread.

5.5.1 Distributed neighbor sampler

DistDGLv2 keeps DGL’s Python sampling API design for supporting a variety of neighbor sampling algorithms in the literature [13, 36] while optimizing the ones that are friendly for distributed sampling.

Figure 8 illustrates DistDGLv2’s sampling code. It samples neighbor vertices in the ego-network of the seed vertices in multiple layers to constructs a mini-batch. DistDGLv2 provides some builtin functions to sample neighbors. For example, `sample_neighbors` samples neighbors of each vertex from the previous layer from the distributed graph. After sampling neighbors, `to_block` reconstructs the sampled subgraph

into a more compact form for mini-batch computation.

DistDGLv2 optimizes vertex-wise neighbor sampling algorithm [13], the most commonly used sampling method. This algorithm samples neighbors for each vertex on the adjacency matrix of the graph directly. The neighbor sampling computation runs on each vertex independently and, thus, can be easily decomposed and executed in multiple machines. Therefore, this algorithm is more lightweight than other sampling strategies, such as layer-wise sampling [36], which requires to collect all neighbors of the current layer to construct a vertex set before performing sampling computation. The main drawback is that mini-batches generated by the vertex-wise algorithm usually has more vertices than other sampling methods, such as layer-wise sampling, which leads to heavier burdens in data movement.

To perform the node-wise neighbor sampling in a distributed fashion, the trainer dispatches sampling requests for each seed vertex to the machines based on the vertex assignment produced by the graph partitioning algorithm. Upon receiving the request, sampler servers call DGL’s sampling operators on their partitions to sample neighbors and transmit the result back to the trainer process. Due to the METIS partitioning, majority of the seed vertices reside in the partition where the sampling requests are issued. To speed up the sampling on the local seed nodes, the trainer process access the graph structure of the local partition via shared memory to reduce the communication overhead. Finally, the trainer collects the results from different partitions and stitches them together to generate a subgraph.

A key optimization for accelerating sampling is to move some sampling computation to GPUs. `to_block` is a good candidate for two reasons: 1) the `to_block` computation is

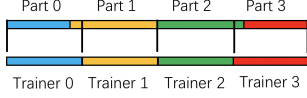


Figure 9: Split the workloads evenly to balance the computation among trainers.

much heavier than *sample_neighbors*; 2) *to_block* operates on sampled subgraphs, which are usually small enough to fit in GPU memory. As such, after sampling a subgraph, we move the subgraph to GPU and perform *to_block* on GPUs. To avoid the interference of GPU computations, we postpone the *to_block* computation and run it in the training thread.

5.5.2 Node/edge data prefetcher

It is time-consuming to load data to GPU for mini-batch computation. DistDGLv2 incorporates a prefetcher to move data closer to the computation devices in advance to overlap data movement with mini-batch computation. Data prefetching is constrained by memory consumption. For example, we cannot prefetch data too aggressively to GPUs, which consumes much GPU memory.

DistDGLv2 provides multiple prefetchers, which works together to load data to GPUs. The *CPU prefetcher* preloads data from remote machines and buffer the prefetched data in the local CPU memory. Because CPU memory has a large memory capacity, the CPU prefetcher can preload data for multiple mini-batches. The CPU prefetcher runs in the sampling thread. The *GPU prefetcher* loads data from CPU memory to GPUs and buffer the prefetched data in the GPU memory. Because GPU memory is scarce, the GPU prefetcher in DistDGLv2 only prefetches data for one mini-batch ahead of time. The data copy from CPU to GPU may interfere with GPU computations. Therefore, DistDGLv2 run the GPU prefetcher in the training thread.

5.6 Distributed mini-batch training

Mini-batch trainers jointly estimate gradients and update parameters of GNN models. We usually use synchronous SGD, commonly used to train deep neural networks, to update dense model parameters.

5.6.1 Split training workloads

To reduce data communication and achieve good load balancing, DistDGLv2 deploys a split algorithm to divide the training set with respect to the graph partitions in the system. The objective of the split algorithm is to localize most data accesses from local partitions and balance mini-batch computation of different trainers as well as local and remote data access. The split algorithm runs independently to select the training set for the local trainer when the distributed training

job is launched. The multi-constraint algorithm in METIS can only assign roughly the same number of training data points to each partition (as shown by the rectangular boxes on the top in Figure 9). Thus, the algorithm first ensures the same number of training data points is assigned to each trainer to work with synchronous SGD. It evenly splits the training data points based on their IDs and assigns the ID range to a machine whose graph partition has the largest overlap with the ID range. This is possible because we relabel vertex and edge IDs during graph partitioning and the vertices and edges in a partition have a contiguous ID range. There is a small misalignment between the training data points assigned to a trainer and the ones that reside in a partition. In practice, as long as the graph partition algorithm balances the number of training data points between partitions, the tradeoff is negligible.

