Survey of Graph Neural Network Systems

Yuhao Zhang
University of California, San Diego
yuz870@eng.ucsd.edu

ABSTRACT

A brief survey of GNN systems.

1 OVERVIEW

1.1 Compared systems

GNN systems are designed to execute (and scale) GNN workloads. Some focus on single-node execution and optimizing data movement across memory hierarchy; some focus on distributed settings and optimizing network communications and graph partitioning. The usual goals are better runtime efficiency, measured as per-epoch training time, and data/model scalability, tested by the system capable of executing the workload without OOM issues. This is an emerging field, and many of these systems are still developing and changing rapidly.

Although there are many sub-branches in GNN study, the vast majority of these systems focus on the spatial-based methods, that is, recurrent GNNs and spatial-based graph convolutional neural networks (also known as GCNs). Majority of spatial methods can be expressed via a general update rule:

$$\mathbf{h}_v^k = \psi(\mathbf{x}_v^k, \prod_{u \in \mathcal{N}(v)} \phi(\mathbf{h}_v^{k-1}, \mathbf{h}_u^{k-1}, \mathbf{x}_{e_{vu}})), \tag{1}$$

where ψ, ϕ, Γ are potentially learnable and differentiable functions, Γ is typically further required to be commutative and associative. ψ is called the update function, ϕ the message function and Γ the aggregate function. $\mathbf{h}_v^k$ is the node embedding for node v at the k-th layer. $\mathcal{N}(v)$ is a function that returns all the neighbors for node v. $\mathbf{x}_{e_{vu}}$ is the edge feature.

PyTorch Geometric (PyG) Equation 1 fits in the paradigm of vertex-centric programming, which is a common programming model in the graph processing world [6, 15, 16] and is very easy to parallelize. One of the first GNN systems, PyG [4] follows this pattern and builds GNNs as dataflow graphs on top of PyTorch [19], a dataflow system. Figure 1 (B) shows an example computational graph generated[1]. Each (vectorized) operation is then put on a separate processor to achieve parallelism. Built on top of PyTorch, PyG can easily utilize the features PyTorch provides, such as autograd, GPU acceleration, etc.

Deep Graph Library (DGL) and DistDGL. Arguably the most popular GNN system, DGL [22] takes a different approach for the GNN execution, especially for the message passing (gather-scatter) part. It takes an algebraic approach for the execution by expressing GNN as sparse matrix multiplications (SpMM). This is analogous to those algebraic approaches to general graph analytics tasks [10].

Figure 1 (C) gives an example of the algebraic approach taken by DGL; each step is summarized as linear algebra and executed as SpMM, which is executed via DGL's own specialized kernel. DGL is built upon existing deep learning systems, and the user can also write user-defined functions to extend the built-in functions. DGL describes itself as a framework-independent system, that is, independent of the underlying deep learning system. It has full support for several popular frameworks such as Tensorflow or PyTorch as backend. Backpropagation can also be done via another SpMM, and DGL functions are registered in the underlying deep learning framework to take advantage of autograd.

The algebraic approach opens possibilities of operator fusion. For many GNNs, the message function and aggregate function can be represented with purely matrix multiplications. Note in this form, the message function can be fused with the aggregation function, and there is no need to materialize the intermediate messages, thus saving space. For scalability, DGL goes for distributed processing as DistDGL [25].

NeuGraph. NeuGraph [14] is another GNN system built on top of a dataflow system and designed for a single-node multi-GPU setting. It extends the GAS [6] programming model and partitions the graph into chunks. It feeds chunks instead of the full graph to GPU, avoiding GPU OOM issues for GNNs without sampling the data. After feedforward propagation, chunks are swapped back to the main RAM. There are also a series of processing order and scheduling challenges that it tries to solve. It further exploits GPU-to-GPU communication to avoid latencies of GPU-to-DRAM.

