ByteGNN: Efficient Graph Neural Network Training at Large Scale

Chenguang Zheng[1,2,†], Hongzhi Chen[2,*], Yuxuan Cheng[2], Zhezheng Song[1], Yifan Wu[2,3,†], Changji Li[1,2,†], James Cheng[1], Hao Yang[2], Shuai Zhang[2]
[1]{cgzheng, cjli, jcheng}@cse.cuhk.edu.hk, [1]szaizai18@gmail.com, [2]{zhengchenguang, chenhongzhi, chengyuxuan.911, wuyifan.18, lichangji, yanghao.2019, zhangshuai.root}@bytedance.com, [3]yifanwu@pku.edu.cn
[1]The Chinese University of Hong Kong, [2]ByteDacne Inc, [3]Peking University

ABSTRACT

Graph neural networks (GNNs) have shown excellent performance in a wide range of applications such as recommendation, risk control, and drug discovery. With the increase in the volume of graph data, distributed GNN systems become essential to support efficient GNN training. However, existing distributed GNN training systems suffer from various performance issues including high network communication cost, low CPU utilization, and poor end-to-end performance. In this paper, we propose ByteGNN, which addresses the limitations in existing distributed GNN systems with three key designs: (1) an abstraction of mini-batch graph sampling to support high parallelism, (2) a two-level scheduling strategy to improve resource utilization and to reduce the end-to-end GNN training time, and (3) a graph partitioning algorithm tailored for GNN workloads. Our experiments show that ByteGNN outperforms the state-of-the-art distributed GNN systems with up to 3.5-23.8 times faster end-to-end execution, 2-6 times higher CPU utilization, and around half of the network communication cost.

PVLDB Reference Format:
Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. ByteGNN: Efficient Graph Neural Network Training at Large Scale. PVLDB, 15(6): 1228 - 1242, 2022.
doi:10.14778/3514061.3514069

1 INTRODUCTION

Existing systems for neural network training, such as TensorFlow [5] and PyTorch [54], are designed for training euclidean data such as images, texts and audio data. In recent years, a new type of neural networks, called **graph neural networks (GNNs)**, become popular because of the ubiquity of graph data today such as semantic web graphs, knowledge graphs, social networks, and e-commerce networks. GNNs combine the non-euclidean graph structures with traditional neural networks to extract rich information from graph data for machine learning. Recent research results [16, 31, 43, 64, 71, 75] have shown that GNNs achieve significant performance improvements over traditional methods on many important tasks such as node classification, link predication, and graph clustering. GNNs have been applied in a broad range of applications including recommendation systems [52, 75], computer vision [50, 58], natural language processing [55, 73], drug discovery [24], and social networks [68].

Although many graph computing systems [8, 9, 18, 22, 27, 42, 49, 51, 67, 72, 79, 81] have been proposed, they are designed for batch graph analytics workloads such as the computation of PageRank, shortest paths, label propagation and connected components, and thus they lack of operators for neural network training. Thus, dedicated GNN systems have been developed upon neural network training systems (e.g., TensorFlow, PyTorch) for GNN training.

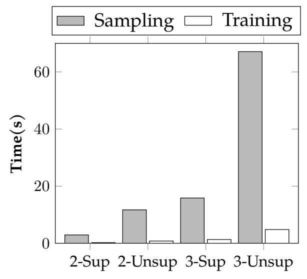
Among existing GNN systems, most of them are still single-machine systems, e.g., DGL [66], PyTorch Geometric (PyG) [21], NeuGraph [48], FeatGraph [33] and Seastar [70], which are optimized for training GNN models on a relatively small graph but cannot scale to process large graphs generally available in industry today. Note that for GNNs, a graph not only contains the graph topology information (which is typically used for computations such as PageRank, shortest paths, etc.), but each vertex and edge in the graph also contain a feature vector. Thus, depending on the dimensions of the feature vectors (typically around 100 to hundreds), the size of a graph for GNNs can be easily many times larger than the graph topology processed by existing graph computing systems. For example, for the Ogbn-Papers [32] graph used in our experiments, a feature vector has 128 dimensions and the size of the features is 4 times larger than the size of the graph topology.

For GNN training on large graphs, distributed systems such as Euler [1], GraphLearn (also called AliGraph) [80], AGL [77] and DistDGL [78] have been proposed. During the training, these systems collect and aggregate the feature vectors of the K-hop neighbors in order to compute the feature vector of each vertex, where K is the number of layers of the GNN model to be trained. However, the K-hop neighbors of a vertex can be many, especially for a power-law graph, and a large portion of them can be located in remote machines. Thus, fetching **all** the K-hop neighbors to a local machine for each vertex (referred to as **full mini-batch training**) incurs high network communication overheads and memory consumption.

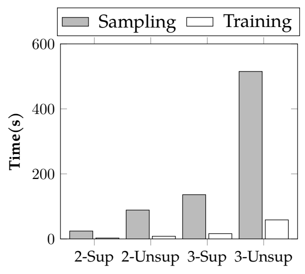
To address the problem of full mini-batch training, **mini-batch sampling training** was proposed [13, 31, 34], which works as follows. Distributed GNN training is conducted in iterations and for each iteration, a machine processes a mini-batch of vertices in

[*] Hongzhi Chen is the Corresponding Author.
† This work was done when the authors were in ByteDance.
The work of Chenguang Zheng, Changji Li and James Cheng was partially supported by a ByteDance Research Collaboration Project.

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514069



(a) Reddit



(b) Ogbn-Product

Figure 1: Sampling and training time of GraphLearn

its partition in two phases: (1) **the sampling phase** — for each vertex v in the mini-batch, the sampler samples a limited number of neighbors of v in each hop, fetches the sampled remote neighbors to the local machine, and constructs the neighborhood subgraph of v from its sampled neighbors locally; (2) **the training phase** — the trainer trains the model on the neighborhood subgraphs of the vertices in the mini-batch locally.

While distributed mini-batch sampling has become the default method for GNN training on a large graph (for which full-batch training and full mini-batch training are not practical), existing distributed GNN training systems suffer from a number of performance problems. One main problem is that sampling can take significantly longer time to complete than training, due to large amounts of random data access and remote data fetching involved in the sampling phase. For example, Figure 1 reports the average sampling time and training time in an epoch of training a 2-layered and 3-layered GraphSAGE model (both supervised and unsupervised) on four machines by GraphLearn [80] on the Reddit dataset [31] and Ogbn-Product dataset [32], which shows that the sampling phase takes an order of magnitude longer time than the training phase. For example, in the 2-layer supervised GraphSAGE training on Ogbn-Product, GraphLearn's training time is only 2.66s while its sampling time is 24.17s. Under the same setting, the sampling time of DistDGL [78] is also 4.22x of its training time.

The imbalance between the sampling and training phases also leads to the under-utilization of computing resources and the problem is worsen if GPUs are used for training (which further widens the gap between the sampling and training time) [59]. To address this imbalanced computing pattern in mini-batch GNN training, existing systems have attempted to apply neighborhood caching [46] and fixed size prefetching [78] to shorten the sampling time. However, it is difficult to set the right hyper-parameters (i.e., cache ratio and prefetching number) for training different GNN models on different graphs. Nextdoor [36] proposed to sample neighborhood using GPUs, but the GPU memory capacity limits the size of the graph it can handle. Graph partitioning has also been applied to reduce the cost of remote data fetching [80]. However, existing graph partitioning algorithms are designed for traditional graph workloads (e.g., distributed PageRank) and they do not consider the data access pattern and load balancing in GNN training.

In this paper, we propose ByteGNN, a distributed GNN training framework to support fast end-to-end GNN training in large graphs. To improve the efficiency of sampling, we abstract the sampling phase of a mini-batch as a directed acyclic graph (DAG) of small tasks, so that we can run DAGs and tasks within each DAG in parallel. The fine-grained task abstraction in DAG modeling also leads to the design of a two-level scheduling. First, coarse-grained scheduling determines how much resources should be used for mini-batch sampling, in order to dynamically adjust the computation loads between the sampling and training phases to avoid resource contention and maximize CPU utilization. Then, fine-grained scheduling decides the execution order of tasks in the DAGs in order to pipeline the sampling outputs to be consumed by the training phase at the right pace. The two scheduling strategies work together to minimize the end-to-end GNN training time. We also propose an effective graph partitioning algorithm tailored for mini-batch graph sampling, which maintains the data locality according to the data access pattern of mini-batch sampling and balances the computation loads in the training, validation and testing stages.

We implemented ByteGNN based on GraphLearn [4]. Our performance evaluation shows that ByteGNN achieves significantly higher training throughput and is more scalable than the state-of-the-art distributed GNN systems. Experimental results show that ByteGNN achieves up to 23.8x speedup over GraphLearn and 3.5x over DistDGL. The results verify that our system designs lead to efficient GNN training.

2 BACKGROUND AND MOTIVATION

We first introduce the necessary background of GNN and briefly discuss sampling-based GNN training. Then, we motivate our work by presenting the limitations of existing systems for large-scale GNN training.

2.1 Graph Neural Networks

GNN models are designed to capture the information contained in both the relationship among vertices in a graph and the vertex/edge attributes. The core idea of GNNs is recursively aggregating the neighbor information, including the features of the neighbors and the features of the connecting edges, and then applying feature transformation functions.

Take the GraphSAGE model [31] as an example. The training process for one layer of the model can be expressed as follows:

$$h^k_{\mathcal{N}(v)} \leftarrow AGGREGATE_k(\{h^{k-1}_u, \forall u \in \mathcal{N}(v)\}), \tag{1}$$

$$h^k_v \leftarrow \sigma(W^k \cdot CONCAT(h^{k-1}_v, h^k_{\mathcal{N}(v)})), \tag{2}$$

where $\mathcal{N}(v)$ is the set of neighbors of vertex v. In this k-th convolution layer, each vertex v first uses the *AGGREGATE* function to collect the feature vectors of v's neighbors from the $(k-1)$-th layer. The aggregation result is then concatenated with v's feature vector from the $(k-1)$-th layer, followed by a dot-product operation with a learnable weight matrix $\mathbf{W}$. The dot-product result is further transformed by a nonlinearity activation function σ such as the sigmoid function, which gives the feature vector of v for the k-th layer.