Each trainer samples data points only from the set assigned to it to generate mini-batches. If each trainer samples uniformly at random, collectively the data points in a mini-batch iteration of all trainers are also sampled uniformly at random across the entire dataset. 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.

5.6.2 Memory management

Due to synchronous SGD, trainers are synchronized at the end of every mini-batch iteration. Thus, any slowdown in any of the trainers causes slowdown of all other trainers. We notice that CUDA memory allocation in both GPU and the host memory can block the remaining GPU operations and, thus, slows down the trainer and the entire training process. To overcome this problem, we use Pytorch memory allocator to allocate memory for all GPU operations, including the DistDGLv2 operations. In this way, we can effectively avoid memory allocation from CUDA directly.

6 Evaluation

In this section, we evaluate DistDGLv2 with multiple GNN models on large graph datasets. We benchmark three commonly used GNN models (GraphSAGE [13], Graph Attention Networks (GAT) [30] and Relational Graph Convolution Networks (RGCN) [27]) with two general tasks (node classification and link prediction) to evaluate the performance of DistDGLv2.

Our benchmarks use two medium-size graphs (ogbn-product [15] and Amazon [6]) and two large graphs (ogbn-papers100M [15] and mag-lsc [14]) (see Table 1 for various statistics). Note that even though all datasets contain labels for node classification, the number of labeled nodes in all but the smaller dataset (ogbn-product) is similar. As a result, the

Table 1: Dataset statistics.

Dataset	# Nodes	# Edges	Node features	# train nodes	# train links
OGBN-PRODUCT	2.4M	61.9M	100	197K	61.9M
AMAZON [6]	1.6M	264M	200	1.3M	264M
OGBN-PAPERS100M	111M	3.2B	128	1.2M	3.2B
MAG-LSC	240M	7B	756	1.1M	7B

cost to train node classification models for the larger graphs is not as high as the size of the graphs suggest. 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.

We perform hyperparameter search and select a set of hyperparameters that achieves good model accuracy on these datasets. For node classification, we use three GNN layers and the hidden size of 256; the fanout of each layer in mini-batch training 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. 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) to train GNN models and a cluster of four AWS EC2 r5d.24xlarge instances (96 vCPU, 768GB RAM) to preprocess the graph. Both clusters connect machines with 100Gbps network.

In all experiments, we use DGL v0.7 and Pytorch 1.8. For Euler experiments, we use Euler v2.0 and TensorFlow 1.12.

6.1 DistDGLv2 vs. other distributed GNN frameworks

We compare the training speed of DistDGLv2 with DistDGL [33] and Euler [1], the state-of-the-art distributed GNN mini-batch training frameworks, on four g4dn.metal instances. Both DistDGL and Euler are designed for distributed CPU training, so we run them on four r5dn.24xlarge instances as well to collect their CPU training speed. To have a fair comparison with DistDGLv2, we change them 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. Both DistDGLv2 and DistDGL’s distributed training use METIS partitioning and co-locate data with computation. DistDGL uses multithreading and multiprocessing to parallelize computation. Therefore, it does not require many training processes to speed up. In contrast, Euler uses random partitioning and parallelizes computation completely with multiprocessing for both mini-batch computation as well as sampling inside a trainer. To have a fair comparison, we run mini-batch training with the same global batch size (the total size of the batches of all trainers in an iteration) to get the

same convergence.

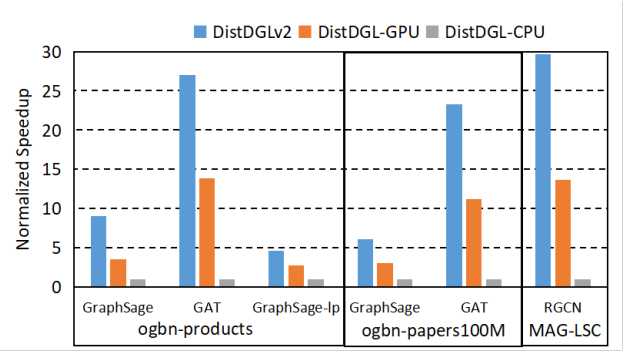


Figure 10: The speedup of DistDGLv2 and DistDGL-GPU over DistDGL-CPU.

Figure 10 shows that DistDGLv2 gets $2 - 3\times$ speedup over DistDGL-GPU on various datasets and tasks. We cannot train GraphSage with link prediction (GraphSage-lp) on OGBN-PAPERS100M by using DistDGL because it runs out of memory in this task. 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 uses METIS to partition 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 with DistDGL on CPUs (DistDGL-CPU). DistDGL is optimized for CPU training. Figure 10 shows up to $30\times$ speedup over DistDGL-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, which has much less computation, DistDGLv2 gets $6 - 9\times$ speedup.

