SoK: Distributed GNN Acceleration/Benchmarking

Mehmet Bagci
Electrical and Computer Engineering
UC San Diego
La Jolla, USA
mbagci@ucsd.edu

Merve Kilic
Electrical and Computer Engineering
UC San Diego
La Jolla, USA
mkilic@ucsd.edu

***Abstract*—Distributed Graph Neural Network (GNN) acceleration strategies are crucial for learning large GNNs on complex graph datasets. In this systemization of knowledge paper, we contribute to the field by providing a comprehensive and analytical comparison of recent advancements. We classify information by acceleration mechanism, compare and rank recent systems and their performances, and draw valuable conclusions to guide future research and explore unexplored areas. Overall, we find that the current best systems of acceleration are BGL, P^3, and MGG.**

***Index Terms*—GNN, distributed, accelerate, benchmark**

I. Introduction

GNNs have revolutionized graph-based learning tasks. However, the increasing scale and complexity of modern graph datasets pose significant challenges in terms of computational efficiency and scalability. Thus, distributed GNN acceleration strategies have gained attention due to their ability to address these challenges and enable efficient training and inference of GNN models.

In this systemization of knowledge paper, we analyze and compare recent distributed GNN acceleration strategies. Our work goes beyond existing surveys by incorporating the most recent advancements in the field, providing a holistic overview of distributed GNN acceleration techniques. This paper's distinguishing factors include an up-to-date overview, a categorization of methods based on acceleration mechanisms, a comparison of system performance, a new analytical system ranking, and the identification of future research directions. By shedding light on the gaps and challenges in the existing literature, our work aims to inspire novel approaches and foster advancements in distributed GNN acceleration techniques. We hope to contribute to the development of efficient and scalable solutions for learning large GNNs, addressing the pressing challenges faced by the graph analytics community.

II. Background

GNNs have gained significant attention in recent years due to their ability to capture complex patterns in graph-structured data. With the increasing size of graph datasets and the complexity of GNN models, distributed GNN training has become necessary to handle computational and memory requirements.

The training of distributed GNNs involves partitioning the graph across multiple machines or GPUs, and performing computations in parallel.

A. Graph Neural Networks

GNNs have emerged as a powerful tool for learning graph-structured data. They are designed to handle irregular data structures and have been successfully applied in various domains, including social network analysis, molecular chemistry, and recommendation systems.

The core idea of GNNs is to learn a function that can aggregate the features of neighboring nodes to generate a new representation for each node. This is typically done through a two-step process: aggregation and update. The aggregation step collects information from the neighboring nodes, and the update step generates a new representation for the node. Mathematically, this can be represented as:

$$a_v^{(l)} = AGGREGATE^{(l)}(\{h_u^{(l-1)} | u \in N(v)\}) \quad (1)$$

$$h_v^{(l)} = UPDATE^{(l)}(a_v^{(l)}, h_v^{(l-1)}) \quad (2)$$

where $a_v^{(l)}$ is the aggregated feature of node v at layer l, $h_u^{(l-1)}$ is the feature of node u at layer $l-1$, and $N(u)$ is the set of neighbors of node u. The $AGGREGATE^{(l)}$ and $UPDATE^{(l)}$ functions are defined by the specific type of GNN being used. For aggregate functions mean, sum, or max functions can be used. This equation can be an example to $UPDATE$ function:

$$h_v^{(l)} = \text{ReLU}\left(W^{(l)} \cdot a_v^{(l)}\right) \quad (3)$$

$W^{(l)}$ is a learnable weight matrix at layer l, and $\text{ReLU}(x) = \max(0, x)$ is the ReLU activation function. The dot product $\cdot$ denotes matrix multiplication.

Through this iterative process, GNNs are able to capture the complex dependencies between nodes in a graph, making them a powerful tool for learning on graph-structured data.