As the number of layers increases, the vertices are required to gather and aggregate more and more information from neighbors that are farther away (i.e., expanding from the i-hop neighbors to the j-hop neighbors for $j > i$). When the training for all the K layers completes (for a user-specified K), the final feature vector h_v^K for each vertex v is fed into a mapping function for a specific downstream task (e.g., node classification, link prediction).

2.2 Distributed Mini-Batch Graph Sampling

Existing distributed GNN systems adopt data parallelism and sampling is commonly applied in order to train a GNN model on a large graph efficiently. However, the sampling process in distributed GNN training is quite different from that in training DNN models for computer vision and natural language processing, for which each sample is independent and small. For GNNs, the distributed training may access the entire neighborhood of a sampled vertex, including both vertices and edges along with their feature vectors. Due to the structural connection among vertices in different partitions, the data access pattern usually leads to high network communication cost.

In a K-layered GNN model, for each sampled *seed* vertex v, we need to obtain the K-hop neighborhood of v to construct a neighborhood subgraph to update v's feature vector. As most real-world graphs have a power-law degree distribution, the size of the K-hop neighborhood subgraph of a vertex grows exponentially as the number of hops increases. To address this problem, **mini-batch neighborhood sampling** [46, 78, 80] has been used to sample a limited number of neighbors for each sampled seed vertex. Figure 2 illustrates how mini-batch graph sampling is applied in the training of a 2-layered GNN model. We show the sampled 2-hop neighborhood subgraphs of two seed vertices, v_1 and v_2, where we set the sampling configuration $D_1 = 2$ and $D_2 = 2$, meaning that a vertex v first samples at most D_1 of its 1-hop neighbors, and then each u of v's sampled 1-hop neighbors further samples D_2 of u's neighbors. Remote sampling requests are sent to remote devices to access the i-hop neighbors that are stored there. After the sampling finishes, the sampled neighbors, along with their attributes (which are used to construct the initial feature vectors), are fetched to the local device of v_1 (and v_2) to construct its sampled 2-hop neighborhood subgraph, which is then fed into the GNN model to calculate the gradients and update the model parameters.

Although the tradeoff is a potential loss in the model accuracy, mini-batch graph neighborhood sampling still converges to the required model accuracy. Take the mean aggregator in GraphSAGE [31] as an example, using the Monte Carlo estimation, for the layer k, we obtain:

$$\mathbb{E}[h^k_{\mathcal{N}^s(v)}] = \mathbb{E}[\frac{1}{|\mathcal{N}^s(v)|}\sum_{u\in\mathcal{N}^s(v)} h_u^{k-1}] = \frac{1}{|\mathcal{N}(v)|}\sum_{u\in\mathcal{N}(v)} h_u^{k-1} = h^k_{\mathcal{N}(v)},$$

where $\mathcal{N}^s(v)$ is the set of random sampled neighbors of vertex v and $|\mathcal{N}^s(v)| = D_k$, which is the fanout of layer k. Unfortunately, though $h^k_{\mathcal{N}^s(v)}$ is an unbiased estimator of $h^k_{\mathcal{N}(v)}$, $h^k_{v^s}$ is not an unbiased estimator of h_v^k due to the non-linearity of $\sigma(\cdot)$ in Equation (2) [14]. Thus, the gradient is biased and the convergence of SGD is not guaranteed, unless the fanout D_k goes to infinity. But in practice, GraphSAGE sets $D_1 = 25$ and $D_2 = 10$ to provide statistically significant gains over existing approaches [43, 56]. In addition, AGL [77] also reported that with a suitable sample size, the sample can well approximate the ground truth.

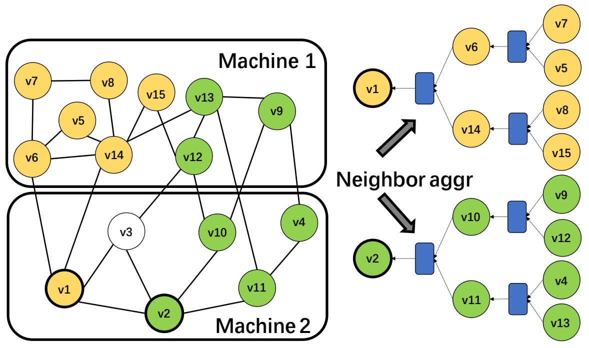


Figure 2: 2-hop mini-batch graph sampling

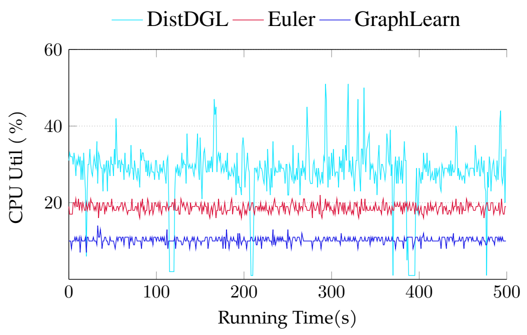


Figure 3: The CPU utilization of different systems

2.3 Limitations of Existing GNN Systems

Existing systems for distributed GNN training suffer from the following major limitations.

(1) The overhead of network communication is large. As the sampling procedure shows in Section 2.2, for every sampled seed vertex v in each iteration, we need to construct v's K-hop neighborhood subgraph together with the feature vectors of the vertices and edges in the subgraph. For example, in a 3-hop neighborhood subgraph where the sampling configuration is set at $k_1 = 15$, $k_2 = 10$ and $k_3 = 5$ in the Reddit dataset [31], there are 915 vertices each with a 602-dimension feature vector, which are what we need to prepare for one sampled seed vertex. As many of the neighbors and their features may be stored in remote machines, the K-hop neighborhood subgraph construction incurs a high network communication overhead. Figure 13(a) shows that the number of remote vertices is about six times that of local vertices with the widely used Hash partitioning. In fact, existing graph partitioning algorithms only consider to reduce the inter-partition edges, but do not consider the data access pattern and load balancing of graph sampling in GNN training. This calls for a new design of a more effective graph partitioning strategy tailored for GNN training.

(2) CPU utilization is low. Our performance profiling shows that existing distributed GNN systems had poor CPU utilization as shown in Figure 3. By analyzing their system designs, we list the main causes to their low CPU utilization below.

The sampling phase of GraphLearn [80] is handled using the Gremlin semantics [2, 3] to express each sampling step. For each step, the Gremlin statement is translated by a parser and converted into several execution operations. An operation is a minimum execution unit in GraphLearn. GraphLearn has low CPU utilization since all the graph sampling operations within each device do not overlap with each other. DistDGL [78] takes a similar approach but also has many optimizations such as replicating the neighbors of its local vertices.

Euler [1], on the other hand, wraps each graph operator into one TensorFlow dataflow operator. This design is convenient for users to build the whole computation graph in TensorFlow. However, as all the sampling operators and the training operators are contained in one big computation dataflow graph, existing deep learning systems (including TensorFlow) cannot process it efficiently. It is difficult to have the optimal execution order for the dataflow graph with the newly defined graph sampling operators, which is totally different from the normal tensor computation. Besides, due to the convergence requirement, TensorFlow only runs one dataflow graph at a time. Each iteration always starts graph sampling after the previous training process finishes. This design also eliminates the opportunities to apply the data prefetching mechanism to the independent sampling stages.

(3) GPU does not bring enough benefit for GNN training in large graphs. As mentioned in Section 1, distributed GNN training on large graphs consists of the sampling phase and training phase. Due to limited GPU memory capacity, graph data are stored in the host memory of the machines and thus the sampling phase is conducted by CPUs. When GPUs are used to conduct the training phase, the sampling results are loaded into GPU memory from CPU host memory via PCIe links. As shown in [46, 59], even in the case of single-machine GNN training using GPU (i.e., data are not fetched through network), the sampling and data loading time still take a significant portion of the end-to-end training time. The training phase can indeed be accelerated using GPUs (compared with using CPUs), but this only reduces the model updating time while the sampling phase still dominates the overall processing cost. This is because most GNN models are considerably small (unlike DNN models) and the training phase only needs to conduct model computation on densely packed vectors, while sampling a large graph involves large amounts of random data access and remote data fetching in order to construct the neighborhood subgraph for each sampled seed vertex.

We evaluated the performance of DistDGL [78] on a GPU server (40 cores Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz, 256 GB Memory, and one Nvidia RTX 2080Ti graphics card). We tested DistDGL with different fanout and hidden sizes to show the influence from the workloads of sampling and training. As Figure 4 shows, the largest difference in epoch time is only 10% between using GPU and using CPU. The average GPU utilization for GNN training is only around 20%, which is consistent with the GPU utilization of DGL reported in [46]. In fact, even if we purely use CPUs for the training phase, the sampling phase still dominates the overall cost as we have shown in Figure 1. In addition, we also need to consider the operational costs. GPU servers are expensive and GPU quota is more restricted to training DNNs even in big companies like ByteDance. In the cloud environment, Dorylus [61] also shows that

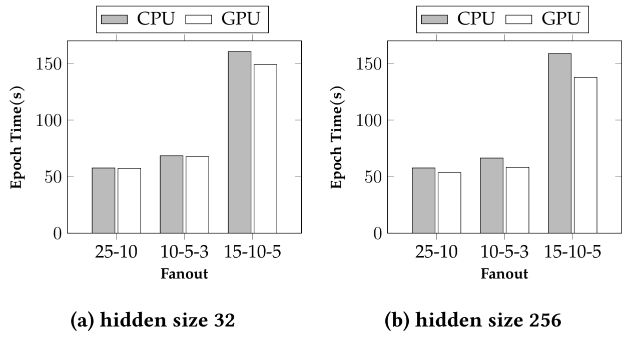


Figure 4: The epoch time of DistDGL with different hidden sizes and fanout on the Ogbn-product dataset

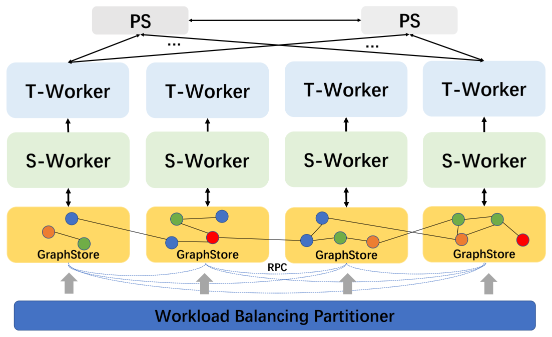


Figure 5: System architecture of ByteGNN

GPU-based training is only cost-effective for small, dense graphs. All the above concerns therefore divert our focus to designing a CPU-based framework for large-scale GNN training.