PaGraph. PaGraph [1, 12] also targets single-node multi-GPU settings and claims that the majority of the time (74%) during GNN training is spent on data movement between the main RAM and GPU memory. Therefore, to reduce such data movement, they utilize the spare GPU memory to cache certain graph nodes' features statically. They select this set of cache based on the nodes' degree. They then proposed a GNN-aware graph partition scheme for multi-GPU support. This partition scheme would ensure that each partition contains all of the training graph nodes' K-hop neighbors by replication. In a subsequent work [1], the authors also incorporated pipelining. PaGraph is built on top of DGL and PyTorch.

AliGraph. AliGraph [27] is one of the early distributed GNN systems that come with only CPU support. The paper proposes a GNN training platform and a GNN algorithm to tackle real-world e-commerce graph analytics challenges. On the system side, they propose to store the graph structural information and node features data separately. AliGraph then uses indices to represent the node features, therefore reduces the storage cost. They then adopt an LRU cache to reduce access time for the most frequent graph nodes above a threshold based on their degrees. Furthermore, they adopt a lock-free distributed data structure (with cache) that facilitates graph sampling common in GNN training.

[1]Note this observation is based on the paper of PyG, in the recent versions of the release, PyG has implemented support for algebraic approach as well.

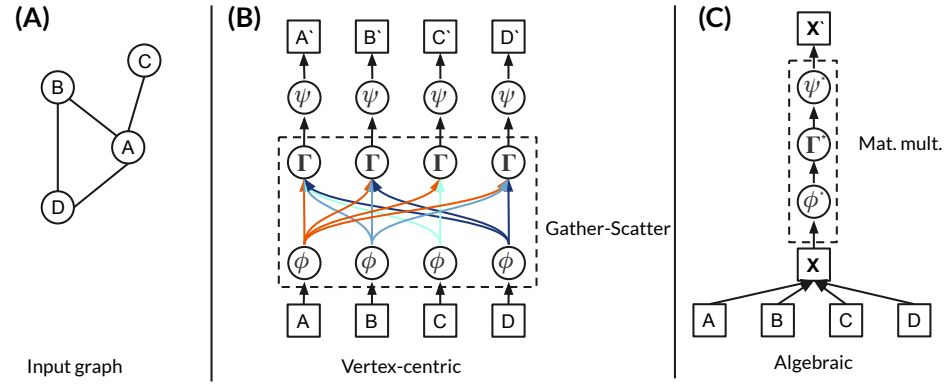

Figure 1: An example of two different approaches for GNN execution. (A): Input graph. (B): Vertex-centric approach. (C): Algebraic approach.

ROC. ROC [9] is a distributed GNN system that attempts to boost both the efficiency and scalability of GNN training. Towards the former goal, it adopts an ML-based method for determining the graph partition. For the later pursuit of scalability, it proposes system DRAM spill when the GPU memory is insufficient to execute the GNN workload. ROC is built upon a custom-built DNN execution backend with customized CUDA kernels, FlexFlow [13].

P^3. P^3 [5] is a distributed GNN system that proposes to use a "hybrid" parallelism for GNN training. They argue that under the distributed setting, training time is dominated by communication time. They also claim that the bulk of communication happens during the first layer of GNNs. P^3 then proposes to 1. partition the graph structure and node features independently. 2. partition the node features uniformly along the dimensions. This way, they execute GNNs' first layer via model parallelism since each machine contains only several dimensions of the features. They then invoke all-to-all communications, let each machine pull all the needed features, and proceed as normal data parallelism for subsequent layers. This execution scheme would bring extra overhead compared to simple data parallelism, but the authors argue such overheads are compensated by the performance gain brought by communication reduction.

DeepGalois. DeepGalois [8] is a CPU-only distributed GNN system built on a graph processing system Galois [20]. Additionally, it is also built on top of a series of existing graph partitioning and distributed communication frameworks. Because of these mature components, DeepGalois can support many graph partitioning schemes and can outperform distDGL. At the moment, there is not yet a full paper about DeepGalois, and no public release can be found.