B. Distributed Graph Processing

Distributed GNN training presents several challenges including efficient graph partitioning, minimizing communication overhead between machines, and ensuring the consistency of model parameters. Despite these challenges, distributed GNN training allows for new possibilities for training larger and more complex GNN models and is an active area of research. The first step of distributed GNN training involves partitioning the extensive graph into smaller subgraphs. This

step is crucial to allow the training of large-scale graphs that would otherwise be computationally intensive. Following the partitioning, the subgraphs are distributed to local machines. Each local machine is then in charge of doing computations for its assigned subgraph. During this process, it is essential to continuously update the original graph in a coordinated manner because GNN training relies on neighboring nodes for updates. This makes it so that a high degree of synchronization and communication between the local machines and the main graph is necessary. The specific methodologies for partitioning, data distribution, synchronization, and communication can vary across different frameworks. However, the general structure of distributed GNN training follows this pattern. This approach allows for the efficient training of large-scale GNNs, which would be challenging to achieve using a single computational resource due to memory and computational constraints.

III. Our Contributions

In this SoK paper, we provide five contributions that separate us from existing papers. First, we present an up-to-date overview of new systems. We include works ranging from 2019 to last month. This way, we are able to analyze and compare the newest advancements in the distributed GNN acceleration space. This also allows us to find gaps in current works that can be improved upon in future research. Next, we categorize methods based on acceleration mechanisms as shown in table II to compare the structure, techniques, and novelties of each model. Third, we compare system performances. Many papers compare their speed to a baseline Deep Graph Library (DGL) [14] model. We compiled this speedup factor for comparison between models as shown in table I. Fourth, we have created a new benchmarking method to analytically compare and rank existing systems. We include measures of distributive ability, speed, memory, and novelty to arrive at an overall score for each system. This way, we are able to rank the systems. Lastly, we identified future research directions. We point out some existing gaps and challenges in the field of distributed GNN acceleration in hopes of inspiring future advancements.

IV. Distributed GNN Setup

GNNs can have millions of nodes and billions of edges. The current most popular technique is distributed GNN training. In a distributed GNN training setup, a large graph is partitioned into smaller subgraphs, each of which can be processed independently on separate machines or processors. This partitioning strategy allows for the parallel processing of graph data, significantly reducing the computational time and resources required for GNN training. However, there are challenges in matters of sampling, partitioning, communication, and memory management. In this section, we discuss the most recent and advanced methods and approaches.

A. Sampling

While training large GNNs, sampling, the process of selecting a subset of nodes or edges from a large graph to form a smaller subset, is a technique that is often used in the initial setup stages. Sampling is necessary to make the most efficient use of available resources and derive meaningful representations from vast amounts of data.

The main objectives of sampling include scalability, efficiency, and generalization. It allows scalability and efficiency by reducing the computational and memory demands, allowing for the more efficient utilization of computing resources. When done well, sampling accomplishes generalization by capturing a representative subset of the graph that maintains statistical properties and structural information. This ensures that the sampled subset reflects the characteristics of the entire graph.

Some examples of recent sampling methods include neighborhood, adaptive, randomized, and computation-aware sampling. Neighborhood sampling selects nodes based on their local connections within a graph. It aims to select immediate neighbors or the local information of each node to allow localized analysis. By sampling the neighborhood around each node, the computational complexity and memory requirements can be reduced while still retaining the essential structural information and capturing important relationships within the graph. This sampling strategy is used in the systems P^3 [1], DistDGL [18], and NeuGraph [11]. Another sampling method, adaptive sampling, samples based on node or edge importance or relevance to the learning task. The sampling strategy is adjusted during training based on the evolving characteristics and information learned from the graph. Thus, more resources are allocated towards more informative nodes critical for accurate predictions while fewer are used towards less important areas. This strategy is used in DistDGL and ZIPPER [17]. ZIPPER also uses randomized sampling which allows for more representative subgraphs. PaGraph [8] uses computation-aware sampling which maximizes the reuse of intermediate computation results. It dynamically selects a subset of nodes based on their expected computational workload in order to balance the load across its distributed system.