Motivation summary. The analysis above motivates us to design (1) a computation paradigm that uses only CPUs and aims to maximize CPU utilization by adaptively allocating computing resources to the sampling and training phases according to their needs, and (2) a new graph partitioning algorithm in order to reduce massive network communication caused by graph sampling in GNN training.

3 SYSTEM DESIGN

Figure 5 shows the architecture of ByteGNN, which consists of four main components in each machine where ByteGNN is deployed. **Graph Store** stores a partition of the input graph data and the Graph Stores of all machines form a distributed Graph Store. **PS** is a parameter server that stores the model parameters. **Sampling Worker** (**S-Worker**), handles the sampling phase and constructs sampled neighborhood subgraphs for sampled seed vertices. **Training Worker** (**T-Worker**), handles the training phase, which computes model gradients on the sampled neighborhood subgraphs constructed by the S-Worker in the same machine and synchronizes the gradients with PS to update the model parameters.

In Sections 3.1-3.3 we focus on three key designs in ByteGNN, which address the limitations of existing GNN systems discussed in Section 2.3.

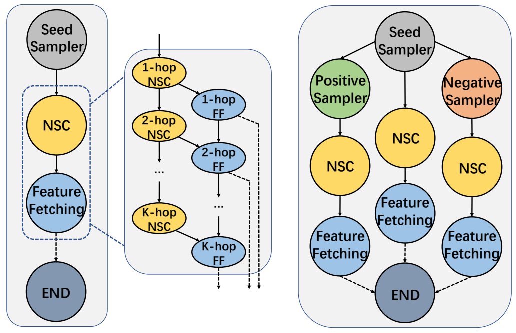


Figure 6: The DAG of the sampling workflow

3.1 Abstraction of Mini-Batch Graph Sampling

The sampling process in existing GNN systems [21, 78, 80] is not well-organized as the tasks in each sampling phase are executed without overlapping, which often leads to CPU under-utilization. In addition, sampling is conducted for one iteration (i.e., one mini-batch) after another, even though different mini-batches are independent of each other. To support parallel sampling within a mini-batch and among mini-batches, so as to maximize CPU utilization, we model the sampling process as a DAG of tasks. Then, we can execute the DAGs of sampling multiple mini-batches in parallel. We also introduce a scheduler in Section 3.2 to effectively utilize the computing resources for both intra-DAG and inter-DAG parallelization, while balancing the loads between sampling and training so that one is not waiting for the other to finish in order to continue.

To construct this DAG for a general GNN model, we analyzed the sampling phase of a broad range of existing GNN models, i.e., those that follow a similar neighborhood aggregation as described in Section 2.1, which cover most of the widely-adopted models such as GCN [43], GAT [64], GraphSAGE [31], PinSAGE [75], and GraphSAINT [76]. We provide a common abstraction for the sampling phase of these GNN models with a set of five operators: (1) **Seed Sampler**: sampling a set of vertices as *seeds* from the local graph store; (2) **Positive Sampler**: sampling vertices from the direct neighbors of each seed; (3) **Negative Sampler**: sampling vertices from those that are not the direct neighbors of each seed; (4) **Neighborhood Subgraph Construction (NSC)**: sampling vertices from the multi-hop neighborhood of a given vertex and constructing the sampled neighborhood subgraph; (5) **Feature Fetching**: fetching the attributes of a given vertex/edge to construct its feature vector.

With the above five operators, we can present the workflow of the sampling phase as a DAG, as shown in Figure 6. The DAG on the left of Figure 6 models supervised training, which consists of three tasks: Seed Sampling, NSC, and Feature Fetching. For unsupervised training, we also need to construct the neighborhood subgraphs of each positively and negatively sampled vertices of the seed vertices, as shown in the DAG on the right of Figure 6. The three branches in the DAG for unsupervised training can be executed in parallel, and the results are then collected in the "End" node to be fed into a T-Worker for training.

To enable higher parallelism for both supervised and unsupervised training, we create an instance of the two dominating operations (i.e., NSC and Feature Fetching, as they access multi-hop neighbors and their attributes) for each sampled vertex and execute these instances in parallel. In addition, as NSC (along with Feature Fetching) is executed repeatedly for each hop of neighborhood expansion, we can break the multi-hop operations into many smaller tasks of one-hop operations. As shown in Figure 6, each small task of Feature Fetching can start immediately when the corresponding small NSC task finishes. The more fine-grained task abstraction results in higher parallelism and better resource utilization (e.g., less head-of-line blocking and stragglers, less fragmentation in resource utilization).

To construct a DAG, users only need to specify the customized sampling functions in Seed Sampler, Positive Sampler, Negative Sampler, and also in NSC (e.g., how and how many neighbors in each hop should be sampled). This design also leaves space for researchers and engineers to explore new, high-quality sampling strategies using the framework. Note that the logical DAG is created only once and physical instances are generated and executed for each mini-batch by the S-Workers.

3.2 Two-Level Scheduling

ByteGNN adopts a two-level scheduling strategy to improve CPU utilization and reduce the end-to-end GNN training time. Although many scheduling strategies have been proposed, they are mostly for job scheduling at the cluster level [17, 19, 20, 25, 28–30, 35, 40, 47, 57, 60, 63, 74] or heterogeneous jobs/tasks in dataflow systems [39], which are over-complicated and incur extra overheads for scheduling the simple tasks in our system (note that for the training of a GNN model, we only need to schedule instances of the same DAG instead of many different DAGs).

Coarse-grained scheduling. The S-Worker in each machine executes multiple DAGs in parallel to increase throughput and reduce the end-to-end GNN training time. The first question we need to answer is how many DAGs should be launched in a machine. If we launch too many DAGs, which means more resource is needed by sampling, then resource contention becomes a problem. Resource contention does not just occur among the DAGs, but also between sampling (i.e., DAG execution) and training (i.e., model computation). The training time increases significantly when too many DAGs are launched. On the other hand, if too few DAGs are running, the resource is under-utilized. The training phase finishes quickly and the next iteration's training waits for the neighborhood subgraphs to be produced by the DAGs.

To control the resource utilization, we need to decide when to launch a DAG. We can model this problem as a variation of the classical Job-Shop Scheduling Problem (JSP) [7]. Each DAG can be regarded as a job, where a set of operations (tasks) in each job need to be processed in a specific order, and we have a set of jobs that are to be processed on a given set of workers. Knowing the best timing for DAG launching is equivalent to getting the earliest starting time of each job in the solution to this special Job-Shop Scheduling Problem. The Job-Shop Scheduling Problem has been well studied and to find a schedule that minimizes the makespan or minimizes the sum of the job completion time was

Algorithm 1: The Coarse-Grained Scheduling Strategy

```
Variable: $C_{util}$, $Q_{size}$, $T_{gap}$
Given: $\sigma$=launch-score
while more_dag do
    //more_dag=1 when more DAGs can be launched
    $balance = \frac{T_{avg\_sample}}{T_{avg\_train} * Q_{size}}$;
    $f(C_{util}) = (101 - e^{C_{util}/c})$, where $c = \frac{100}{\ln_{101}}$;
    launch-score = $T_{gap} * f(C_{util}) * balance$;
    if launch-score $\geq \sigma$ then
        more_dag = Launch_DAG();
        // launches a new DAG; returns 0 when no more DAG to launch
        $T_{last\_launch} = Time()$ // used to calculate $T_{gap}$
    else
        sleep(5ms);
    end
end
```

proved to be strongly NP-hard [11]. Some new research also shows that the currently best approximation algorithms have worse than logarithmic performance guarantee [26].

We propose a heuristic strategy to decide when to launch a DAG based on three runtime measures: C_{util}, Q_{size}, and T_{gap}.

C_{util} is the CPU utilization rate. If C_{util} is low, we may launch a new DAG; otherwise, we may wait until C_{util} drops to a suitable level. Note that high C_{util} does not necessarily result in better performance because there could be much contention and switching among DAGs and between sampling and training.

In addition to CPU utilization, We also need to consider the memory footprint. The neighborhood subgraph constructed from each DAG execution is kept in the DAG output queue in the S-Worker and Q_{size} is the size of this queue. The neighborhood subgraphs are then consumed by the T-Worker for training. Thus, Q_{size} is essentially an indicator of the speed of production (by the S-Worker) and the speed of consumption (by the T-Worker) of the neighborhood subgraphs. If Q_{size} is small, we may launch new DAGs; otherwise, we pause the launching. If Q_{size} is large, it implies an over-supply of neighborhood subgraphs and we may shift more computing resource from sampling to accelerate training. Thus, Q_{size} not only controls the memory usage, but also balances the overall resource usage between sampling and training.

We also found that the real-time measure for C_{util} is not sensitive enough since newly launched DAGs may not change the CPU utilization in a short time period and many DAGs may be launched during the period. Later, when the tasks in these DAGs start to run in parallel and use up the computing resource, the system suffers from severe resource contention. To avoid such delayed performance punishments, we introduce T_{gap}, which is the time gap elapsed since the previous DAG launch. If T_{gap} is too small, we may want to wait for a bit longer before we launch a new DAG.