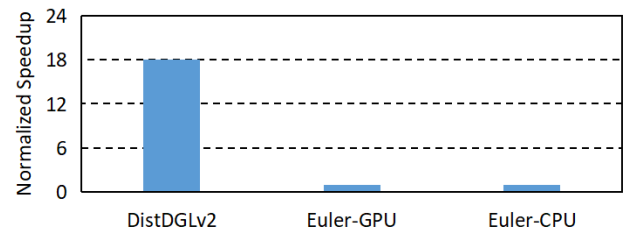


Figure 11: The speedup of DistDGLv2 and Euler-GPU over Euler-CPU for training GraphSage on OGBN-PRODUCT.

We further compare DistDGLv2 with Euler on CPUs and GPUs when training GraphSage on OGBN-PRODUCT (Figure

11). 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 good performance. This parallelization strategy works relatively well on CPU clusters. However, it 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. Therefore, Euler cannot well parallelize the sampling computation and fetch data efficiently for GPU trainers. This indicates that effective distributed GNN training on GPUs requires both multiprocessing and multithreading.

6.2 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. We use the batch size of 1000 per trainer.

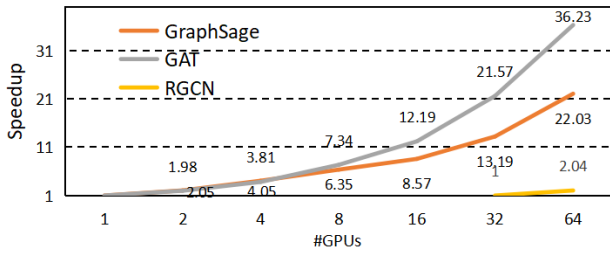


Figure 12: The speedup of DistDGLv2 on OGBN-PAPERS100M (GraphSage and GAT) and MAG-LSC (RGCN).

As shown in Figure 12, DistDGLv2 achieves about $20\times$ speedup in GraphSage and $36\times$ 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. The training speed doubles when we scale the RGCN training on MAG-LSC 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. These experiments show that as the result of the various optimizations, DistDGLv2 can scale to large graphs with hundreds of millions of nodes. It takes only 5 seconds to train the GraphSage and 7 seconds to train GAT model on the OGBN-PAPERS100M graph in a cluster of 64 GPUs.

6.3 Training convergence

Each trainer of DistDGLv2 samples data points from its graph partition, but collectively, the data points in a global mini-

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

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

batch are sampled uniformly at random from the entire training set. This training method is a little similar to ClusterGCN, 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.

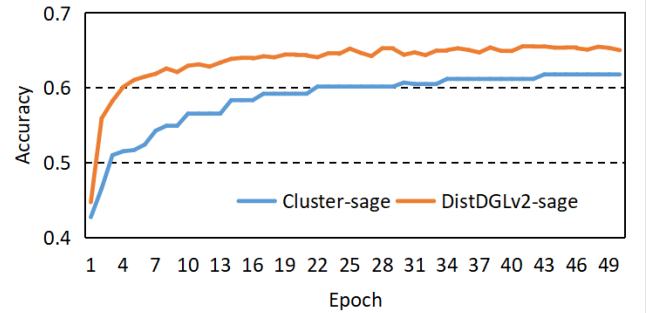


Figure 13: Convergence of DistDGLv2 vs. ClusterGCN on OGBN-PAPERS100M.

As shown in Figure 13, we observe slower convergence of ClusterGCN 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 can estimate the neighbor aggregation in an unbiased fashion, while ClusterGCN’s estimation of neighbor aggregation depends on graph partitioning results. This indicates that we have to sample neighbors in other partitions to achieve good model accuracy.

6.4 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 [18], a distributed version of METIS, 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.

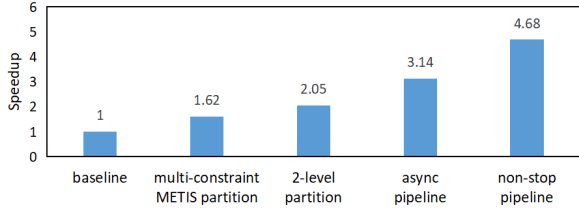


Figure 14: The speedup of various techniques for GraphSage on OGBN-PRODUCT.

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 node 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.

6.5 Ablation Study

DistDGLv2 deploys many optimizations. In this section, we study the effectiveness of the main optimizations: 1) multi-constraint METIS partition that performs the first-level partition on the graph with min-edge cut and balance the number of nodes and edges as well as the training set; 2) 2-level partition that splits the graph for both machines and GPUs; 3) asynchronous pipeline that performs every operation in mini-batch generation asynchronously to overlap CPU computation, GPU computation and network I/O; 4) non-stop asynchronous pipeline that runs the pipeline continuously until the training completes. 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. We do not study the graph partitioning with load balancing separately. When we partition a graph, we split it into balanced partitions; we also divide workloads evenly for both the first-level and second-level partitions. We use a cluster of four g4dn.metal instances to run the experiments.