Although there are many different sampling strategies that are useful in varying degrees based on the input graph and overall model, sampling is a great contribution to the effort of accelerating large GNNs due to its ability to reduce computational complexity and memory.

B. Partitioning

A necessary aspect of accelerating distributed GNNs is partitioning. Partitioning is the process of dividing a graph into smaller subgraphs or partitions that can be processed independently by distributed computing resources. It is essential for scalability and load balancing.

Partitioning enables the parallel processing of subgraphs. This reduces the computational and memory requirements, making it feasible to train large graphs that a single machine cannot handle. Thus, partitioning allows scalability. Additionally, partitioning ensures load balancing which is when the computational workload is evenly distributed across the distributed system. Careful partitioning allows each machine

Name	Date	Description	Performance
NeuGraph [11]	Jul 2019	Scales to multiple GPUs and optimizes partitioning, scheduling, pipelining, & transfers.	N/A
AGL [16]	Mar 2020	A scalable, fault-tolerant, and integrated system, addressing challenges related to data dependency in graphs and the limitations of existing systems.	2.5-3.5
PaGraph [8]	Oct 2020	Incorporates a lightweight caching policy that considers both graph structural information and data access patterns and implements a fast GNN-computation-aware partition algorithm for scaling out on multiple GPUs.	4.8
ROC [3]	Feb 2020	Captures long-range dependencies, allows efficient parallel execution, leverages a hierarchical sampling strategy and optimizes the computation and communication costs.	N/A
ZIPPER [17]	Jul 2021	Exploits tile- and operator-level parallelism by combining fine-grained intra-kernel communication-computation pipelining with operator-level parallelism.	1.62-2.36
P^3 [1]	Jul 2021	Pipelined push-pull parallelism based execution strategy for fast model training of large GNNs.	3-7
DistDGL [18]	Aug 2021	A distributed training framework for DGL.	N/A
DistGNN [12]	Nov 2021	Optimizes DGL for full-batch training on CPU clusters through efficient shared memory implementation, communication reduction using minimum vertex-cut graph partitioning, and communication avoidance with delayed-update algorithms.	3.7
BGL [9]	Dec 2021	Incorporates a dynamic cache engine to minimize feature retrieving traffic, improves the graph partition algorithm to reduce cross-partition communication during subgraph sampling, and implements resource isolation to reduce contention between different data preprocessing stages.	7
SALINET [4]	Mar 2022	A distributed multi-GPU approach with neighborhood sampling.	1.5
DistDGLv2 [19]	Aug 2022	Trains GNNs on massive and heterogeneous graphs in a mini-batch fashion using distributed hybrid CPU/GPU training.	2-3
MGG [15]	May 2023	Accelerates GNNs with fine-grained intra-kernel communication-computation pipelining on multi-GPU platforms.	4.41

TABLE I

DESCRIPTION OF THE SYSTEMS AND THEIR PERFORMANCES (PERFORMANCE WAS SPEED COMPARED TO BASELINE DGL. N/A IS FOR SYSTEMS THAT DID NOT COMPARE SPEED WITH DGL.)

or processing unit to handle an equal portion of the graph, balancing the computations and optimizing resource utilization. This helps prevent bottlenecks and maximizes the efficiency of distributed GNN training.

Some partitioning methods include label-aware and hash-based partitioning. AGL [16] partitions large graphs into subgraphs based on the labels or attributes of nodes. Its goal is to group together nodes with similar labels so that they are in the same partition. This enhances locality and improves the efficiency of learning computations by reducing inter-partition communication. Hash-based partitioning partitions a graph across multiple machines based on a hash function. Each vertex or edge is assigned to a machine by computing the hash value of its identifier. This allows elements with the same identifier to consistently be assigned to the same machine. This is a simple and efficient partitioning method that ensures load balancing due to even distributions based on hash values. Some frameworks that use hash-based partitioning include P^3 [1], PaGraph [8], and ZIPPER [17].