It would be undesirable if users need to set the thresholds for the three measures, as it is hard to determine what values of C_{util}, Q_{size}, and T_{gap} are good and how to relate them to each other. To this end, we integrate them into one single score, *launch-score*, to decide whether we should launch a new DAG. The idea is to maintain the balance between the production speed and the consumption speed of neighborhood subgraphs, while keeping CPU utilization high. Ideally, we hope that the output of each DAG will be consumed immediately by the training phase, which means that Q_{size} should be close to 0 all the time. However, in most of the cases a very low Q_{size} happens with a very low C_{util}. Thus, we need to consider Q_{size} together with C_{util}.

Algorithm 1 shows the algorithm for coarse-grained scheduling. First, we want to maintain $balance = \frac{T_{avg\_sample}}{T_{avg\_train} * Q_{size}} = 1$, where T_{avg_sample} and T_{avg_train} are the average time for sampling and training a mini-batch. If $balance > 1$, it means that it would take less time to consume the current Q_{size} sampling results than to produce a new sampling result, which is an indicator that a new DAG should be launched. Next, we first attempt to use $(100 - C_{util})$ to give a higher weight to *balance* if C_{util} is low and penalize *balance* (i.e., delay new DAG launching) when C_{util} is high. However, simply using $(100 - C_{util})$ does not work well as it is a linear scale. Instead, we want to quickly increase CPU utilization when C_{util} is low and prevent contention promptly when C_{util} is already very high. Thus, we use an exponential function, $f(C_{util}) = 101 - e^{C_{util}/c}$, where $c = \frac{100}{ln101}$ is a constant used to align the range of $f(C_{util})$ with that of C_{util}, i.e., $f(0) = 100$, $f(100) = 0$, and $0 \leq f(C_{util}) \leq 100$. Finally, we also put T_{gap} as a weight to reflect the delay in the real-time measurement of C_{util}, which leads to the definition of *launch-score* in Algorithm 1.

We monitor *launch-score* in real time and launch a new DAG when *launch-score* $\geq \sigma$, where σ is a threshold set as follows. As shown in Algorithm 1 and explained above, *launch-score* connects $balance$, $f(C_{util})$ and T_{gap} together to determine whether a new DAG job should be launched. In practice, there exist reasonable values of $balance$, $f(C_{util})$ and T_{gap} for which a new DAG should be launched; Note that there are always trade-offs between $balance$ and $f(C_{util})$, e.g., a higher $balance$ and a lower $f(C_{util})$, to achieve a high *launch-score*. Such tradeoffs in runtime allows the system to automatically adjust the resource allocation to balance the sampling and training progress.

Fine-grained scheduling. After new DAGs are launched, the S-Worker executes the tasks in the DAGs, in parallel with the tasks in other DAGs. These tasks are put in a queue when their dependency is cleared (i.e., their parent tasks in the DAG are completed) and are handled by a pool of processing threads. If we execute the tasks in an FIFO order, some tasks of newly launched DAGs could be in front of the tasks in those almost-finished DAGs. For example, when the DAG_1 pushes the "END" node in the task queue and there are already "NSC" tasks from DAG_2 and DAG_3 in the queue, the "NSC" tasks will be executed first and the "END" task will be processed later even although the "END" task is the last task in DAG_1, completing which will immediately return the sampled data to the T-Worker for training. Meanwhile, one task may unlock a lot of downstream tasks in the same DAG, and heavy tasks may block many light tasks. Thus, the average completion time of the DAG jobs and hence the end-to-end GNN training time can be significantly increased.

We schedule tasks according to the following orders: (1) tasks in a DAG with a smaller ID will be executed first; (2) tasks in the same DAG will be executed in ascending order of their costs. We assign a smaller ID to a DAG launched earlier to prioritize earlier DAGs to be

Algorithm 2: Block Assignment

```
Input: List of Blocks $\mathbf{B} = B_1, B_2, ....., B_n$
Output: Graph partitions $P_1, P_2, P_3, ......, P_k$
for each block $B_i$ in $\mathbf{B}$ do
    for $j \leftarrow 1$ to $k$ do
        CE[j] = $|\text{Cross\_Edge}(P_j, B_i)| / |P_j|$
        BS[j] = $(1 - \alpha * \frac{|P_j(train)|}{C(train)} - \beta * \frac{|P_j(val)|}{C(val)} - \gamma * \frac{|P_j(test)|}{C(test)})$
    end
    $x = \mathbf{argmax}_{1 \le t \le k}$ {CE[t] * BS[t]}
    $P_x = P_x \cup B_i$
end
return $P_1, P_2, P_3, ......, P_k$
```

completed first. We calculate the cost of a task by the data it needs to handle. For example, for sampling tasks in each hop of NSC, the cost is equal to the total number of neighbors of the input vertices; for Feature Fetching, the cost is the number of vertices/edges to be fetched multiplied by the vertex/edge feature dimension. As tasks may require data from remote machines, the S-Worker sends data fetching requests to the local Graph Store, which communicates with remote Graph Stores to fetch the data. The remote requests are also scheduled in a similar way and the network operations are processed concurrently with the CPU operations.

3.3 GNN-based Graph Partitioning

Existing graph partitioning algorithms [37, 41, 46] are mainly designed to reduce inter-partition edges and balance the workload. They have been widely adopted in distributed graph processing systems [27, 53, 81] to reduce inter-machine communication. However, sample-based GNN training focuses on the K-hop neighborhood of only the vertices in the *training*, *validation* and *test* sets (instead of all vertices). For example, in Figure 7, traditional partitioning strategies cut the graph into two parts by the left dotted line since it not only balances the vertices but also has the least cut edge. But for a 2-layer GNN training, since vertex A and vertex B are the labeled vertices, partitioning by the right dotted line is actually a better choice. Even if this results in two cut edges, it would not cause any data movement in the training process as only the 2-hop neighbors of the labeled vertices are required.

In addition, the ratio of the sizes of the training, validation, and test sets of different real-world graphs may differ significantly. For example, in the Ogbn-Product dataset, the test set size is 11 times the training set size and 56 times the validation set size; while in the Ogbn-Papers dataset, the test set size is only 0.18 times the training set size and 1.7 times the validation set size. Thus, the partitioning algorithm should consider both the special data access pattern of k-layer GNN training and the balanced distribution of the training, validation, and test sets.

It is known that the traditional graph partitioning problem is proved to be APX-hard [6]. Thus, our graph partitioning problem is also APX-hard as it can be reduced to the traditional graph partitioning problem. We propose a heuristic two-step graph partitioning strategy tailored for GNN sampling workloads. The main idea is to group vertices into multi-hop neighborhood-based blocks and

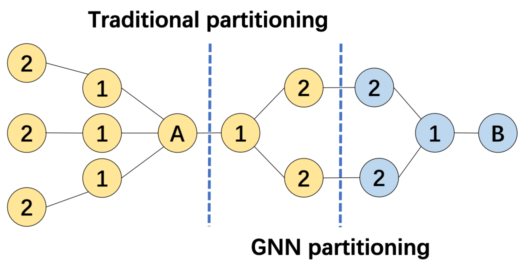


Figure 7: Traditional partitioning vs. GNN partitioning

then assign these blocks to partitions by balancing the numbers of *training*, *validation* and *test* vertices in the partitions.

Step (1) neighborhood block construction. To better preserve the locality of graph data for GNN sampling workloads, we construct a neighborhood block for each vertex in the training, validation and test sets. We start a K-hop breadth-first search from each vertex v (v is called the **block center**) and broadcast the unique block ID of v to its K-hop neighbors being visited. Every vertex only keeps the first block ID it receives, except for block centers which keep their own block ID. A block is then formed of all the vertices that keep the same block ID. Figure 7 demonstrates how to construct the blocks.

Step (2) block assignment. Just as existing graph partitioning algorithms aim to balance the number of vertices in the partitions, our objective is to also balance the number of training, validation and test vertices in the partitions so that the work of training, validation and test is also balanced among the machines. Algorithm 2 shows how to assign the blocks. For each block B_i, it is assigned to the partition with the highest score. P_j is the set of vertices that have already been assigned to partition j. $CE[j]$ is the number of cross-edges between B_i and P_j, which will be eliminated if B_i is assigned to P_j. Thus, the larger $CE[j]$ is, the more likely B_i is assigned to P_j. As the size of different partitions may vary during the assignment, we normalize $CE[j]$ by $|P_j|$. $BS[j]$ is the balancing score that controls the number of training/validation/test vertices in partition j to be close to the average value. For example, the expected number of training vertices in each partition is $C(train) = |V(train)|/N$, where $V(train)$ is the set of all training vertices and N is the total number of partitions. Let $P_j(train)$ be the set of training vertices currently in partition j. Thus, a smaller $\frac{|P_j(train)|}{C(train)}$ means that more training vertices can be assigned to partition j. The above applies to the validation and test vertices as well. In addition, we also use a weight to put more attention on a specific type of vertices according to the scale of that type in order to obtain a better overall performance. For example, if the number of training vertices is significantly more, we may set a larger α to favor the training process, which can improve the end-to-end processing time.

Before the block assignment, we sort the blocks in descending order of $max\{|V(train)|, |V(val)|, |V(test)|\}$. Then, we start the block assignment according to this order. In this way, larger blocks are assigned to different partitions first, so that smaller blocks may be used later to fill the partitions more easily when the partitions begin to fill up.

4 SYSTEM IMPLEMENTATION

We implemented ByteGNN based on GraphLearn [4], using TensorFlow [5] as the backend deep learning framework for the training phase. We used the data loader and distributed graph storage in GraphLearn, where the graph topology data is stored in adjacency list format and the features are stored separately and indexed by their vertex/edge ID. Our implementation focuses on efficient DAG construction and execution, graph partitioning, and gradient synchronization.