Neo4j, TigerGraph, and GraphScope. Both Neo4j [18] and TigerGraph [3] are graph DBMSes that have some GNN training capabilities. GraphScope [23] is a recent graph processing system that primarily focuses on traditional graph analytics such as community detection, paths and connectivity, and centrality, but it also has extensible connectors to other systems to enable GNN processing.

Neo4j has implemented a specialized in-memory graph store for GNNs and other more advanced analytics. It first moves data from the graph DB to this store at runtime and invokes a custom-built library for building GNN models and SGD training. As of now, it only supports GraphSage.

GraphScope takes a similar approach to Neo4j by moving data to a dedicated GNN facility. But instead of a custom-built framework, it can utilize TensorFlow and provides a few pre-built implementations of several popular GNNs.

TigerGraph, similar to Neo4j, only provides limited GNN support for simple GCNs. However, it takes a different approach by also storing all the NN neurons in graph DB. As mentioned above, such GNNs can be captured under the GAS abstraction. This way, the GNN can be expressed mostly in the form of graph DB operators instead of the external library's dataflow operators. Forward and backward propagation can be written using TigerGraph's DSL: GSQL. Gradient computing is implemented in the form of UDFs.

1.2 Other related systems

Custom-built, Specialized GNN Frameworks. Due to the lack of support for GNNs from the commonly used deep learning frameworks such as TensorFlow and PyTorch and various other reasons, many ML researchers sought to build their own library for their specific GNNs/applications [7, 24]. These frameworks usually are very specific and have little consideration for system performances such as efficiency and scalability. Therefore, they are excluded from our discussion.

Knowledge Graph Embedding Systems. There exists a connected, yet still different field of knowledge base graph embedding systems [11, 17, 21, 26]. The goal of such systems is to obtain embeddings for each node within the graph. However, instead of training a GNN and using its internal latent node representations as node embeddings, these methods train the embeddings directly using the graph's structural information. They do not use GNNs, and they usually work on graphs that are not featured, i.e., node and edge features unavailable. These systems also face efficiency and scalability challenges, mostly brought by the sheer size of knowledge base graphs instead of GNNs. Many of them sought to use distributed processing, intelligent caching, and disk-aware processing to tackle these problems. Some of the techniques can be adopted for GNN

systems, but many are specific to knowledge graph embedding workloads.

2 COMPARISONS OF GNN SYSTEMS

An overview of all the systems compared is shown in Table 1.

2.1 Availability (licenses)

To benchmark and development on top of these systems, ideally, we want them to be free and open-sourced.

Open-source license. The majority of the GNN systems surveyed are open-sourced; exceptions are below.

Proprietary license. TigerGraph is released under a proprietary license; the free version comes with a limitation of graph storage size. Neo4j, on the other hand, has both a free open-sourced version that comes with various limitations and a paid commercial version.

Not available. There are no public releases found for NeuGraph, DeepGalois, and P^3.

2.2 Accelerator (GPU) Support

With GPU support, two extra problems arise 1. efficient data movement between main memory and GPU memory 2. memory management to prevent crashes, as GPU RAM is usually more limited. Not all systems have GPU support.

GPU-supported. PyG, DGL, NeuGraph, PaGraph, ROC, P^3, and GraphScope all support GPU training. Among these, PyG, DGL, and ROC have custom-built GPU kernels for GNNs. The rest are based on existing systems for GPU support.

CPU-only. AliGraph, DeepGalois, Neo4j, and TigerGraph are CPU-only at the moment.

2.3 Runtime Efficiency (Speculated)

This section is based on the published results from the papers. These results are not directly comparable, as they are not done on the same platform, nor the same datasets/models. Hence, currently, without more rigorous tests, the efficiency can only be speculated. There are many aspects to the runtime efficiency: 1. data movement efficiency across memory hierarchy, pipelining, and efficiency latency hiding. 2. The network communication efficiency, graph partitioning quality, cache, and novel parallelization schemes for distributed systems. 3. actual training efficiency, i.e., runtime spent in GPU.

