Scalable Graph Neural Network Training: The Case for Sampling

Marco Serafini, Hui Guan
University of Massachusetts Amherst
United States

Abstract

Graph Neural Networks (GNNs) are a new and increasingly popular family of deep neural network architectures to perform learning on graphs. Training them efficiently is challenging due to the irregular nature of graph data. The problem becomes even more challenging when scaling to large graphs that exceed the capacity of single devices. Standard approaches to distributed DNN training, such as data and model parallelism, do not directly apply to GNNs. Instead, two different approaches have emerged in the literature: *whole-graph* and *sample-based* training.

In this paper, we review and compare the two approaches. Scalability is challenging with both approaches, but we make a case that research should focus on sample-based training since it is a more promising approach. Finally, we review recent systems supporting sample-based training.

1 Introduction

Many datasets are relational in nature: they are best represented as entities connected by relationships rather than as a single uniform dataset or table. Graphs are a universal formalism to model relational data. They are also commonly used to integrate data coming from multiple and possibly diverse data sources, such as relational databases, social networks, financial transaction logs, and many others. Graph analysis can leverage relations to get a deeper insight into data.

Machine learning and deep learning are being increasingly used for graph analytics. Graph Neural Networks (GNNs), in particular, represent a new family of Deep Neural Network (DNN) architectures tailored for graphs, where the structure of the neural network overlaps with the structure of the graph itself [35, 40]. A GNN is composed of several layers and each layer transforms input features to output features. The output features from a GNN are usually referred to as *embeddings* and used for downstream tasks. They are the state-of-the-art approach for several prediction and classification tasks on graphs.

Training GNNs presents unique challenges due to the irregularity of graph data. The input to a GNN is not a set of independent data items but a graph consisting of interconnected and inter-dependent vertices. Each layer in GNN is thus modeled as a message-passing process whereby each vertex aggregates the features of its neighbors [12, 34]. This results in an irregular and sparse computation, which is hard to perform efficiently. The problem becomes even more challenging when one wants to scale training to large graphs and multiple devices. Partitioning the graph inevitably splits some neighboring vertices across different partitions, leading to significant communication overhead.

Scaling GNN training is an open research question. Frameworks designed for GNN training such as DGL [34] and PyG [12] translate message-passing specifications of GNNs into those of DNN models, which are then run by existing DNN frameworks (e.g., TensorFlow [1] or PyTorch [30]). These DNN frameworks, however, are not specifically designed to scale GNNs to large input graphs. Recent work has proposed dedicated techniques to scale GNN training that fall into one of two approaches: *whole-graph* and *sample-based* training. In whole-graph training, message passing among vertices is performed on the entire graph. Sample-based training first samples the graph to obtain mini-batches, then maps each mini-batch to one device, and finally performs training on each mini-batch independently.

In this paper, we describe and compare the two approaches from a system scalability perspective. We discuss how scaling is challenging under both models, but argue that sample-based training is a more promising approach. Whole-graph training introduces inherent coordination and communication overheads that are hard to overcome as the system scales. Scaling sample-based training, on the other hand, requires (a) sampling algorithms that can form mini-batches without incurring into the "neighbor explosion" problem [14, 24] and (b) scalable systems to execute these sampling algorithms efficiently. We review recent research that addresses these requirements. Based on this review, we argue that sample-based training is a more promising approach to scaling GNN training.

2 Background

Graph Neural Networks GNNs perform *representation learning*: they take a graph as an input and map each vertex to a d-dimensional vector, known as an *embedding*. Embeddings are then used as inputs for downstream machine learning tasks, such as vertex classification and link prediction.

GNN frameworks allow expressing models using the message-passing paradigm [12, 34]. Each GNN layer can be expressed as a vertex-centric message-passing round. At the k-th layer, each vertex v aggregates the features $h_u^{(k-1)}$ of its neighbors $u \in N(v)$ and uses that information to compute its new feature $h_v^{(k)}$. This happen through three functions: a message function ϕ, a reduce function ρ, and an update function ψ.