DAG construction and execution. We adopt the Gremlin syntax to help us construct the DAG. We redesigned the parsing method to encode necessary metadata from a Gremlin query for generating DAG nodes. Since one Gremlin statement may become several nodes in the final DAG, we implemented the parsing phase to carefully handle the complex dependency among the task nodes. We also changed all the communication methods from synchronous in GraphLearn to asynchronous in ByteGNN.

Graph partitioning. We implemented our graph partitioning strategy on the streaming graph partitioning framework in [12, 46]. The random start seed vertices in [12] were replaced with labeled vertices/edges. The framework first does the multi-source distributed BFS to build the K-hop neighborhood blocks, and then applies our block assignment strategy in Section 3.3 to assign blocks to the partitions. The partitions are written into HDFS and then loaded by the system for sampling and training.

Gradient synchronization. To address the potential convergence issue, we implemented the bulk synchronous parallel (BSP) and stale synchronous parallel (SSP) models based on the TensorFlow API, so that users may also choose to use BSP or SSP to obtain faster model convergence and reduce the training time.

5 SYSTEM EVALUATION

We evaluate the performance of ByteGNN by comparing with Graph-Learn [4], Euler [1] and Distributed DGL (DistDGL) [78]. We also examine the effects of our system designs on the performance.

Testbed. We ran our experiments on a cluster of machines where each machine is equipped with 1T DDR4 main memory and two 2.40GHz Intel(R) Xeon(R) Platinum 8260 CPU (each CPU has 24 cores or 48 virtual cores by hyper-threading). All the machines are connected by a 25Gbps network and the OS is the Debian 9.13 with Linux kernel 4.19.117.

Datasets. We used three datasets in the evaluation, as shown in Table 1. *Ogbn-Product* and *Obgn-Papers* are the largest two graphs in the Open Graph Benchmark (OGB). *Ogbn-Product* is an undirected and unweighted graph modeling an Amazon product co-purchasing network [32]. *Obgn-Papers* is a directed citation graph of 111 million papers indexed by MAG [65]. The *Social* dataset is a directed graph in industry from the social network scenario.

Table 1: Graph datasets

Dataset	Ogbn-Product (Product)	Ogbn-Papers (Papers)	Social
Vertices	2,449,029	111,059,956	66,351,656
Edges	123,718,280	1,615,685,872	1,751,915,191
Feature	100	128	150
Classes	47	172	2
Training set	196,615	1,207,179	6,631,989
Validation set	39,323	125,265	19,908,461
Test set	2,213,091	214,338	39,811,206

Models. We used three representative GNN models, Graph Convolutional Network (GCN) [43], GraphSAGE [31] and Graph Attention network(GAT) [64], in our evaluation. In order to demonstrate the expressiveness and efficiency of ByteGNN, we also tested the unsupervised variants of these three models. Although unsupervised learning shares most of the GNN architectures with supervised learning, it involves the negative sampling operator in the sampling phase and is also widely used in important tasks such as link prediction. Since many works are proposed to improve the sampling of GNN models, we used GraphSAINT [76] as a typical example to show how our sampling abstraction can be applied.

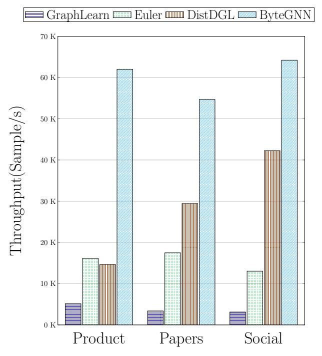
As shown by prior works [16, 32, 37], deeper and larger GNN architectures can achieve better model accuracy. We used three network layers for the models and set the sampling configuration to $k_1 = 10$, $k_2 = 5$ and $k_3 = 3$ for the neighborhood sampling models. The mini-batch size was set to 512 in all the experiments.

Systems. We compared with three distributed GNN training systems, GraphLearn, Euler (v1.0) and DistDGL (DGL v0.5.3). GraphLearn is a distributed framework designed for the development and application of GNNs on large scale graphs within Alibaba. Euler is also developed by Alibaba but it has been used in many companies for large scale GNN training. Both GraphLearn and Euler use TensorFlow as the backend system. DistDGL is a popular GNN system and its latest version (v0.5.3) supports distributed GNN training. The computational patterns of DistDGL are highly optimized by dedicated sparse tensor operations, which are currently lacking in ByteGNN as this work focuses on improving the sampling performance. Unless otherwise stated, we used the default configuration of these systems in our experiments. ByteGNN used the BSP model to obtain better test accuracy. All the systems adopt the random neighborhood sampling method as the default sampling method and use the same hop number and fanout.

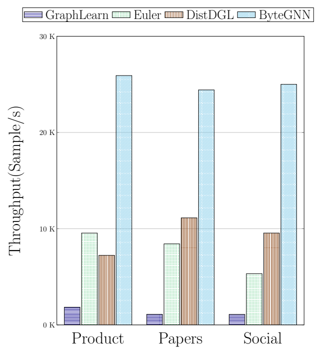
5.1 Overall Performance

We first compared the overall performance of the systems. We report the throughput of each system, i.e., the number of samples being processed per second, which is a metric commonly used to measure the performance of model training of a system. The throughput is calculated as the total number of seed vertices processed divided by the end-to-end GNN training time. Thus, the larger the throughput of a system, the shorter is the end-to-end GNN training time of the system. The hidden size is set to 32 in GCN and GraphSAGE. For GAT, we used 4 attention heads with hidden size 16. Since Euler failed to run unsupervised GAT training, we ignore this result.

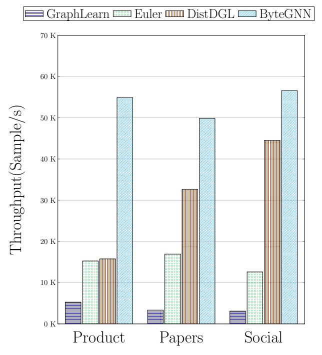
Figure 8 reports the results. ByteGNN achieves 7.5 to 16.2 times speedup compared with GraphLearn on supervised training and up to 23 times on unsupervised training. As ByteGNN is implemented on GraphLearn and the key differences from GraphLearn are the three system designs presented in Sections 3.1-3.3, the results show



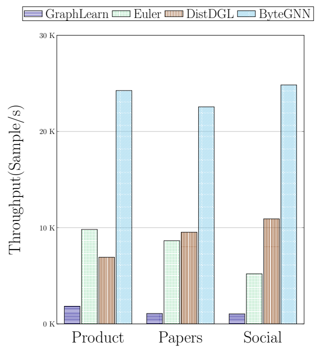
(a) GCN



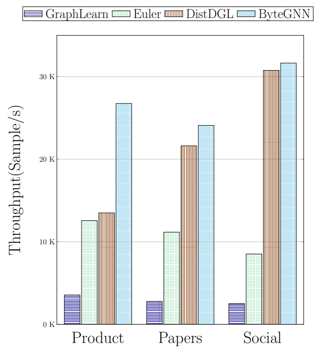
(b) Unsupervised GCN



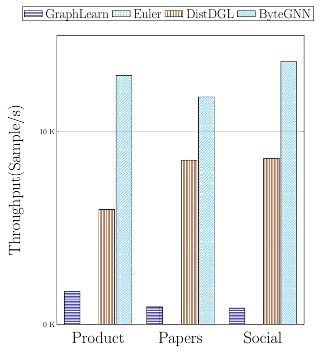
(c) GraphSAGE



(d) Unsupervised GraphSAGE



(e) GAT



(f) Unsupervised GAT

Figure 8: The throughput of GraphLearn, Euler, DistDGL, and ByteGNN for training different models on 4 machines

that our designs are effective. In particular, the performance improvement obtained by ByteGNN is more significant for unsupervised training that has more parallel sampling tasks, which is as a result of the high parallelism enabled for tasks within a DAG and among DAGs.

Compared with Euler, although Euler also adopts the data-flow graph by TensorFlow for sampling the mini-batch neighborhood and training, ByteGNN can still achieve up to 4.7 times performance speedup. Euler cannot run two TensorFlow's computation graphs at the same time as otherwise it would lead to a convergence problem. In contrast, the separation of sampling phase and training phase in ByteGNN enables concurrent execution of multiple DAGs to maximize CPU utilization.

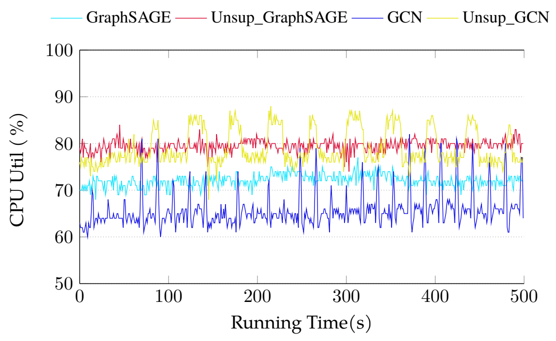


Figure 9: Average CPU utilization of ByteGNN

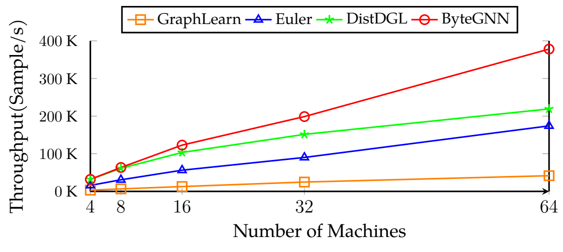
Compared with DistDGL, ByteGNN achieves 2.1 ~ 3.5 times speedup for training the dense graph *Ogbn-Product* in both supervised and unsupervised training. For the sparse graphs *Ogbn-Papers* and *Social*, ByteGNN still has better performance. In the supervised training, ByteGNN is 1.5x and 1.3x faster than DistDGL in GCN and GraphSAGE. But the speedup is less significant compared with that on the dense graph, especially for GAT. This is because sparse tensor operations in the training phase of DistDGL have been highly optimized, while currently there is no such optimization in ByteGNN. For unsupervised training that has heavier sampling workloads, Figures 8(b)&(d)&(f) show that ByteGNN achieves considerably better performance as ByteGNN's design enables higher parallelism in sampling execution. (e.g., 2.4x for unsupervised GraphSAGE and 1.6x for unsupervised GAT).