In the literature, the more common baselines are DGL and PyG. Between the two, both of their papers claim to be faster and more scalable; it is unknown if they have a substantial performance gap.

Data movement efficiency. A few systems have efforts to optimize the data movement, especially the DRAM-GPU movement. The common techniques are caching and pipelining. NeuGraph, ROC, and PaGraph all adopt some form of caching to reduce the data movement and try to overlap computation with data transfer. NeuGraph further exploits opportunities for GPU-GPU data movement.

Network communication efficiency. Networking becomes part of the problem when distributed processing comes to the picture. AliGraph and ROC both consider graph partitioning algorithms and caching to boost data locality and reduce communication. DeepGalois relies on the underlying graph processing software stack for optimized graph partitioning and distributed communication. P^3, on the other hand, proposes a parallel execution scheme that mixes model parallelism and data parallelism to reduce communications.

Training efficiency. Most of the systems should have close performance in this department. However, as PyG, DGL, NeuGraph, and ROC all have their custom-made GPU kernels for graph propagation, it is unknown which one is the winner as no such drill-down benchmark is currently available. In addition, P^3 adopts a model parallelism training scheme that is different from all other systems, and this scheme, although it may save communications, but would likely add overheads to training.

2.4 Scalability (Speculated)

Due to the neighborhood explosion problem and the scale of large graphs, GNNs suffer from scalability issues as the entire graph is usually much larger than the GPU memory can hold. A series of algorithmic efforts, focusing on sampling, have been proposed to mitigate this issue. However, even with mini-batch SGD and sampling over the data, it is claimed that many of such workloads can still face scalability issues. Common techniques include 1. distributed processing that enables multi-GPUs, and 2. spilling to the main RAM to prevent OOM. Distributed processing would lead to topics such as graph partitioning, network communication, and parallel schemes. Spilling to the main RAM is tightly connected to topics such as caching and pipelining.

Distributed processing. DGL, ROC, DeepGalois, P^3, AliGraph, and TigerGraph can all support distributed GNN training. PyG, NeuGraph, PaGraph are single-node systems.

Neo4j and GraphScope are unknown if their export-based methods can be made distributed. Apart from this, Neo4j is currently still primarily a single-node and not fully distributed graph DB[2].

Graph sampling. There are mainly two categories of systems in terms of graph sampling and training scheme: full-batch and mini-batch.

(1) **Full-batch. (Gradient descent and unsampled)**. The first try to optimize for full-batch training of GNNs, i.e., naively, would require the entire graph to fit in the main or GPU RAM. This category includes NeuGraph, ROC, and DeepGalois. TigerGraph currently has some support for GCNs and can work under a full-batch setting. However, it is unknown to what extend it is scalable.
(2) **Mini-batch. (SGD and sampled)**. The second category assumes a mini-batched and sampled graph. Therefore they only require a mini-batch of data to fit in the RAM. This category includes PaGraph, AliGraph, and P^3. Neo4j only supports GraphSage algorithm at the moment, which is a mini-batch method with sampling.
(3) **Generic. (Unspecified)**. These systems claim to be generic or not specifically optimized for any one particular setting. It includes DGL, PyG, and GraphScope.

[2] https://neo4j.com/news/neo4j-going-distributed-with-graph-database/

Table 1: Overview of comparisons.