Overall, partitioning is a necessary step in training distributed GNNs because it enables parallel processing. Differing methods have different pros and cons depending on graph elements and computational complexity, so partitioning schemes must be carefully selected.

C. Inter-Process Communication & Data Transferring

In this section, we discuss the crucial aspects of communication and data transfer between the original graph and the worker nodes in a distributed GNN training setup. After a graph has been partitioned, the data associated with each subgraph needs to be transferred to the corresponding worker nodes. Following the computation step, the gradients are

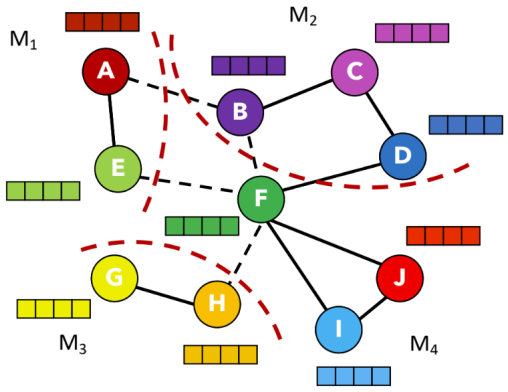


Fig. 1. Graph partitioning

computed and then pushed back to update the model in the main graph. We can divide systems into two based on where they store their main graph: centralized and decentralized. Centralized methods save information in parameter servers that are accessible by all the workers ([7], [6], [16]). Instead of parameter servers, some models, such as GraphLab [10], PowerGraph [2], DistDGL, and DistGNN, use shared memory, or decentralized architectures. Centralized methods use push or pull techniques to update the main graph but for the older versions, there was no communication ability between workers until P^3 was introduced. In the push system, machines upload their gradient values to the main graph. In the pull system, the main machine sends the updates to each machine. Neither of them has intercommunication between worker machines. Only after distributing the models and data can computation be performed. The computation may occur in a synchronous,

asynchronous, or hybrid manner. In the synchronous method, workers must wait for the other machines to complete their tasks and upload their progress. A major drawback of this type of system is that faster tasks have to wait for slower ones. To overcome this, systems either distribute the tasks in a balanced manner so that they can end at the same time, or they work in an asynchronous way. One well-known implementation of the synchronous update is bulk synchronous parallel (BSP) [13]. The BSP model uses the boss/worker paradigm and represents problem data in the form of lists, which simplifies the logic of building applications. AGL [16] is a synchronous system containing a module named GraphFlat. GraphFlat arranges the workload in an efficient way to reduce sizes, putting related nodes' calculations in sequence to reduce the duration of memory transfer. Additionally, it fills up the memory of working nodes so that they can finish tasks at the same time. PaGraph [8] also arranges the workload of the nodes, and uses a computation-aware partition algorithm to give the same amount of tasks to machines. DistDGL [18] and DistDGLv2 [19] use the METIS [5] partitioning algorithm to equalize the memory and processing needs. Synchronous methods can be utilizable for static data but asynchronous methods are also very promising with their schedulers. NeuGraph [11] employs a scheduler that filters out unnecessary data transfer and strategically assigns tasks that utilize the same data to the same nodes. ROC [3] achieves balanced workload distribution under the cost model calculation, the model basically calculates the number of learnable parameters and minimum data need to be transferred to memory. BGL [9] dynamically changes the data in the machines and names the module dynamic cache engine. This allows it to reduce the underutilization of GPUs. Some models just follow the working machines and adaptively assign tasks (DistGNN [12], SALIENT [4], ZIPPER [17]) because they trust their data-transferring ability while processing.

V. Distributed Neural Network Training

Distributed neural network training is a powerful approach that enables the efficient and scalable training of large GNN models by leveraging parallelism and pipelining techniques. The combination of the two achieves improved scalability, reduced training time, and enhanced utilization of computing resources.

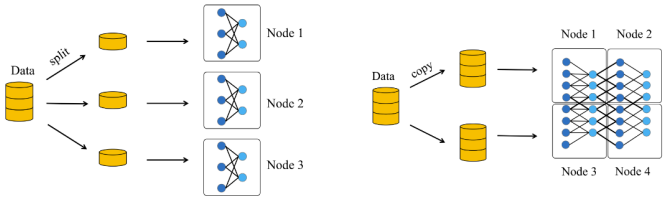
A. Parallelism

Parallelism involves dividing the training process across multiple computing resources, such as GPUs or machines, allowing for simultaneous computations to be computed on different subsets of data. This parallel execution significantly reduces training time by distributing the workload and leveraging the computational power of multiple devices.

Two types of parallelism are data and model parallelism as shown in figure 2. In data parallelism, the data is partitioned into subsets. Then, multiple copies of the model are created so that each can train a different subset. Thus, each copy computes gradients independently and then shares them with the other copies during gradient synchronization. Data parallelism allows for efficient scaling to larger batch sizes, faster convergence, and increased training throughput. Model parallelism divides the model across multiple machines. Then, the data is copied to each machine which then computes a portion of the model. This is a useful technique when the model's size exceeds the memory capacity of a single device or when some operations within the model are computationally intensive and need to be offloaded to separate devices. While model parallelism enables the training of larger models and the efficient utilization of resources, a drawback is that it requires good communication between devices to ensure that the correct computations are performed and gradients are synchronized accurately.

Another alternative is to combine the two parallelism techniques in a hybrid manner. Hybrid parallelism combines the two to achieve a balance between training efficiency and model scalability. First, the system behaves in a data parallel manner, creating multiple copies of the model that each process a subset of the training data. Next, the system behaves in a model parallel manner by dividing each copy across multiple devices, each responsible for computing a certain part of the model. Hybrid parallelism combines the best of both worlds, allowing for efficient scaling to larger batch sizes and model size while effectively utilizing multiple devices or machines. It offers flexibility in adapting to the memory and computational constraints of the distributed system.

While hybrid parallelism seems to be the most advantageous, the choice of parallelism technique ultimately depends on factors such as model size, memory capacity, computational requirements, and the distributed system's architecture. Parallelism helps speed up GNN training greatly. However, unlike in deep neural networks, parallelism has limitations in GNNs because the information is connected. Thus, further steps such as scheduling are necessary.



(a) Data Parallelism (b) Model Parallelism

Fig. 2. Data parallelism vs. model parallelism

B. Pipeline of Training

The acceleration of system performance following graph parallelization can be achieved through a combination of communication and memory management strategies. These strategies collectively contribute to enhancing the speed and efficiency of graph parallelized systems, enabling them to handle larger and more complex tasks.

NeuGraph [11] adeptly partitions the graph data, including vertex and edge data, into subgraphs known as chunks. It

Name	Sampling	Partitioning	Synchronization	Coordination
NeuGraph	Neighborhood	METIS	Asynchronous	Centralized
AGL	Neighborhood	Label-aware	Synchronous	Centralized
PaGraph	Computation-aware	Hash	Synchronous	Centralized
ROC	Oversampling	Online Linear	Asynchronous	Centralized
ZIPPER	Randomized Adaptive	Hash	Asynchronous	Centralized
P^3	Neighborhood	Hash	Asynchronous	Centralized
DistDGL	Neighborhood Adaptive	METIS	Synchronous	Distributed
DistGNN	N/A	Minimum vertex-cut	Asynchronous	Distributed
BGL	Neighborhood	BGL partitioning	Asynchronous	Distributed
SALIENT	Neighborhood	Slicing	Asynchronous	Distributed
DistDGLv2	Neighborhood	Min-edge cut	Synchronous	Distributed
MGG	Neighborhood	Remote neighbor partitioning	Asynchronous	Distributed

TABLE II
THE MODEL'S SAMPLING, PARTITIONING, SYNCHRONIZATION, COORDINATION

then constructs a dataflow that manages the operations and transfers of these chunks. Furthermore, NeuGraph schedules parallel executions of the dataflow on GPUs and facilitates the transfer of chunks between neighboring GPUs. AGL [16] implements pruning and edge-partitioning for pre-processing to reduce memory needs. In the training section, there are model computation and subgraph vectorization tasks which differ AGL from other machine learning models and these processes are pipelined to increase the speed. It uses mapping to describe the data to prevent the repetition of embedding inference and reduces time cost. In the GPU cache of PaGraph [8], vertex features are managed by maintaining two separate spaces: one for feature data organized as large matrices, and another for vertex metadata organized into a hash table for fast lookup, significantly optimizing the retrieval process. ZIPPER [17] chooses to tile the subpartitions and pipelines the processing and transferring of data like many other models such as MGG [15], BGL [9], and SALIENT [4]. P^3 [1] is one of the best models that pipelines the computation process because the machines have communication ability. When we analyze the methods, they are concentrated on reducing the data size and synchronizing transition and processing.

VI. DISCUSSION

While there have been significant advancements in aspects of memory and communication regarding GNNs, the focus on Neural Architecture Search (NAS) for GNNs is still relatively limited and warrants further attention. Currently, most systems are evaluated on static graphs. However, dynamic graphs hold considerable potential, especially for real-time tasks. Additionally, the advent of deeper GNN architectures and the variability in feature vector sizes present new challenges that need to be addressed. The P^3 model has made a notable contribution by proposing simultaneous data and model parallelism. However, this approach is still in its evolving, and there is plenty of room for further innovation and discovery in this direction.

VII. CONCLUSION

In conclusion, this paper provides a comprehensive analysis and comparison of recent advancements in distributed GNN acceleration strategies. Our contributions include an overview of new systems, classification based on acceleration mechanisms, performance comparisons, a new ranking of the frameworks, and identification of future research directions. Through our analysis, we find that BGL, P^3, and MGG are the current best systems for distributed GNN acceleration. However, we also identify several gaps and challenges in the field, such as the need for further exploration of NAS for GNNs, the consideration of dynamic graphs, and the challenges posed by deeper GNN architectures and variable feature vector sizes. Addressing these challenges will require innovative partitioning, communication, and synchronization strategies to ensure the efficient utilization of distributed resources.

In summary, our work contributes to the understanding of distributed GNN acceleration, highlights the top-performing systems, and outlines future research directions. We hope that our findings inspire researchers to explore new approaches and overcome existing challenges, ultimately advancing the field of distributed GNN acceleration and enabling efficient learning on large-scale graph datasets.

Name	Distributability	Speed	Memory	Novelty	Overall Score
NeuGraph	2	1	3	5	11
AGL	2	3	4	2	11
PaGraph	2	4	4	2	12
ROC	3	2	4	2	11
ZIPPER	3	2	3	2	10
P^3	2	5	3	5	15
DistDGL	4	1	2	1	8
DistGNN	2	3	1	2	8
BGL	4	5	5	4	18
SALINET	4	1	2	3	10
DistDGLv2	4	3	2	2	11
MGG	5	4	4	2	15

TABLE III
THE SCORES OF THE MODELS

A. Scoring The Models

In SoK research, it is important to compare systems analytically, which is distinct from other types of papers. However, comparing aspects such as speed, memory, parallelism, innovation, and compatibility can be challenging. These comparisons may not yield accurate results unless the systems are evaluated on the same dataset and computer. Even then, using a single computer and dataset can lead to misleading comparisons due to the unique advantages each system has in terms

of platform and data storage model. Therefore, making an accurate comparison is not straightforward.

In Table III, we attempt to compare the systems based on the innovations they introduced and the numerical values they provided. We also conducted a general evaluation over other evaluation metrics in cases where data was insufficient.

For memory usage, a score of 1 indicates the highest memory usage, while a score of 5 signifies the most efficient memory usage. In terms of speed, a score of 1 is given to the slowest system and a score of 5 to the fastest. For innovation, a score of 1 is given to systems that merely combine existing systems, while a score of 5 is awarded to those introducing new systems. Distributability is scored based on the system's methodologies for sampling, partitioning, synchronization, and coordination. A score of 1 is given to systems that use common, outdated, or inefficient techniques, and a score of 5 is given to systems that successfully distribute tasks using methodologies such as distributed coordination rather than centralized, asynchronously and effectively train, and have thoughtful sampling and partitioning strategies rather than basic or outdated.

REFERENCES

[1] Swapnil Gandhi and Anand Padmanabha Iyer. "P3: Distributed Deep Graph Learning at Scale." In: *OSDI*. 2021, pp. 551–568.

[2] Joseph E Gonzalez et al. "Powergraph: Distributed graph-parallel computation on natural graphs". In: *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 2012, pp. 17–30.

[3] Zhihao Jia et al. "Improving the accuracy, scalability, and performance of graph neural networks with roc". In: *Proceedings of Machine Learning and Systems* 2 (2020), pp. 187–198.

[4] Tim Kaler et al. "Accelerating training and inference of graph neural networks with fast sampling and pipelining". In: *Proceedings of Machine Learning and Systems* 4 (2022), pp. 172–189.

[5] George Karypis and Vipin Kumar. "A fast and high quality multilevel scheme for partitioning irregular graphs". In: *SIAM Journal on scientific Computing* 20.1 (1998), pp. 359–392.

[6] Mu Li et al. "Communication efficient distributed machine learning with the parameter server". In: *Advances in Neural Information Processing Systems* 27 (2014).

[7] Mu Li et al. "Scaling distributed machine learning with the parameter server". In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 583–598.

[8] Zhiqi Lin et al. "Pagraph: Scaling gnn training on large graphs via computation-aware caching". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 401–415.

[9] Tianfeng Liu et al. *BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing*. 2021. arXiv: 2112.08541 [cs.LG].

[10] Yucheng Low et al. "Graphlab: A new framework for parallel machine learning". In: *arXiv preprint arXiv:1408.2041* (2014).

[11] Lingxiao Ma et al. "NeuGraph: Parallel Deep Neural Network Computation on Large Graphs." In: *USENIX Annual Technical Conference*. 2019, pp. 443–458.

[12] Vasimuddin Md et al. "Distgnn: Scalable distributed training for large-scale graph neural networks". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–14.

[13] Leslie G Valiant. "A bridging model for parallel computation". In: *Communications of the ACM* 33.8 (1990), pp. 103–111.

[14] Minjie Wang et al. *Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks*. 2020. arXiv: 1909.01315 [cs.LG].

[15] Yuke Wang et al. *MGG: Accelerating Graph Neural Networks with Fine-grained intra-kernel Communication-Computation Pipelining on Multi-GPU Platforms*. 2023. arXiv: 2209.06800 [cs.DC].

[16] Dalong Zhang et al. *AGL: a Scalable System for Industrial-purpose Graph Machine Learning*. 2020. arXiv: 2003.02454 [cs.SI].

[17] Zhihui Zhang et al. *ZIPPER: Exploiting Tile- and Operator-level Parallelism for General and Scalable Graph Neural Network Acceleration*. 2021. arXiv: 2107.08709 [cs.AR].

[18] Da Zheng et al. *DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs*. 2021. arXiv: 2010.05337 [cs.LG].

[19] Da Zheng et al. "Distributed hybrid CPU and GPU training for graph neural networks on billion-scale heterogeneous graphs". In: *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2022, pp. 4582–4591.