We also report the average CPU utilization of ByteGNN for training all the models in Figure 9. The result is reported for *Ogbn-Papers*, while ByteGNN's CPU utilization for the other two datasets is similar. Compared with the average CPU utilization of GraphLearn, Euler and DistDGL as shown in Figure 3, ByteGNN achieves 3 - 6 times higher CPU utilization. ByteGNN has lower CPU utilization for supervised GCN and GraphSAGE because the number of neighborhood subgraphs in the DAG output queue is sufficient, S-Worker dynamically frees up some resource to T-Worker and the training workload for GCN is not heavy.

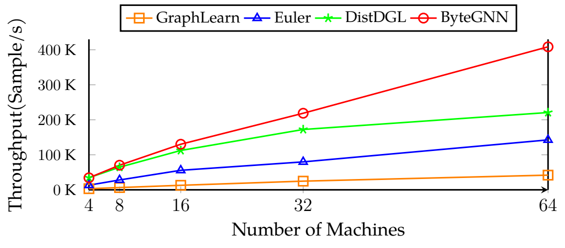
5.2 Scalability

Figure 10 reports the throughput scalability of the systems for the *Ogbn-Papers* dataset, where we increase the number of machines from 4 to 64. ByteGNN achieves better scalability than all the other three systems. We omit the results for the other two datasets due to the page limitation, but the patterns are similar and ByteGNN's performance on the dense graph *Ogbn-Product* is even better. The hidden size is set to 256 here to demonstrate the performance of our system in different configuration.

In general, the throughput performance in distributed GNN training has sub-linear scalability due to the synchronization overhead (when the BSP model is used to achieve high accuracy) and heavy network I/O among the machines. GraphLearn and Euler scale poorly and their throughputs are relatively low. Although GraphLearn and Euler are built on top of TensorFlow, the default asynchronous gradient update in distributed TensorFlow does not



(a) GraphSAGE



(b) GCN

Figure 10: Scalability comparison

cause much synchronization overhead (though with potential accuracy loss). However, without an effective graph partitioning algorithm to preserve the locality of neighborhood access, remote data fetching results in high network communication overhead. In contrast, DistDGL's main issue in scalability is due to the synchronization overhead for gradient update. If the sampling output of a mini-batch cannot return on time, the trainer will get the forward loss later and all the other machines will wait for this loss to begin the back propagation. Even with the fixed prefetching mechanism, the possibility of the back propagation waiting increases as the number of machines increases. In comparison, ByteGNN's scheduling allows the sampling outputs to be pipelined to the trainers while other sampling processes continue, which results in better resource utilization. ByteGNN's GNN-tailored graph partitioning algorithm also leads to lower network communication overhead as the number of machines increases. As a result, ByteGNN achieves better scalability than the other systems.

5.3 Model Accuracy

We also report the correctness of ByteGNN by evaluating the test accuracy of the GraphSAGE model on the *Ogbn-Product* dataset, comparing with GraphLearn and DistDGL. Euler has similar accuracy as GraphLearn. In Figure 11, we report the test accuracy of different systems at every epoch until the training converges. The result shows that the systems achieve similar or the same accuracy eventually, but ByteGNN converges the fastest, in both the single-machine setting (1M) and distributed 4-machine setting (4M). We also note that as the mini-batch training can update the model many times in one epoch, the accuracy increases quickly in the first several epochs. The single-machine accuracy of GraphLearn can also be seen as the baseline to demonstrate that our code changes to GraphLearn do not affect the semantics of the GNN models. And as ByteGNN uses BSP to ensure model convergence in distributed training, it achieves approximately the same accuracy as DistDGL but uses less time.

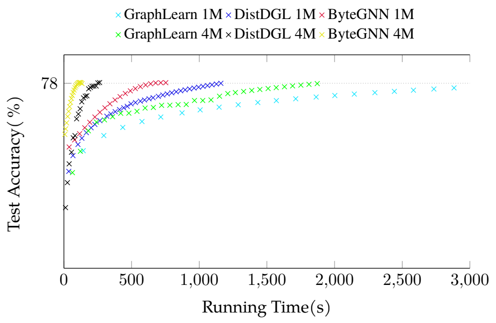


Figure 11: Accuracy comparison

5.4 Evaluation on System Designs

We further evaluate the effectiveness of each individual system design in ByteGNN.

5.4.1 *Sampling Abstraction.*

We used the GraphSAINT model to demonstrate how to build a DAG using our sampling abstraction. Different from sampling neighbors across the layers, GraphSAINT constructs mini-batches by sampling the whole input graph once and then building a full GCN on the sampled subgraph. It provides three light-weight and efficient samplers, NodeSampler, EdgeSampler, and RandomWalkSampler. Due to space constraints, we only show the implementation of the Seed Sampler function in our sampling abstraction for GraphSAINT's NodeSampler and EdgeSampler. Note that the training part is the same for different samplers.

```
// NodeSampler
def seed_sampler(self):
  return self.g.V(node_type="train")
                  .batch(batch_size = n)
                  .by("InDegree")

// EdgeSampler
def seed_sampler(self):
  return self.g.E(edge_type="train")
                  .batch(batch_size = m)
                  .by("EdgeWeight").bothV()
```

For GraphSAINT's NodeSampler, we sample n vertices from all the training vertices according to a vertex probability distribution $P(u) \propto ||\tilde{A}_{:,u}||^2$. We call this "InDegree" sampling as it is associated with the in-degree of each vertex. For EdgeSampler, the edge probability distribution follows $p_{e(v,u)} \propto \frac{1}{deg(u)} + \frac{1}{deg(v)}$. Normally, it can be pre-calculated dependent on the graph topology only and become the weight of edges. The code above shows the Seed Sampler function of these two samplers using our sampling abstraction. Using Gremlin, users can easily write the sampling logic. Then, the sampling stage can be completed by the NSC and Feature Fetching functions as discussed in Section 3.1.

We also implemented the GraphSAINT model in GraphLearn to compare the end-to-end training performance. Table 2 reports the speedup ratio of ByteGNN over GraphLearn for training GraphSAINT on different graphs using four machines, using the same sampling setting from [76]. Even though GraphSAINT has a light sampling workload, ByteGNN can still achieve significant speedup compared with GraphLearn. Note that ByteGNN has better performance with EdgeSampler because EdgeSampler needs to obtain the two end-vertices of the sampled edge and has a higher workload than NodeSampler.

Table 2: Speedup ratio of ByteGNN over GraphLearn

Type	Ogbn-Product	Ogbn-Papers	Social
NodeSampler	3.40	2.80	4.05
EdgeSampler	4.72	3.25	5.89

Table 3: The execution time (sec) of one epoch for different sampling settings, running on *Ogbn-Papers* using 8 machines

(a) The execution time of light sampling workload

	Sequential	Fixed DAGs	Coarse-Grained
512	79.04	19.56	18.62
1024	75.29	19.21	17.52
2048	74.52	19.90	17.75

(b) The execution time of heavy sampling workload

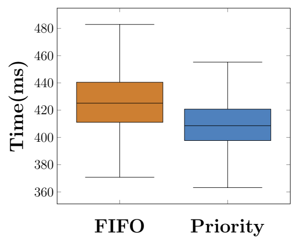
	Sequential	Fixed DAGs	Coarse-Grained
512	314.04	63.41	56.70
1024	312.14	68.72	57.20
2048	310.43	78.10	62.46

5.4.2 Coarse-Grained Scheduling. We first evaluate the performance of the coarse-grained scheduling strategy. We used three different batch sizes: 512, 1024 and 2048. We created a light sampling workload by setting the sampling configuration to $k_1 = 10$, $k_2 = 5$ and $k_3 = 3$. We also created a heavy sampling workload by setting the sampling configuration to $k_1 = 15$, $k_2 = 10$ and $k_3 = 5$.

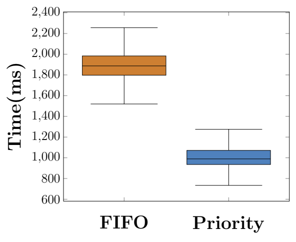
We used two baselines. The first baseline is sequential DAG execution, which runs DAGs one after another. The second baseline is running a fixed number of DAGs at any time. The DAG size is set to 16 which is the same as the default in DistDGL prefetching.

Table 3 shows that coarse-grained scheduling achieves the best performance in all cases. For sequential DAG execution, the execution of a single DAG at a time results in resource under-utilization and thus has poor performance. For the light sampling workload, the fixed number of DAGs has performance close to that of coarse-grained scheduling. This is because the sampling workload is light and can be finished quickly so that DAGs can already produce the sampling results fast enough for the trainer to consume. However, the lower sampling rate leads to more biased results and the light workload also results in resource under-utilization. When the sampling workload is heavier, the higher random data access overhead and higher network I/O cost to retrieve remote neighbors become the performance bottleneck. In this case, our coarse-grained scheduling strategy becomes effective as it dynamically adjusts the resource allocation to sampling and training in order to maximize resource utilization.