Figure 14 shows each optimization has impact in performance and we get overall $4.7\times$ speedup for GraphSage on OGBN-PRODUCT after adding all optimizations. Even though this cluster already has 100Gbps network, we observe $1.62\times$ speedup by using balanced METIS partitions. 2-level partitioning gives further speedup because confining the training

vertices in a smaller partition leads to better intra-batch locality and a smaller number of neighbor vertices in a mini-batch. Asynchronous sampling pipeline overlaps the CPU computation, GPU computation and network I/O to hide the latency of network I/O, PCIe data transfer and CPU data copy. When using non-stop asynchronous pipeline, we get another performance boost. This suggests that the asynchronous pipeline has relatively large startup overhead in a graph with a small training set. By running the pipeline asynchronously, we can effectively remove the startup overhead.

7 Conclusion

We develop DistDGLv2 for distributed GNN training in a GPU cluster. It adopts hybrid CPU/GPU training by keeping graph data in CPU memory and perform mini-batch computation in GPU to support efficient GNN training on very large graphs. We show that hybrid CPU/GPU training can get speedup by a factor of 5 – 30 over distributed CPU training. The more complex GNN model is, the higher speedup we can get in this training strategy.

DistDGLv2 adopts many optimizations to make GNN training more efficient and scalable in a cluster of GPUs. We show that using METIS partitioning and co-locate data with mini-batch computation is important even in a cluster with very fast network. By deploying an asynchronous pipeline for generating mini-batches, we can effectively hide the latency of network I/O, PCIe data transfer and CPU memory copy. By having all optimizations, DistDGLv2 gets $2 - 3\times$ speedup over DistDGL and $18\times$ speedup over Euler on GPUs.

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] Saide Isilay Baykal, Deniz Bulut, and Ozgur Koray Sahingoz. Comparing deep learning performance on bigdata by using cpus and gpus. In *2018 Electric Electronics, Computer Science, Biomedical Engineering's Meeting (EBBT)*, pages 1–6. IEEE, 2018.
- [4] Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. volume 80 of *Proceedings of Machine Learning Research*, pages 942–950, Stockholmssmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

- [5] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations*, 2018.
- [6] 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.
- [7] David M. Dawson. Lessons learned from porting llnl applications to sierra. In *GTC*, 2019.
- [8] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.
- [9] 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.
- [10] 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.
- [11] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, November 2012.
- [12] Google. Freebase data dumps. <https://developers.google.com/freebase/data>.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. Adaptive sampling towards fast graph representation learning. *arXiv preprint arXiv:1809.05343*, 2018.
- [17] 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.
- [18] 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>.
- [19] 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.
- [20] 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.
- [21] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI’14*, page 583–598, USA, 2014. USENIX Association.
- [22] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.
- [23] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Paragraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 401–415, 2020.
- [24] 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.
- [25] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD ’10*, page 135–146, New York, NY, USA, 2010.

- [26] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramana-
narayan Mohanty, Evangelos Georganas, Alexander Hei-
necke, Dhiraj D. Kalamkar, Nesreen K. Ahmed, and
Sasikanth Avancha. Distgnn: Scalable distributed train-
ing for large-scale graph neural networks. *CoRR*,
abs/2104.06700, 2021.
- [27] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem,
Rianne van den Berg, Ivan Titov, and Max Welling. Mod-
eling relational data with graph convolutional networks.
CoRR, abs/1703.06103, 2017.
- [28] 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 train-
ing with distributed CPU servers and serverless threads.
In *15th USENIX Symposium on Operating Systems De-
sign and Implementation (OSDI 21)*, pages 495–514.
USENIX Association, July 2021.
- [29] Alok Tripathy, Katherine Yelick, and Aydin Buluc. Re-
ducing communication in graph neural network training.
arXiv preprint arXiv:2005.03300, 2020.
- [30] Petar Veličković, Guillem Cucurull, Arantxa Casanova,
Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph
attention networks. *CoRR*, abs/1710.10903, 2018.
- [31] 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.
- [32] Dalong Zhang, Xin Huang, Ziqi Liu, Zhiyang Hu, Xi-
anzheng 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.
- [33] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qi-
dong Su, Xiang Song, Quan Gan, Zheng Zhang, and
George Karypis. Distdgl: Distributed graph neural net-
work training for billion-scale graphs. *arXiv preprint
arXiv:2010.05337*, 2021.
- [34] 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.
- [35] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xi-
aosong Ma. Gemini: A computation-centric distributed
graph processing system. In *12th USENIX Sympo-
sium on Operating Systems Design and Implementation
(OSDI 16)*, November 2016.
- [36] 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.