	License	GPU	Efficiency				Scalability		Expressibility	Memory Hierarchy
			Data movement	Communication	Training	Overall	Distributed	Sampling		
PyG [4]	Open	Yes	-	N/A	+	-	No	Generic	GAS	GPU-only
DGL [22, 25]	Open	Yes	-	-	+	-	Yes	Generic	MM	GPU-only
NeuGraph [14]	N/A	Yes	++	N/A	+	+	No	GD	GAS	Main-aware
PaGraph [1, 12]	Open	Yes	++	N/A	+	++	No	SGD	MM	Main-aware
AliGraph [27]	Open	No	+	+	-	-	Yes	SGD	GAS	Main-only
ROC [9]	Open	Yes	+	+	+	++	Yes	GD	GAS	Main-aware
P^3 [5]	N/A	Yes	-	++	--	++	Yes	SGD	MM*	GPU-only
DeepGalois [8]	N/A	No	-	+	-	+	Yes	GD	GAS	Main-only
Neo4j [18]	Mixed	No	--	N/A	-	--	No	SGD	?	Main-only?
TigerGraph [3]	Closed	No	-	-	-	-	Yes?	GD?	GAS?	Main-only?
GraphScope [23]	Open	Yes	--	?	-	-	No?	Generic?	?	GPU-only?

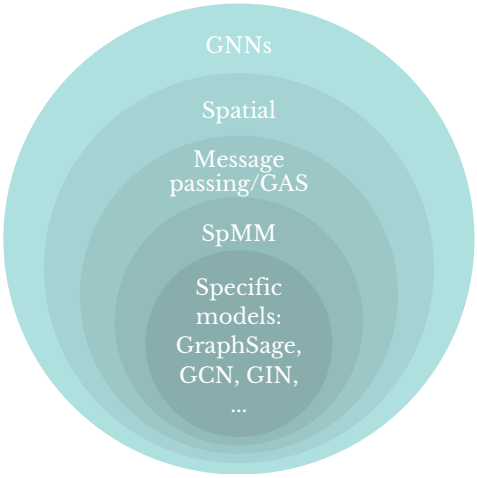

Figure 2: Venn diagram of different scopes of GNNs.

Technically the first category should also work under the mini-batched setting, but the reverse may not be true.

2.5 GNN Expressibility

We now evaluate how generic these systems are and the family of GNN models that can be expressed/executed by these systems. Figure 2 shows the overall scope of comparisons.

- First, all of these systems are primarily designed for spatial methods; most of the spectral-based GNNs are well-supported, except a few methods like ChebNet [2] that bridge the spectral- and spatial-based GNNs.
- Furthermore, many systems are based on the message passing interface shown in Equation 1, also can be captured under the GAS framework. Such systems include PyG, NeuGraph, AliGraph, ROC, and DeepGalois.
- DGL further requires the GNN to be expressible as matrix multiplications. PaGraph is built on top of DGL, so it shares the same expressibility. On the other hand, for P^3 to show performance gain, the update function of the model needs to be model-parallelizable, which usually means linear operations/simply matrix multiplications.
- Neo4j, TigerGraph, and GraphScope did not focus on the GNN expressibility and are unknown how general their programming interface is, or they do not have a generic programming interface for expressing various GNN models.

2.6 Span of Memory Hierarchy

We now inspect each system's usage of the different memory hierarchy, from accelerator memory to disk. Generally speaking, the wider the span is, the better the system can scale with large models and data. Not all systems support GPUs, so some of them do not have GPU RAM. But as more tiers hierarchy are involved, the latencies grow and the demand for buffering and memory management also grow.

GPU RAM Only. DGL, PyG, and P^3 are GPU RAM only, meaning they require the full graph/minibatch to fit in the memory.

Main RAM Only/Aware. AliGraph and Neo4j are main RAM only. NeuGraph, PaGraph, and ROC all utilize the main memory and spill intermediate results to it. They have different strategies to handle the main-memory to GPU data movement. All of the above require the data, GNN model, and intermediate results to fit in the main memory.

Disk Aware. At the moment, there are no systems that explicitly claim that they are disk aware. Neo4j and TigerGraph, as graph DBs, have disk spilling capability for the DB part. However, neo4j processes GNN workloads in a strictly in-memory store, while it is unclear whether TigerGraph can spill GNN intermediate states to disk. GraphScope, on the other hand, relies on external libraries for GNN execution and can be flexible.