Let $G = (V, E)$ be the input graph. The message function computes, for each edge e, a message from the source to the destination vertex given the features of the incident vertices and the edge:

$$m_e^{(k)} = \phi(h_u^{(k-1)}, h_v^{(k-1)}) \quad \forall e = (u, v) \in E$$

The message function can simply output the features of the source vertex $h_u^{(k-1)}$. Some algorithms assign features also to edges, and use those features as input to the message function.

The reduce function aggregates multiple messages sent to the same receiving vertex, for example by summing them or using an LSTM network:

$$a_v^{(k)} = \rho(\{m_e^{(k)} : e = (u, v) \in E\}) \quad \forall v \in V$$

Finally, the update function takes the result of message aggregation and computes a new feature vector by applying DNN operators such as fully-connected layers or convolutions:

$$h_v^{(k)} = \psi(h_v^{(k-1)}, a_v^{(k)}) \quad \forall v \in V$$

By using n GNN layers, the output features of each vertex (also called embedding) can reflect features from all its n-hop neighbors. The three functions ϕ, ρ, and ψ constitute the GNN model and encapsulate its parameters.

Distributed DNN Training Before discussing approaches to scale GNN training, it helps to review the basic approaches to scale DNN training that are adopted by existing frameworks such as TensorFlow [1] and PyTorch [30]. Scalability requires distributing the workload across devices, typically GPUs, that execute the DNN model. Data and model parallelism are the most common paradigms to do large-scale DNN training.

Data parallelism partitions the input data into mini-batches and assigns each mini-batch to one device. Each device has a full copy of the model and it independently performs iterations over the model (forward and backward propagation) using the mini-batch as input. The iteration produces gradients for the model parameters. Gradients from all devices are then aggregated, using a parameter server [22] or an all-reduce protocol [31], and finally applied to each copy of the model.

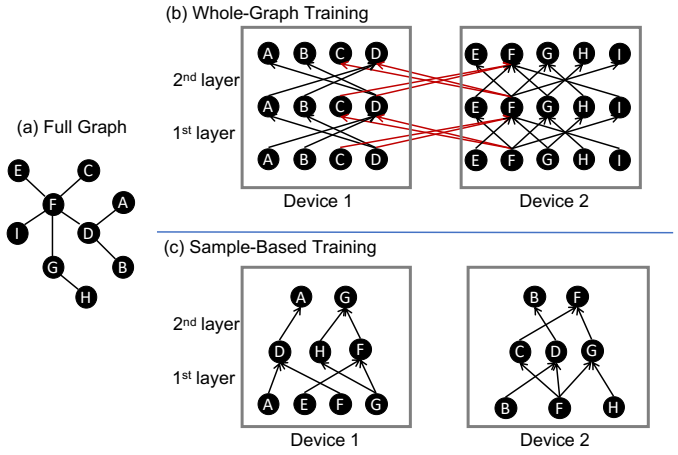

Figure 1: Example of whole-graph and sample-based training.

Data parallelism is the default choice to parallelize training because each device can complete iterations independent of each other. It is particularly suitable for small models, such as models with small convolutional filters, because the cost of aggregating gradients across devices is limited [21]. However, for large models, gradient aggregation can be a substantial overhead. Furthermore, if the model is too large or complex, it might exceed the capacity of one device, making data parallelism insufficient.

Model parallelism partitions the model across multiple devices, for example by assigning a subset of consecutive layers to the same device. Completing an iteration requires interaction among devices, which exchange the features produced by their partition of the model. In this case, the model parameters are not replicated and gradients can directly be applied by the device that computes them.

Model parallelism is the best fit for models that exceed the capacity of one device. They also perform well when each partition of the model requires a large amount of computation and produces small features. It does not require exchanging gradients, so it works well with models with a large number of parameters, such as models with many fully-connected layers [21]. An important drawback of model parallelism is that it requires tighter coordination among devices: downstream devices can only make progress once upstream devices send their features. The communication and coordination cost of model parallelism can be reduced by using techniques like pipelining that combine mini-batching and model partitioning [28].

3 Scaling GNN Training

When the input graph is too large to be handled by one device, it is necessary to use a distributed approach. However, neither data nor model parallelism are a good fit for GNN training. A graph is a single data structure consisting of interconnected vertices, not a collection of small independent samples, as typically assumed in data parallelism. There is no way to partition the vertices of most graphs without having edges between partitions. As discussed, on the other hand, GNN models are relatively shallow and small compared to DNNs, making model parallelism a bad fit for GNN as well.

Because of the uniqueness of GNNs, prior work has proposed two approaches that are tailored to distributed GNN training: *whole-graph* and *sample-based* training. We now present and compare them. Table 1 reports a summary of the comparison and Figure 1 shows an example of a two-layer GNN trained on two devices using the two approaches.

Whole-Graph Training Whole-graph training was adopted by two systems for GNN training, NeuGraph [25] and Roc [20]. It partitions the input graph and assigns each partition to one device. One way to look at whole-graph training is to treat the input graph as a single sample, which is then partitioned across one of the attribute dimensions [20]. The GNN model parameters are replicated on all devices. Each iteration is processed one GNN layer at a time. At each layer k, each device computes the feature vector $h_v^{(k)}$ for each vertex v in its partition. This requires fetching the feature $h_u^{(k-1)}$ for each neighbor $u \in N(v)$ of v, which typically include features computed by other devices in the previous layer. Therefore, in whole-graph training devices must exchange features to complete iterations, a characteristic whole-graph training shares with model parallelism. Whole-graph training is a form of full-batch training: each iteration is executed on the entire input.

In the example of Figure 1, whole-graph training partitions the vertices on the left side across two devices. The edges marked in red connect vertices in two different partitions. At every layer, each vertex must aggregate the features of its neighbors, which creates cross-device communication and coordination along those edges. In the forward pass, devices exchange features (i.e., the vertex features) at each layer, whereas in the backward pass they exchange the gradients of the features.

Sample-Based Training Sample-based training is supported by AliGraph [41], DistDGL [39], and PaGraph [23]. The first two target distributed training, the last one runs on multi-GPU systems. This approach first performs graph sampling to create mini-batches (also referred to as *samples*), and then trains on the mini-batches in parallel on different devices. The goal of sample-based training is to enable devices to complete iterations independently on their mini-batches without having to exchange features. Each mini-batch includes the data necessary to compute the output feature vector for some vertices called *target vertices*. In a GNN model with n layers, each mini-batch includes the input features of the n-hop neighborhood of those target vertices. Sample-based training is a form of mini-batch training since iterations are computed on a piece of graph data.

This approach differs from data parallelism since mini-batches contain redundant data. Each vertex is included in the n-hop neighborhoods of multiple vertices, and can hence be included in multiple mini-batches. Nonetheless, this approach has analogies to data parallelism. Devices can complete iterations without communicating with each other. Furthermore, exchanging the gradients among devices is not expensive since the model is small.

Consider again the example of Figure 1. Sample-based training creates two mini-batches and assigns them to the two devices. The first device computes the output features of target vertices A and G, whereas the second one computes them for B and F. Each mini-batch includes all vertices in the two-hop neighborhood of the target vertices assigned to the device. Sampling the graph to obtain mini-batches is part of the training process but it is not depicted in the figure.

Why Scaling Whole-Graph Training is Difficult The main drawback of whole-graph training is that devices need to exchange features (the updated vertex features) at each GNN layer (see Figure 1(b)). The communication complexity depends on the number of edges across partitions and can be large. Exchanging features also introduces control dependencies between devices within the context of one iteration.

Data-dependent communication and coordination constraints among devices are a major hurdle when scaling to large clusters. Increasing the number of partitions reduces the computational load per worker but it also increases the number of edges across partitions, resulting in higher communication and coordination costs both globally and locally at each device.

These scalability bottlenecks are not unique to whole-graph GNN training. Distributed graph computation systems such as Pregel also partition vertices among workers [6, 13, 26, 42]. Each worker must coordinate with the others at each superstep and exchange messages among neighboring vertices. There are important differences between the two classes of systems. In whole-graph GNN training, devices exchange known data types (tensors) and perform known operators, for example sparse matrix multiplication. This enables more fine-grained vectorization and scheduling compared to graph processing systems, where vertices execute UDFs and exchange messages that are opaque to the system. However, the communication and coordination patterns are similar. Scaling out graph computation systems to large clusters has always been a difficult research challenge (see for example [16]) and the same holds for whole-graph GNN training.

Moving towards sample-based training for better scalability does not necessarily means degraded model accuracy.

Model	Unit of parallelism	Gradient descent	Exchange of activations	Exchange of gradients	Redundant computation	External overheads
Whole-graph	Vertex-centric	Full-batch	Yes (per-layer)	Yes	No	Graph partitioning (per-task)
Sample-based	Subgraph-centric	Mini-batch	No	Yes	Yes	Sampling (per-iteration)

Table 1: Comparison of approaches to scalable GNN training.

Although some recent system work on improving GNN training scalability point out that sampling techniques come with potential model accuracy loss for large real-world graphs, they compare whole-graph training with few graph sampling approaches [20, 25]. The more recent graph sampling literature has consistently shown better accuracy on common graph benchmarks [8, 38].

A Case for Sample-Based Training In this paper, we claim that sample-based training is a more promising approach to distributed GNN training. Instead of revisiting known scalability bottlenecks in the new context of GNNs, we should leverage the new opportunities offered by GNNs to avoid those bottlenecks altogether.

Sample-based training eliminates per-layer coordination and communication costs. Devices only need to exchange gradients, which is a small cost since the size of the model is small and independent of the size of the graph. Using asynchrony or bounded-staleness for gradient exchange can further reduce these coordination costs.

Sample-based training also results in a more modular system design. Once devices obtain mini-batches, they can calculate gradients by using any GNN training frameworks designed for graphs that fit in one device. Gradient exchange can occur just like in data parallelism, which is already well supported.

This does not mean that scaling sample-based training is straightforward. There are two main challenges to scalability: redundant work and the sampling overhead. However, recent research indicates that both challenges can be solved.

Redundant work arises from the "neighbor explosion" problem: each vertex can have a very large number of n-hop neighbors, which must be included in its mini-batch. Multiple mini-batches are likely to overlap in many common vertices. Multiple devices must then compute the features for these vertices, resulting in redundant computation and memory costs. For example, in Figure 1, the first-layer feature of vertex D, $h_D^{(1)}$, is computed by both devices, and the input features of vertex A are replicated at both devices.

A solution to this problem is to sub-sample the subgraphs constituting the mini-batches to prune some vertices and edges. Re-sampling mini-batches at each iteration reduces the chances of missing important information. Developing sampling algorithms to create mini-batches that maximize accuracy and minimize training time is a very active area of research and has shown that this is feasible. Table 3 lists some of these algorithms, which we discuss further in Section 4.1.

Input Graphs	PPI	Reddit
GraphSAGE [14]	51%	45%
FastGCN [5]	26%	52%
LADIES [43]	40%	62%
ClusterGCN [8]	4.1%	24%
GraphSAINT [38]	25%	30%
MVS [9]	24%	25%

Table 2: Fraction of time spent in graph sampling and training by different GNN algorithms (from [19]).

Graph sampling can take a significant portion of the total training time in real-world applications. The computation is irregular and is typically performed using the CPU. In our previous work, we found that graph sampling can take up to 62% of an epoch's time if the host has a single GPU (see Table 2) [19]. This bottleneck is further exacerbated if the CPU is attached to multiple GPUs consuming samples for training. Therefore, speeding up and scaling sampling is an important problem for graph sampling. Preliminary work on this front shows promising results too, as we will discuss in Section 4.2.

4 Scaling Sample-Based Training

In this section, we review recent work on graph sampling for GNN training. We first describe algorithms to obtain the subgraphs that constitute the mini-batches of sample-based training. Then, we discuss systems to speed up the execution of sampling algorithms.

4.1 Sampling Algorithms

Sampling algorithms in GNN training aim to select a subset of vertices and edges based on certain rules. After sampling, instead of using all neighbors as in the whole-graph training, sample-based training constructs a vertex's feature by only aggregating the features of the sampled set of neighbors. Existing sampling approaches largely fall into four categories: *node-wise* sampling, *layer-wise* sampling, *subgraph-based* sampling, and *heterogeneous* sampling [24]. They differ in the granularity of the sampling operation in one training mini-batch. *Heterogeneous* sampling applies to heterogeneous graphs whose edges and vertices are of different types. As our discussion focuses on homogeneous graphs, we next elaborate on the first three types of sampling algorithms.

Node-wise sampling applies sampling operations to each vertex's neighbors: a part of neighbors of a vertex are sampled based on specific probability (e.g., uniform distribution) to compute the vertex's feature. One typical example is GraphSAGE [14]. It uniformly samples a fixed number of neighbors for each vertex in the graph and aggregates their features to generate the vertex's feature in each GNN layer. The output feature of each vertex from the final GNN layer is then used for the GNN model's weight update and downstream tasks. Its variants include PinSage [37], SSE [10], VR-GCN [4], and MVS [9], which differ in the design of sampling functions.

Layer-wise sampling samples multiple vertices simultaneously for each GNN layer in one sampling step. This approach is usually faster than node-wise sampling as it avoids the exponential extension of neighbors. Example algorithms include FastGCN [5] which samples a fixed number of vertices in each GNN layer based on pre-calculated probability independently and LADIES [43] and AS-GCN [15] which introduce layer dependencies and sample vertices in the k-th GNN layer based on vertices sampled in the $k+1$-th layer.

Subgraph-based sampling samples a subgraph, which is composed of selected vertices and edges, and conducts training using the subgraph. Existing work generate subgraphs by either partitioning the whole graph or extending vertices and edges using specific policies [2, 8, 38]. For example, ClusterGCN [8] first partitions the whole graph into multiple clusters using graph clustering algorithms and then randomly samples a fixed number of clusters as a mini-batch by combining these clusters into a subgraph. GraphSAINT [38], on the other hand, leverages random walk to sample neighbors of a vertex and generate subgraphs with the selected vertices.

4.2 Systems for Efficient Sampling

Building samples of a graph is an irregular computation that is hard to perform efficiently. Graph sampling algorithms are designed to preserve good accuracy while generating small mini-batches. The researchers who develop these algorithms should not have to deal with the low-level details of hardware architectures to optimize performance. There is an emerging need for systems that can abstract away the graph sampling process by offering a high-level yet generic API and an efficient runtime to execute these programs. These systems should be able to scale to large graphs and integrate with the training process.

Fast Graph Sampling on GPUs Sample-based GNN implementations often rely on frameworks such as DGL and PyTorch Geometric to perform GNN training on GPUs. However, they perform graph sampling on the CPU. This is in part due to the lack of frameworks for efficient and generic sampling on GPUs.

Our recent work on NextDoor addresses this problem [19]. NextDoor enables users to express graph sampling tasks using a general, high-level API. It then executes these tasks efficiently using GPUs. The API is expressive enough to support the sampling algorithms described previously. Using NextDoor can speed up the end-to-end training time of existing GNN systems by up to 4.75× (see Table 3).

	PPI	Reddit	Orkut	Patents	LiveJ
GraphSAGE	1.30×	1.21×	OOM	1.20×	1.22×
FastGCN	1.25×	1.52×	4.75×	2.3×	4.31×
LADIES	1.07×	1.37×	2.27×	2.1×	2.34×
ClusterGCN	1.03×	1.20×	OOM	1.4×	1.51×

Table 3: End-to-end speedups after integrating NextDoor in GNNs over vanilla GNNs (from [19]).

```
Vertex next(s, transits, srcEdges, step) {
    int idx = randInt(0, srcEdges.size());
    return srcEdges[idx];}
int steps() {return 2;}
int sampleSize(int step) {
    return (step == 0) ? 25 : 10;}
bool unique(int step) {return false;}
SamplingType samplingType(){
    return SamplingType::Individual;}
Vertex stepTransits(step, s, transitIdx){
    return s.prevVertex(1, transitIdx);}
```

Figure 2: GrapSAGE's 2-hop neighbors sampling implemented using the NextDoor API (from [19]).

Figure 2 shows an implementation of a sampling algorithm using the NextDoor API. The algorithm is used by GraphSAGE [14], a classic GNN algorithm. This particular implementation samples the 2-hop neighborhood of a vertex. The `steps` function indicates that sampling performs two recursive hops from the vertex. The `sampleSize` function returns how many neighbors of a vertex are sampled at each step: 25 neighbors of the starting vertex in the first step and 10 neighbors of each sampled neighbor in the second. The `next` function specifies how to pick a neighbor of a vertex.

Beyond offering an easy-to-use API, NextDoor introduced a novel approach to parallelize graph sampling called *transit parallelism*. This is better suited to GPU architectures than the approach used by other systems for graph sampling, such as KnightKing [36] and C-saw [29], and graph mining, such as Arabesque and others [3, 7, 11, 17, 18, 27, 32, 33]. All these systems expand multiple samples in parallel and assign each sample to a group of consecutive threads, which could be part of the same warp. This approach, which we call *sample parallelism*, results in sub-optimal performance on GPUs because of their hardware architecture. GPU kernels achieve optimal performance when threads in a warp access contiguous memory locations, cache data on shared memory, and perform the same steps. Sample parallelism does not have these properties.

To see why, consider the example of Figure 1, and suppose that the same GPU is computing the two mini-batches for the

two devices. Each mini-batch corresponds to a sample. Suppose that the algorithm is now expanding the first sample by adding neighbors of its vertices A and G. A sample-parallel execution of the sampling algorithm would associate the sample to a group of consecutive threads, which are likely to be in the same warp. These threads will access the adjacency lists of two different vertices, A and G. This leads to uncoalesced memory accesses and poor access locality since the two adjacency lists are located at random locations in the GPU memory. It also results in warp divergence if the threads scan adjacency lists of different sizes.

NextDoor uses each *transit vertex* as the fundamental unit of parallelism when building the samples. A transit vertex is a vertex whose neighbors may be added to one or more samples of the graph. NextDoor groups all samples that need to "transit across" a vertex and assigns the samples to consecutive GPU threads. Each thread accesses the adjacency list of the transit and adds one neighbor of the transit vertex to one sample. Since all threads access the same list, NextDoor achieves coalesced global memory reads, low warp divergence, and effective caching using the shared memory of the GPU. We compared two versions of NextDoor that parallelize sampling by sample and by transit, using several sampling algorithms implemented using NextDoor's API. Transit parallelism has shown to be consistently faster, as shown in [19].

Consider again the example of Figure 1 and suppose that the same GPU is computing the mini-batches (i.e., the samples) for the two devices. Suppose that the two samples already contain the target vertices ({A, G} and {B, F} respectively) and their one-hop neighbors ({D, H, F} and {C, D, G} respectively). The algorithm is now expanding both samples by adding the two-hop neighbors of the target vertices. Vertex D is a transit vertex for both samples because the algorithm must select neighbors of D for both samples. NextDoor assigns a group of consecutive threads to vertex D and both samples. All these threads read the adjacency list of D. Accesses to the list can be coalesced and the list can be cached in shared memory. If the threads scan the adjacency list, they will perform the same number of steps. These properties result in better performance on GPUs because of their hardware architecture.

Scaling to Larger Graphs Scaling sample-based training to large graphs critically relies on scaling sampling. Once the sampling process generates mini-batches, distributed training on each mini-batch proceeds independently similar to data parallelism. However, efficiently sampling large graphs that are stored on a distributed system is non-trivial.

One research question to address is how to scale graph sampling across devices. Some existing systems have already explored the problem. KnightKing computes random walks on distributed graphs and uses a graph processing system as a substratum [36]. AliGraph [41] and DistDGL [39] are end-to-end GNN training systems, integrating sampling and training. PaGraph performs sample-based GNN training in multi-GPU systems [23]. KnightKing and DistDGL run several sampling workers that incrementally expand their samples and pull remote graph data on demand. Distributed graph mining systems like G-Miner use a similar approach [3]. These systems, however, differ in the way they schedule distributed sampling. Finding the optimal strategy is still an area for research, and it will be interesting to see whether ideas from graph mining systems can be borrowed in this context.

Another problem with the aforementioned systems is that they perform sampling using CPUs only. GPU-based sampling has the potential to significantly reduce end-to-end training time, as shown in NextDoor [19]. Distributed sampling systems can leverage GPUs for improved throughput. NextDoor supports multi-GPU–based sampling if the graph fits in the device memory. How to scale the process to larger graphs is still an open area of research. Given that GPUs are also used for GNN training, another open issue is how to efficiently overlap sampling with training on the same set of devices.

Finally, how to distribute and store the graph data and how to transfer it in and out of the GPUs is another critical aspect in the performance of distributed sample-based training. PaGraph, for example, uses caching to minimize data transfers [23]. More research is likely to unveil additional optimizations.

5 Conclusions

In this paper, we have compared two approaches to scale GNN training: whole-graph and sample-based. Whole-graph training requires devices to coordinate and communicate at each GNN layer, which represents a challenge for scalability. Sample-based training avoids this coordination altogether, so it promises better scalability. Scaling sample-based training requires (a) sampling algorithms that can form mini-batches without losing too much information or generating excessive redundant work, and (b) systems that can execute these algorithms efficiently. Recent work indicates that both requirements can be fulfilled.

Acknowledgements

The authors would like to thank the NextDoor team: Abhinav Jangda, Sandeep Polisetty, and Arjun Guha. This work was partially supported by a Facebook Systems for Machine Learning Award and an AWS Cloud Credit for Research grant.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning.

In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Jiyang Bai, Yuxiang Ren, and Jiawei Zhang. Ripple walk training: A subgraph-based training framework for large and deep graph neural network. *arXiv preprint arXiv:2002.07206*, 2020.

[3] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–12, 2018.

[4] Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. *arXiv preprint arXiv:1710.10568*, 2017.

[5] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations*, ICLR'18, 2018.

[6] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Trans. Parallel Comput.*, 2019.

[7] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proc. VLDB Endow.*, 2020.

[8] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, 2019.

[9] Weilin Cong, Rana Forsati, Mahmut Kandemir, and Mehrdad Mahdavi. Minimal variance sampling with provable guarantees for fast training of graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, 2020.

[10] Hanjun Dai, Zornitsa Kozareva, Bo Dai, Alex Smola, and Le Song. Learning steady-states of iterative algorithms over graphs. In *International conference on machine learning*, pages 1106–1114. PMLR, 2018.

[11] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, 2019.

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

[13] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

[14] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, 2017.

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

[16] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, et al. Ldbc graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proceedings of the VLDB Endowment*, 9(13):1317–1328, 2016.

[17] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, 2018.

[18] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[19] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. Accelerating graph sampling for graph machine learning using gpus. In *European Conference on Computer Systems (EuroSys)*, 2021.

[20] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems*, 2:187–198, 2020.

[21] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[22] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed

machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

[23] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 401–415, 2020.

[24] Xin Liu, Mingyu Yan, Lei Deng, Guoqi Li, Xiaochun Ye, and Dongrui Fan. Sampling methods for efficient training of graph convolutional networks: A survey. *arXiv preprint arXiv:2103.05872*, 2021.

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

[26] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, 2010.

[27] Daniel Mawhirter and Bo Wu. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, 2019.

[28] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[29] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. C-saw: a framework for graph sampling and random walk on gpus. *arXiv preprint arXiv:2009.09103*, 2020.

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

[31] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[32] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A System for Distributed Graph Mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, 2015.

[33] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on a Single Machine. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, 2018.

[34] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. Deep graph library: Towards efficient and scalable deep learning on graphs. 2019.

[35] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 2020.

[36] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. KnightKing: A Fast Distributed Graph Random Walk Engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, 2019.

[37] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 974–983, 2018.

[38] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, ICLR '20, 2020.

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

[40] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

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

[42] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[43] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. Layer-dependent importance sampling for training deep and large graph convolutional networks. In *Advances in Neural Information Processing Systems*, Nuerips '19, 2019.