5.4.3 Fine-Grained Scheduling. We further show the impact of the fine-grained scheduling strategy on the DAG completion time. We ran ByteGNN for 10 epochs and measured the completion time of each DAG of mini-batch sampling under two settings: using the priority-based scheduling in the fine-grained scheduling strategy



(a) Supervised Training



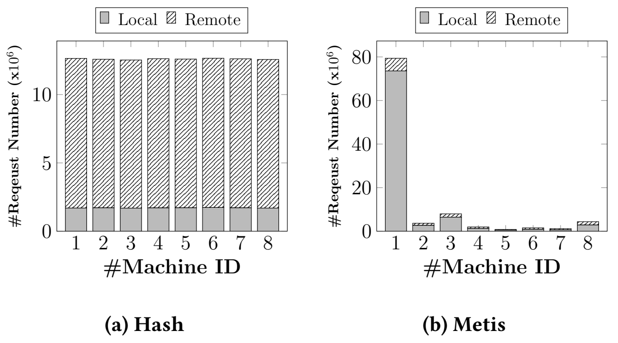
(b) Unsupervised Training

Figure 12: Distribution of DAG completion time

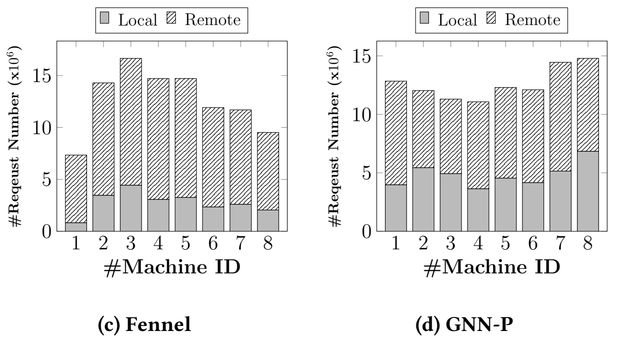
and using FIFO-based scheduling. We use the box plot to report the distribution of the DAG completion time. Figure 12 shows that the priority-based scheduling can significantly reduce the completion time of the DAGs, and the DAG completion time is also more stable, which avoids the short time period of under-supply or over-supply of the sampling outputs. In supervised training, when the number of DAGs is in the suitable range, there are not too many small tasks in the DAGs so that the FIFO scheduling can handle it. However, unsupervised training launches more sampling tasks during the DAG execution. The median DAG completion time of the FIFO scheduling is almost two times greater than the median of the priority-based scheduling.

5.4.4 Graph Partitioning. To validate the effectiveness of our GNN-tailored graph partitioning (**GNN-P**) algorithm, we compared GNN-P with three well-known graph partitioning methods: hash partitioning, Fennel partitioning [62] and METIS partitioning [41]. Both hash and METIS partitioning have been widely adopted in distributed graph computing systems. Fennel is the representative of one-pass streaming partitioning algorithms.

Figure 13 reports the distribution of the requests for remote and local neighborhood data in one training epoch by each machine (the distributions of the requests for validation and test show a similar pattern). First, Figure 13(a) shows that hash partitioning achieves the best balanced distribution because hash partitioning assigns each type of vertices to different partitions with the same possibility. However, it does not consider the locality of neighborhood data access and thus incurs much higher remote data requests, which result in high network communication overhead. The number of remote requests is about 6.32 times the local data requests. Although METIS keeps the total number of vertices similar in each partition, the number of training vertices varies significantly among the partitions (also true for validation and test vertices). Half of the training vertices are assigned to one partition in Machine 1, which indeed reduces remote data requests; however, the imbalanced distribution results in Machine 1 being a severe straggler, which processes around 80% of the data requests in each training epoch. Fennel roughly balances the total load in each partition. Fennel considers data locality but it is only limited to direct neighbors, and thus remote data requests still take a major portion of the total number of data requests in each partition. In contrast, the multi-hop block construction of GNN-P significantly improves the data locality of each partition. The ratio of remote data requests and local data requests in the partitions of GNN-P is also considerably



(a) Hash **(b) Metis**



(c) Fennel **(d) GNN-P**

Figure 13: The distribution of remote/local data requests

smaller than that of the hash and Fennel partitions. In addition, with the balance-aware assignment algorithm, GNN-P achieves a much more balanced distribution of the total workload.

6 RELATED WORK

Single-machine GNN systems. PyG [21] integrates with PyTorch [54] to provide a message-passing API for GNN training. Incorporated with the Apache TVM compiler [15], FeatGraph [33] generates optimized kernels for both CPU and GPU. But to implement new GNN operators, users need to have the background of TVM primitives. NeuGraph [48] proposes Scatter-ApplyEdge-Gather-ApplyVertex programming model to express GNN models and supports full-batch training in a single machine with multiple GPUs. It divides a graph into 2-D chunks and introduces a streaming scheduler to handle the CPU-GPU data transfer when GNN computation for a graph cannot fit in the GPU memory. Seastar [69, 70] proposes a vertex-centric programming model to express GNN models using native Python syntax and identifies a common seastar computation pattern in GNN training to generate high-performance fused kernels. There are also works [36, 46] that focus on addressing the bottleneck of mini-batch sampling. PaGraph [46] is a sampling-based training framework on multi-GPUs that addresses the expensive subgraph data loading issue by a GNN computation-aware cache policy. NextDoor [36] enables users to express the sampling tasks in GPUs by a high-level API and also proposes a novel *transit parallelism* approach to parallelize graph sampling. However, these single-machine systems have the limitation of processing large industrial graphs due to limited GPU memory.

Distributed GNN systems. For training GNNs on large graphs in a distributed manner, AliGraph [80] provides sampling-based distributed GNN training and reduces network communication by caching vertices on local machines. DistDGL [78] uses a distributed in-memory key-value store to support efficient access to graph topology and feature data in distributed GNN training. DGCL [10] proposes an efficient communication library for distributed full-batch GNN training on multi-GPUs using NVLink. DGCL needs to load all the graph data into GPUs first and is not suitable for training large graphs. Based on FlexFlow [38], a distributed DNN training framework, Roc [37] also adopts full-batch training in multi-GPUs using dynamic programming to minimize data swapping between host DRAM and GPU memory. P3 [23] reduces communication by model parallelism for the first layer, while it uses data parallelism for the remaining layers. However, if the hidden size is larger than the input dimension, it still incurs a high cost for the synchronization of the output of the first layer. AGL [77] uses MapReduce to preprocess a graph, which samples multiple neighborhood subgraphs for each vertex and stores them in a distributed file system. During training, AGL loads the required samples of neighborhood subgraphs of the vertices directly from the disk. However, the preprocessing cost is high and the storage overhead can also be large. Dorylus [61] designs a computation separation mechanism and pipelines the different computation patterns in the Amazon EC2 machine and serverless Lambdas in the cloud environment.

Graph partitioning in GNN. METIS [41] is commonly used for graph partitioning in GNN algorithms [16, 44, 45] and systems [10, 48, 78]. Cluster-GCN [16] adopts METIS to build small clusters and then uses the partitions to perform an SGD update. DistDGL [78] adjusts the METIS algorithm to balance the training vertices in each partition. NeuGraph [48] uses the Kernighan-Lin algorithm to make the chunks in the diagonal have as many edges as possible. Roc [37] uses an linear-regression based algorithm to achieve balanced partitioning for both GNN training and inference; but it still treats all the vertices equally, which makes the computation load unbalanced. PaGraph [46] partitions a graph based on the neighborhood of a training vertex. However, with the multi-hop feature cache to avoid feature communication between different trainers, the memory overhead is too high.

7 CONCLUSIONS

We presented ByteGNN, a distributed GNN training system for GNN training in large graphs. ByteGNN abstracts the sampling phase of a mini-batch as a DAG of small tasks to support high parallelism. Leveraging the DAG abstraction, ByteGNN designs a two-level scheduling to improve resource utilization and reduce the end-to-end GNN training time. ByteGNN also tailors graph partitioning for GNN workloads to reduce network I/O and balance the workload. Experimental results show that ByteGNN can significantly shorten the end-to-end training time compared with existing distributed GNN systems.

ACKNOWLEDGMENTS

We thank the anonymous VLDB reviewers, for their constructive comments and suggestions that have helped greatly improve the quality of the paper.

REFERENCES

[1] 2019. Euler. https://github.com/alibaba/euler.

[2] 2019. Gremlin. http://tinkerpop.apache.org/gremlin.html.

[3] 2019. TinkerPop. http://tinkerpop.apache.org/.

[4] 2020. GraphLearn. https://github.com/alibaba/graph-learn.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[6] Konstantin Andreev and Harald Räcke. 2004. Balanced graph partitioning. In *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*. ACM, 120–124. https://doi.org/10.1145/1007912.1007931

[7] David L. Applegate and William J. Cook. 1991. A Computational Study of the Job-Shop Scheduling Problem. *INFORMS J. Comput.* 3, 2 (1991), 149–156. https://doi.org/10.1287/ijoc.3.2.149

[8] Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara* 11 (2011).

[9] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology* 25, 2 (2001), 163–177. https://doi.org/10.1080/0022250X.2001.9990249 arXiv:https://doi.org/10.1080/0022250X.2001.9990249

[10] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: an efficient communication library for distributed GNN training. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. ACM, 130–144. https://doi.org/10.1145/3447786.3456233

[11] Bo Chen, Chris Potts, and Gerhard Woeginger. 1998. *A Review of Machine Scheduling: Complexity, Algorithms and Approximability*. 21–169. https://doi.org/10.1007/978-1-4613-0303-9_25

[12] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. ACM, 32:1–32:12. https://doi.org/10.1145/3190508.3190545

[13] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rytstxWAW

[14] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research)*, Vol. 80. PMLR, 941–949. http://proceedings.mlr.press/v80/chen18p.html

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[16] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*. ACM, 257–266. https://doi.org/10.1145/3292500.3330925

[17] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*. ACM, 153–167.

[18] Andrew A. Davidson, Sean Baxter, Michael Garland, and John D. Owens. 2014. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*. IEEE Computer Society, 349–359. https://doi.org/10.1109/IPDPS.2014.45

[19] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 48. ACM, 77–88.

[20] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 42. ACM, 127–144.