REFERENCES

[1] Y. Bai, C. Li, Z. Lin, Y. Wu, Y. Miao, Y. Liu, and Y. Xu. Efficient data loader for fast sampling-based gnn training on large graphs. *IEEE Transactions on Parallel & Distributed Systems*, (01):1–1, 2021.

[2] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, pages 3837–3845, 2016.

[3] A. Deutsch, Y. Xu, M. Wu, and V. E. Lee. Tigergraph: A native MPP graph database. *CoRR*, abs/1901.08248, 2019.

[4] M. Fey and J. E. Lenssen. Fast graph representation learning with pytorch geometric. *CoRR*, abs/1903.02428, 2019.

[5] S. Gandhi and A. P. Iyer. P3: distributed deep graph learning at scale. In *OSDI*, pages 551–568. USENIX Association, 2021.

[6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30. USENIX Association, 2012.

[7] W. L. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NIPS*, pages 1024–1034, 2017.

[8] L. Hoang, X. Chen, H. Lee, R. Dathathri, G. Gill, and K. Pingali. Efficient distribution for deep learning on large graphs. In *MLSys GNNSys Workshop*. mlsys.org, 2021.

[9] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In I. S. Dhillon, D. S. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020.

[10] J. Kepner and J. R. Gilbert, editors. *Graph Algorithms in the Language of Linear Algebra*, volume 22 of *Software, environments, tools*. SIAM, 2011.

[11] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich. Pytorch-biggraph: A large scale graph embedding system. In *MLSys*. mlsys.org, 2019.

[12] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu. Pagraph: Scaling GNN training on large graphs via computation-aware caching. In R. Fonseca, C. Delimitrou, and B. C. Ooi, editors, *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, pages 401–415. ACM, 2020.

[13] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *HPCA*, pages 553–564. IEEE Computer Society, 2017.

[14] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai. Neugraph: Parallel deep neural network computation on large graphs. In *USENIX Annual Technical Conference*, pages 443–458. USENIX Association, 2019.

[15] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146. ACM, 2010.

[16] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2), Oct. 2015.

[17] J. Mohoney, R. Waleffe, H. Xu, T. Rekatsinas, and S. Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *OSDI*, pages 533–549. USENIX Association, 2021.

[18] Neo4j. Neo4j, Accessed October 12, 2021. https://neo4j.com/.

[19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pages 8024–8035, 2019.

[20] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *PLDI*, pages 12–25. ACM, 2011.

[21] J. Qiu, L. Dhulipala, J. Tang, R. Peng, and C. Wang. Lightne: A lightweight graph processing system for network embedding. In *SIGMOD Conference*, pages 2281–2289. ACM, 2021.

[22] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR*, abs/1909.01315, 2019.

[23] J. Xu, Z. Bai, W. Fan, L. Lai, X. Li, Z. Li, Z. Qian, L. Wang, Y. Wang, W. Yu, and J. Zhou. Graphscope: A one-stop large graph processing system. *Proc. VLDB Endow.*, 14(12):2703–2706, 2021.

[24] D. Zhang, X. Huang, Z. Liu, J. Zhou, Z. Hu, X. Song, Z. Ge, L. Wang, Z. Zhang, and Y. Qi. AGL: A scalable system for industrial-purpose graph machine learning. *Proc. VLDB Endow.*, 13(12):3125–3137, 2020.

[25] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis. Distdgl: Distributed graph neural network training for billion-scale graphs. In *10th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms, IA3 2020, Atlanta, GA, USA, November 11, 2020*, pages 36–44. IEEE, 2020.

[26] D. Zheng, X. Song, C. Ma, Z. Tan, Z. Ye, J. Dong, H. Xiong, Z. Zhang, and G. Karypis. DGL-KE: training knowledge graph embeddings at scale. In *SIGIR*, pages 739–748. ACM, 2020.

[27] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou. Aligraph: A comprehensive graph neural network platform. *Proc. VLDB Endow.*, 12(12):2094–2105, 2019.