[21] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. *CoRR* abs/1903.02428. arXiv:1903.02428 http://arxiv.org/abs/1903.02428

[22] Zhisong Fu, Bryan B. Thompson, and Michael Personick. 2014. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *Second International Workshop on Graph Data Management Experiences and Systems, GRADES 2014, co-loated with SIGMOD/PODS 2014, Snowbird, Utah, USA, June 22, 2014*. CWI/ACM, 2:1–2:6. https://doi.org/10.1145/2621934.2621936

[23] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 551–568. https://www.usenix.org/conference/osdi21/presentation/gandhi

[24] Thomas Gaudelet, Ben Day, Arian R. Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy B. R. Hayter, Richard Vickers, Charles Roberts, Jian Tang, David Roblin, Tom L. Blundell, Michael M. Bronstein, and Jake P. Taylor-King. 2020. Utilising Graph Machine Learning within Drug Discovery and Development. *CoRR* abs/2012.05716 (2020). arXiv:2012.05716 https://arxiv.org/abs/2012.05716

[25] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. 2016. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. 99–115.

[26] Leslie Ann Goldberg, Mike Paterson, Aravind Srinivasan, and Elizabeth Sweedyk. 1997. Better Approximation Guarantees for Job-shop Scheduling. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 5-7 January 1997, New Orleans, Louisiana, USA*. ACM/SIAM, 599–608. http://dl.acm.org/citation.cfm?id=314161.314395

[27] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. USENIX Association, 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[28] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2015. Multi-resource packing for cluster schedulers. In *Proceedings of the ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 455–466.

[29] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic scheduling in multi-resource clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. 65–80.

[30] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. 81–97.

[31] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 1024–1034. https://proceedings.neurips.cc/paper/2017/hash/5dd9db5e033da9c6fb5ba83c7a7ebea9-Abstract.html

[32] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. https://proceedings.neurips.cc/paper/2020/hash/fb60d411a5c5b72b2e7d3527cfc84fd0-Abstract.html

[33] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems. *CoRR* abs/2008.11359 (2020). arXiv:2008.11359 https://arxiv.org/abs/2008.11359

[34] Wen-bing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive Sampling Towards Fast Graph Representation Learning. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. 4563–4572. https://proceedings.neurips.cc/paper/2018/hash/01eee509ee2f68dc6014898c309e86bf-Abstract.html

[35] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. ACM, 261–276.

[36] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating graph sampling for graph machine learning using GPUs. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. ACM, 311–326. https://doi.org/10.1145/3447786.3456244

[37] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org. https://proceedings.mlsys.org/book/300.pdf

[38] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org. https://proceedings.mlsys.org/book/265.pdf

[39] Tatiana Jin, Zhenkun Cai, Boyang Li, Chengguang Zheng, Guanxian Jiang, and James Cheng. 2020. Improving resource utilization by timely fine-grained scheduling. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. ACM, 20:1–20:16. https://doi.org/10.1145/3342195.3387551

[40] Prajakta Kalmegh and Shivnath Babu. 2019. MIFO: A Query-Semantic Aware Resource Allocation Policy. In *Proceedings of the 2019 ACM International Conference on Management of Data*. ACM, 1678–1695.

[41] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (1998), 359–392. https://doi.org/10.1137/S1064827595287997

[42] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*. ACM, 239–252. https://doi.org/10.1145/2600212.2600227

[43] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SJU4ayYgl

[44] Ming Li, Zheng Ma, Yu Guang Wang, and Xiaosheng Zhuang. 2020. Fast Haar Transforms for Graph Neural Networks. *Neural Networks* 128 (2020), 188–198. https://doi.org/10.1016/j.neunet.2020.04.028

[45] Zhiheng Li, Geemi P. Wellawatte, Maghesree Chakraborty, Heta A. Gandhi, Chenliang Xu, and Andrew D. White. 2020. Graph Neural Network Based Coarse-Grained Mapping Prediction. *CoRR* abs/2007.04921 (2020). arXiv:2007.04921 https://arxiv.org/abs/2007.04921

[46] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN training on large graphs via computation-aware caching. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*. ACM, 401–415. https://doi.org/10.1145/3419111.3421281

[47] Libin Liu and Hong Xu. 2018. Elasecutor: Elastic Executor Scheduling in Data Analytics Systems. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 107–120.

[48] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 443–458. https://www.usenix.org/conference/atc19/presentation/ma

[49] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*. 135–146. https://doi.org/10.1145/1807167.1807184

[50] Kenneth Marino, Ruslan Salakhutdinov, and Abhinav Gupta. 2017. The More You Know: Using Knowledge Graphs for Image Classification. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 20–28. https://doi.org/10.1109/CVPR.2017.10

[51] Duane Merrill, Michael Garland, and Andrew S. Grimshaw. 2012. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*. ACM, 117–128. https://doi.org/10.1145/2145816.2145832

[52] Federico Monti, Michael M. Bronstein, and Xavier Bresson. 2017. Geometric Matrix Completion with Recurrent Multi-Graph Neural Networks. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*. 3697–3707. http://papers.nips.cc/paper/6960-geometric-matrix-completion-with-recurrent-multi-graph-neural-networks

[53] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu. 2017. Do We Need Specialized Graph Databases? Benchmarking Real-Time Social Networking Applications. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*. ACM, 12:1–12:7. https://doi.org/10.1145/3078447.3078459

[54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[55] Hao Peng, Jianxin Li, Yu He, Yaopeng Liu, Mengjiao Bao, Lihong Wang, Yangqiu Song, and Qiang Yang. 2018. Large-Scale Hierarchical Text Classification with Recursively Regularized Deep Graph-CNN. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*. ACM, 1063–1072. https://doi.org/10.1145/3178876.3186005

[56] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: online learning of social representations. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*. ACM, 701–710. https://doi.org/10.1145/2623330.2623732

[57] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: workload autoscaling at Google. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. ACM, 16:1–16:16. https://doi.org/10.1145/3342195.3387524

[58] Victor Garcia Satorras and Joan Bruna Estrach. 2018. Few-Shot Learning with Graph Neural Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=BJj6qGbRW

[59] Marco Serafini. 2021. Scalable Graph Neural Network Training: The Case for Sampling. *ACM SIGOPS Oper. Syst. Rev.* 55, 1 (2021), 68–76. https://doi.org/10.1145/3469379.3469387

[60] Xiaoyang Sun, Chunming Hu, Renyu Yang, Peter Garraghan, Tianyu Wo, Jie Xu, Jianyong Zhu, and Chao Li. 2018. ROSE: Cluster Resource Scheduling via Speculative Over-Subscription. In *2018 IEEE 38th International Conference on Distributed Computing Systems*. IEEE, 949–960.

[61] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 495–514. https://www.usenix.org/conference/osdi21/presentation/thorpe

[62] Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. FENNEL: streaming graph partitioning for massive scale graphs. In *Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014, New York, NY, USA, February 24-28, 2014*. ACM, 333–342. https://doi.org/10.1145/2556195.2556213

[63] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the 11th European Conference on Computer Systems*. ACM, 35.

[64] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rJXMpikCZ

[65] Kuansan Wang, Zhihong Shen, Chiyuan Huang, Chieh-Han Wu, Yuxiao Dong, and Anshul Kanakia. 2020. Microsoft Academic Graph: When experts are not enough. *Quant. Sci. Stud.* 1, 1 (2020), 396–413. https://doi.org/10.1162/qss_a_00021

[66] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *CoRR* abs/1909.01315 (2019). arXiv:1909.01315 http://arxiv.org/abs/1909.01315

[67] Yangzihao Wang, Andrew A. Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: a high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*. ACM, 11:1–11:12. https://doi.org/10.1145/2851141.2851145

[68] Zhouxia Wang, Tianshui Chen, Jimmy S. J. Ren, Weihao Yu, Hui Cheng, and Liang Lin. 2018. Deep Reasoning with Knowledge Graph for Social Relationship Understanding. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. ijcai.org, 1021–1028. https://doi.org/10.24963/ijcai.2018/142

[69] Yidi Wu, Yuntao Gui, Tatiana Jin, James Cheng, Xiao Yan, Peiqi Yin, Yufei Cai, Bo Tang, and Fan Yu. 2021. Vertex-Centric Visual Programming for Graph Neural Networks. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2803–2807. https://doi.org/10.1145/3448016.3452770

[70] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chengguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: vertex-centric programming for graph neural networks. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. ACM, 359–375. https://doi.org/10.1145/3447786.3456247

[71] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=ryGs6iA5Km

[72] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. *Proc. VLDB*

Endow. 7, 14 (2014), 1981–1992. https://doi.org/10.14778/2733085.2733103

[73] Liang Yao, Chengsheng Mao, and Yuan Luo. 2019. Graph Convolutional Networks for Text Classification. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019.* AAAI Press, 7370–7377. https://doi.org/10.1609/aaai.v33i01.33017370

[74] Yi Yao, Han Gao, Jiayin Wang, Ningfang Mi, and Bo Sheng. 2016. OpERA: opportunistic and efficient resource allocation in Hadoop YARN by harnessing idle resources. In *Proceedings of the 25th International Conference on Computer Communication and Networks.* IEEE, 1–9.

[75] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018.* ACM, 974–983. https://doi.org/10.1145/3219819.3219890

[76] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020.* OpenReview.net. https://openreview.net/forum?id=BJe8pkHFwS

[77] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. 2020. AGL: A Scalable System for Industrial-purpose Graph Machine Learning. *Proc. VLDB Endow.* 13, 12 (2020), 3125–3137. https://doi.org/10.14778/3415478.3415539

[78] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. *CoRR* abs/2010.05337 (2020). arXiv:2010.05337 https://arxiv.org/abs/2010.05337

[79] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Trans. Parallel Distributed Syst.* 25, 6 (2014), 1543–1552. https://doi.org/10.1109/TPDS.2013.111

[80] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *Proc. VLDB Endow.* 12, 12 (2019), 2094–2105. https://doi.org/10.14778/3352063.3352127

[81] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.* USENIX Association, 301–316. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu