A Comprehensive Survey on Distributed Training of Graph Neural Networks

By Haiyang Lin, Mingyu Yan, Xiaochun Ye, Dongrui Fan, *Senior Member, IEEE*,
Shirui Pan, Wenguang Chen, and Yuan Xie, *Fellow, IEEE*

Abstract—**Graph neural networks (GNNs) have been demonstrated to be a powerful algorithmic model in broad application fields for their effectiveness in learning over graphs. To scale GNN training up for large-scale and ever-growing graphs, the most promising solution is distributed training which distributes the workload of training across multiple computing nodes. However, the workflows, computational patterns, communication patterns, and optimization techniques of distributed GNN training remain preliminarily understood. In this paper, we provide a comprehensive survey of distributed GNN training by investigating various optimization techniques used in distributed GNN training. First, distributed GNN training is classified into several categories according to their workflows. In addition, their computational patterns and communication patterns, as well as the optimization techniques proposed by recent work are introduced. Second, the software frameworks and hardware platforms of distributed GNN training are also introduced for a deeper understanding. Third, distributed GNN training is compared with distributed training of deep neural networks, emphasizing the uniqueness of distributed GNN training. Finally, interesting issues and opportunities in this field are discussed.**

Index Terms—**Graph learning, graph neural network, distributed training, workflow, computational pattern, communication pattern, optimization technique, software framework.**

I. Introduction

GRAPH is a well-known data structure widely used in many critical application fields due to its powerful representation capability of data, especially in expressing the associations between objects [1], [2]. Many real-world data can be naturally represented as graphs which consist of a set of vertices and edges. Take social networks as an example [3], [4], the vertices in the graph represent people and the edges represent interactions between people on Facebook [5]. An illustration of graphs for social networks is illustrated in Fig. 1 (a), where the circles represent the vertices, and the arrows represent the edges. Another well-known example is

Haiyang Lin, Mingyu Yan, Xiaochun Ye, and Dongrui Fan are with State Key Lab of Processors (SKLP), Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing 100864, China and also with the University of Chinese Academy of Sciences (UCAS), Beijing 100049, China (e-mail: linhaiyang18z@ict.ac.cn; yanmingyu@ict.ac.cn; yexiaochun@ict.ac.cn; fandr@ict.ac.cn).

Shirui Pan is with the School of Information and Communication Technology, Griffith University, Queensland 4111, Australia (e-mail: s.pan@griffith.edu.au).

Wenguang Chen is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: cwg@tsinghua.edu.cn).

Yuan Xie is with the Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA 93106 USA (e-mail: yuanxie@ucsb.edu).

Corresponding author: Mingyu Yan

knowledge graphs [6], [7], in which the vertices represent entities while the edges represent relations between the entities [8].

Graph neural networks (GNNs) demonstrate superior performance compared to other algorithmic models in learning over graphs [9]–[11]. Deep neural networks (DNNs) have been widely used to analyze Euclidean data such as images [12]. However, they have been challenged by graph data from the non-Euclidean domain due to the arbitrary size and complex topological structure of graphs [13]. Besides, a major weakness of deep learning paradigms identified by industry is that they cannot effectively carry out causal reasoning, which greatly reduces the cognitive ability of intelligent systems [14]. To this end, GNNs have emerged as the premier paradigm for graph learning and endowed intelligent systems with cognitive ability. An illustration for GNNs is shown in Fig. 1 (b). After getting graph data as input, GNNs use forward propagation and backward propagation to update the model parameters. Then the trained model can be applied to graph tasks, including vertex prediction [15] (predicting the properties of specific vertices), link prediction [16] (predicting the existence of an edge between two vertices), and graph prediction [17] (predicting the properties of the whole graph), as shown in Fig. 1 (c).

Thanks to the superiority of GNNs, they have been widely used in various real-world applications in many critical fields. These real-world applications include knowledge inference [18], natural language processing [19], [20], machine translation [21], recommendation systems [22]–[24], visual reasoning [25], chip design [26]–[28], traffic prediction [29]–[31], ride-hailing demand forecasting [32], spam review detection [33], molecule property prediction [34], and so forth. GNNs enhance the machine intelligence when processing a broad range of real-world applications, such as giving >50% accuracy improvement for real-time ETAs in Google Maps [29], generating >40% higher-quality recommendations in Pinterest [22], achieving >10% improvement of ride-hailing demand forecasting in Didi [32], improving >66.90% of *recall at 90% precision* for spam review detection in Alibaba [33].

However, both industry and academia are still eagerly expecting the acceleration of GNN training for the following reasons [35]–[38]:

- *The scale of graph data is rapidly expanding, consuming a great deal of time for GNN training.* With the explosion of information on the Internet, new graph data are constantly being generated and changed, such as the establishment and demise of interpersonal relationships
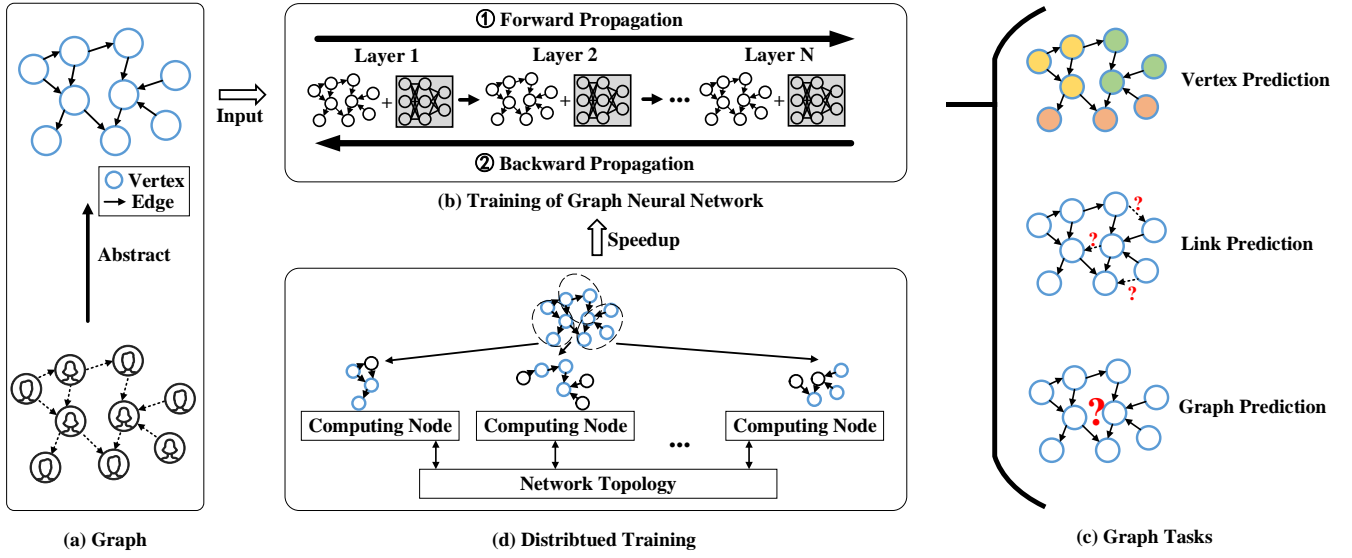
Fig. 1. Distributed GNN training: (a) Illustration of graphs; (b) Illustration of training; (c) Illustration of graph tasks; (d) Illustration of distributed training.

in social communication and the changes in people's preferences for goods in online shopping. The scales of vertices and edges in graphs are approaching or even outnumbering the order of billions and trillions, respectively [39]–[42]. The growth rate of graph scales is also astonishing. For example, the number of vertices (i.e., users) in Facebook's social network is growing at a rate of 17% per year [43]. Consequently, GNN training time dramatically increases due to the ever-growing scale of graph data.

- *Swift development and deployment of novel GNN models involves repeated training, in which a large amount of training time is inevitable.* Much experimental work is required to develop a highly-accurate GNN model since repeated training is needed [9]–[11]. Moreover, expanding the usage of GNN models to new application fields also requires much time to train the model with real-life data. Such a sizeable computational burden calls for faster methods of training.

Distributed training is a popular solution to speed up GNN training [35]–[38], [40], [44]–[58]. It tries to accelerate the entire computing process by adding more computing resources, or "nodes", to the computing system with parallel execution strategies, as shown in Fig. 1 (d). NeuGraph [44], proposed in 2019, is the first published work of distributed GNN training. Since then, there has been a steady stream of attempts to improve the efficiency of distributed GNN training in recent years with significantly varied optimization techniques, including workload partitioning [44]–[47], transmission planning [37], [44]–[46], caching strategy [35], [51], [52], etc.

Despite the aforementioned efforts, there is still a dearth of review of distributed GNN training. The need for management and cooperation among multiple computing nodes leads to a different workflow, resulting in complex computational and communication patterns, and making it a challenge to optimize distributed GNN training. However, regardless of plenty of

efforts on this aspect have been or are being made, there are hardly any surveys on these challenges and solutions. Current surveys mainly focus on GNN models and hardware accelerators [9]–[11], [59]–[62], but they are not intended to provide a careful taxonomy and a general overview of distributed training of GNNs, especially from the perspective of the workflows, computational patterns, communication patterns, and optimization techniques.

This paper presents a comprehensive review of distributed training of GNNs by investigating its various optimization techniques. First, we summarize distributed GNN training into several categories according to their workflows. In addition, we introduce their computational patterns, communication patterns, and various optimization techniques proposed in recent works to facilitate scholars to quickly understand the principle of recent optimization techniques and the current research status. Second, we introduce the prevalent software frameworks and hardware platforms for distributed GNN training and their respective characteristics. Third, we emphasize the uniqueness of distributed GNN training by comparing it with distributed DNN training. Finally, we discuss interesting issues and opportunities in this field.

Our main goals are as follows:

- Introducing the basic concepts of distributed GNN training.
- Analyzing the workflows, computational patterns, and communication patterns of distributed GNN training and summarizing the optimization techniques.
- Highlighting the differences between distributed GNN training and distributed DNN training
- Discussing interesting issues and opportunities in the field of distributed GNN training.

The rest of this paper is described in accordance with these goals. Its organization is shown in Table I and summarized as follows:

- Section II introduces the basic concepts of graphs and

GNNs as well as two training methods of GNNs: full-batch training and mini-batch training.

- Section III introduces the taxonomy of distributed GNN training and makes a comparison between them.
- Section IV introduces distributed full-batch training of GNNs and further categorizes it into two types, of which the workflows, computational patterns, communication patterns, and optimization techniques are introduced in detail.
- Section V introduces distributed mini-batch training and classifies it into two types. We also present the workflow, computational pattern, communication pattern, and optimization techniques for each type.
- Section VI introduces software frameworks currently supporting distributed GNN training and describes their characteristics.
- Section VII introduces hardware platforms for distributed GNN training.
- Section VIII highlights the uniqueness of distributed GNN training by comparing it with distributed DNN training.
- Section IX summarizes the distributed full-batch training

and distributed mini-batch training, and discusses several interesting issues and opportunities in this field.
- Section X is the conclusion of this paper.

Euclidean Data
(a)

Non-Euclidean Data
(b)
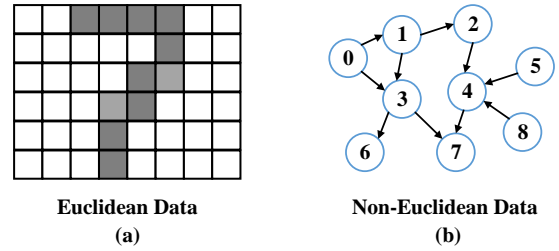
Fig. 2. Illustrations of Euclidean and non-Euclidean data: (a) Image from Euclidean space; (b) Graph data from non-Euclidean space.

II. BACKGROUND

This section provides some background concepts of graphs as well as GNNs, and introduces the two training methods applied in GNN training: full-batch and mini-batch.

A. Graphs

A graph is a type of data structure consisting of vertices and edges. Its flexible structure can effectively express the
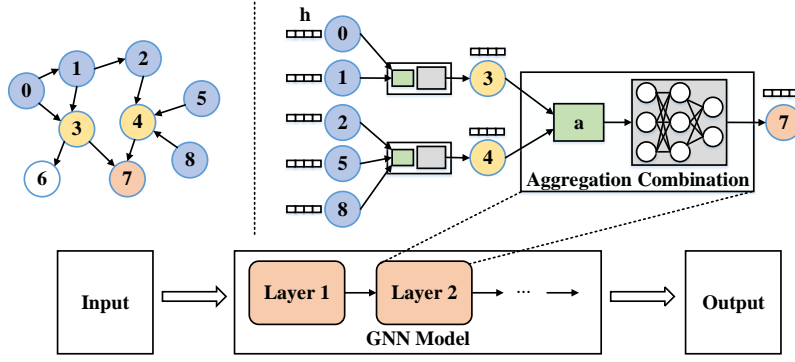
Fig. 3. Model architecture of GNNs. h represents h_v^l, the representation feature of vertex v at the l-th layer. a represents a_v^l, the aggregation result of vertex v at the l-th layer.

relationship among a set of objects with arbitrary size and undetermined structure, which is called non-Euclidean data. As shown in Fig. 2 (a) and (b), the form of non-Euclidean data is not as highly structured as Euclidean data. However, by using vertices to represent the objects and edges to represent relationships among the objects, graphs can effectively represent non-Euclidean data, such as social networks and knowledge graphs.

There are mainly three taxonomies of graphs:

- **Directed/Undirected Graphs:** Every edge in a directed graph has a fixed direction, indicating that the connection is only from a source vertex to a destination vertex. However, in an undirected graph, the connection represented by an edge is bi-directional between the two vertices. An undirected graph can be transformed into a directed one, in which two edges in the opposite direction represent an undirected edge in the original graph.
- **Homogeneous/Heterogeneous Graphs:** Homogeneous graphs contain a single type of vertex and edge, while heterogeneous graphs contain multiple types of vertices and multiple types of edges. Thus, heterogeneous graphs are more powerful in expressing the relationships between different objects.
- **Static/Dynamic Graphs:** The structure and the feature of static graphs are always unchanged, while those of dynamic graphs can change over time. A dynamic graph can be represented by a series of static graphs with different timestamps.

B. Graph Neural Networks

GNNs have been dominated to be a promising algorithmic model for learning knowledge from graph data [63]–[68]. It takes the graph data as input and learns a representation vector for each vertex in the graph. The learned representation can be used for down-stream tasks such as vertex prediction [15], link prediction [16], and graph prediction [17].

As illustrated in Fig. 3, a GNN model consists of one or multiple layers consisting of neighbor aggregation and neural network operations, referred to as the Aggregation step and Combination step, respectively. In the Aggregation step, the Aggregate function $Aggregate(\ )$ is used to aggregate the

feature vectors of in-coming neighboring vertices from the previous GNN layer for each target vertex. For example, in Fig. 3, vertex 4 would gather the feature vectors of itself and its incoming neighboring vertices (i.e., vertex 2, 5, 8) using the Aggregate function. In the Combination step, the Combine function $Combine(\ )$ transforms the aggregated feature vector of each vertex using neural network operations. To sum up, the aforementioned computation on a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ can be formally expressed by

$$a_v^l = Aggregate(\{h_u^{l-1} | u \in N(v) \cup \{v\}\}) \tag{1}$$
$$h_v^l = Combine(a_v^l) \tag{2}$$

where h_v^l denotes the feature vector of vertex v at the l-th layer, and $N(v)$ represents the neighbors of vertex v. Specifically, the input feature of vertex $v \in \mathcal{V}$ is denoted as h_v^0.

C. Training Methods for Graph Neural Networks

In this subsection, we introduce the training methods for GNNs, which are approached in two ways including full-batch training [69], [70] and mini-batch training [13], [71]–[74].

A typical training procedure of neural networks, including GNNs, includes forward propagation and backward propagation. In forward propagation, the input data is passed through the layers of neural networks towards the output. Neural networks generate differences of the output of forward propagation by comparing it to the predefined labels. Then in backward propagation, these differences are propagated through the layers of neural networks in the opposite direction, generating gradients to update the model parameters.

As illustrated in Fig. 4, training methods of GNNs can be classified into full-batch training [69], [70] and mini-batch training [13], [71]–[74], depending on whether the whole graph is involved in each **round**. Here, we define a **round** of full-batch training consisting of a model computation phase, including forward and backward propagation, and a parameter update phase. On the other hand, a **round** in mini-batch training additionally contains a sampling phase, which samples a small-sized workload required for the subsequent model computation and thus locates prior to the other two phases. Thus, an **epoch**, which is defined as an entire pass of the data, is equivalent to a round of full-batch training, while that
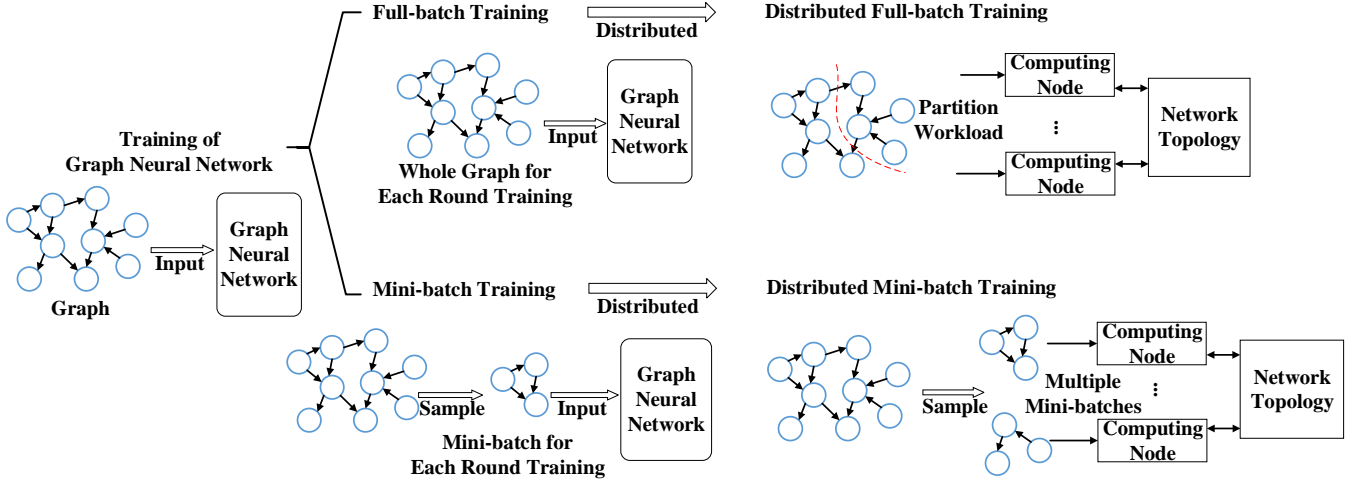
Fig. 4. Training methods for GNNs and their distributed implementations.

in mini-batch training usually contains several rounds. Details of these two methods are introduced below.

1) Full-batch Training: Full-batch training utilizes the whole graph to update model parameters in each round.

Given a training set $\mathcal{V}_t \subset \mathcal{V}$, the loss function of full-batch training is

$$\mathcal{L} = \frac{1}{|\mathcal{V}_t|} \sum_{v_i \in \mathcal{V}_t} \nabla l(y_i, z_i) \tag{3}$$

where $\nabla l(\ )$ is a loss function, y_i is the known label of vertex v_i, and z_i is the output of GNN model when inputting feature x_i of v_i. In each epoch, GNN model needs to aggregate representations of all neighboring vertices for each vertex in $\mathcal{V}_t$ all at once. As a result, the model parameters are updated only once at each epoch.

2) Mini-batch Training: Mini-batch training utilizes part of the vertices and edges in the graph to update model parameters in every forward propagation and backward propagation. It aims to reduce the number of vertices involved in the computation of one round to reduce the computing and memory resource requirements.

Before each round of training, a mini-batch $\mathcal{V}_s$ is sampled from the training dataset $\mathcal{V}_t$. By replacing the full training dataset $\mathcal{V}_t$ in equation (3) with the sampled mini-batch $\mathcal{V}_s$, we obtain the loss function of mini-batch training:

$$\mathcal{L} = \frac{1}{|\mathcal{V}_s|} \sum_{v_i \in \mathcal{V}_s} \nabla l(y_i, z_i) \tag{4}$$

It indicates that for mini-batch training, the model parameters are updated multiple times at each epoch, since numerous mini-batches are needed to have an entire pass of the training dataset, resulting in many rounds in an epoch. Stochastic Gradient Descent (SGD) [75], a variant of gradient descent which applies to mini-batch, is used to update the model parameters according to the loss $\mathcal{L}$.

Sampling: Mini-batch training requires a sampling phase to generate the mini-batches. The sampling phase first samples a set of vertices, called target vertices, from the training set

according to a specific sampling strategy, and then it samples the neighboring vertices of these target vertices to generate a complete mini-batch. The sampling method can be generally categorized into three groups: Node-wise sampling, Layer-wise sampling, and Subgraph-based sampling [56], [76].

- **Node-wise sampling** [13], [22], [77], [78] is directly applied to the neighbors of a vertex: the algorithm selects a subset of each vertex's neighbors. It is typical to specify a different sampling size for each layer. For example, in GraphSAGE [13], it samples at most 25 neighbors for each vertex in the first layer and at most 10 neighbors in the second layer.
- **Layer-wise sampling** [71], [72], [79] enhances Node-wise sampling. It selects multiple vertices in each layer and then proceeds recursively layer by layer.
- **Subgraph-based sampling** [73], [80]–[82] first partition the original graph into multiple subgraphs, and then samples the mini-batches from one or a certain number of them.

III. TAXONOMY OF DISTRIBUTED GNN TRAINING

This section introduces the taxonomy of distributed GNN training. As shown in Fig. 5, we firstly categorize it into distributed full-batch training and distributed mini-batch training, according to the training method introduced in Sec. II-C, i.e., whether the whole graph is involved in each round, and show the key differences between the two types. Each of the two types is classified further into two detailed types respectively by analyzing their workflows. This section introduces the first-level category, that is, distributed full-batch training and distributed mini-batch training, and makes a comparison between them. The second-level category of the two types will later be introduced in Sec. IV and Sec. V, respectively.

A. Distributed Full-batch Training

Distributed full-batch training is the distributed implementation of GNN full-batch training, as illustrated in Fig. 4.
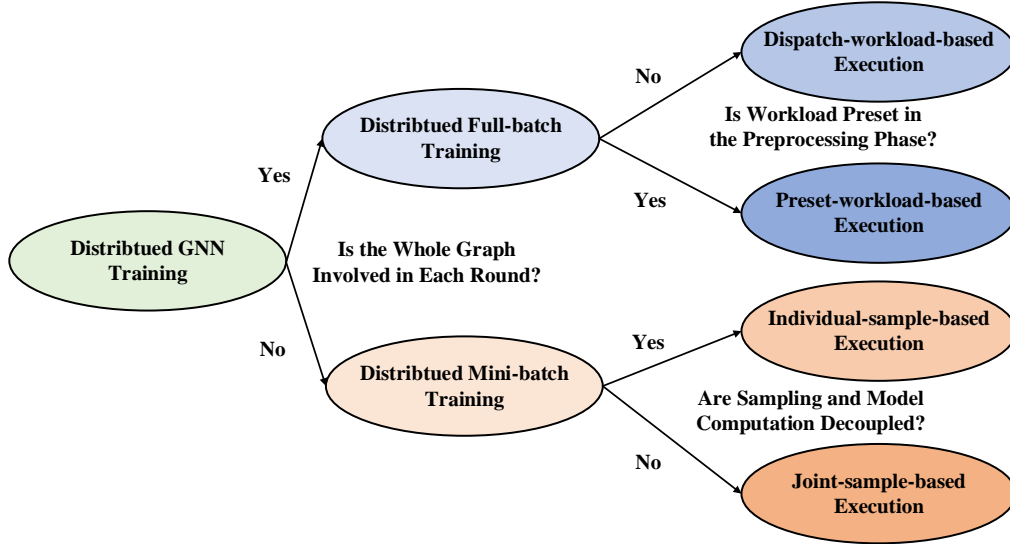
Fig. 5. Taxonomy of distributed GNN training.

Except for graph partition, a major difference is that multiple computing nodes need to synchronize gradients before updating model parameters, so that the models across the computing nodes remain unified. Thus, a round of distributed full-batch training includes two phases: model computation (forward propagation + backward propagation) and gradient synchronization. The model parameter update is included in the gradient synchronization phase.

Since each round involves the entire raw graph data, a considerable amount of computation and a large memory footprint are required in each round [37], [47], [50]. To deal with it, distributed full-batch training mainly adopts the workload partitioning method [44], [45]: split the graph to generate small workloads, and hand them over to different computing nodes.

Such a workflow leads to a lot of irregular inter-node communications in each round, mainly for transferring the features of vertices along the graph structure. This is because the graph data is partitioned and consequently stored in a distributed manner, and the irregular connection pattern in a graph, such as the arbitrary number and location of a vertices' neighbors. Therefore, many uncertainties exist in the communication of distributed full-batch training, including the uncertainty of the communication content, target, time, and latency, leading to challenges in the optimization of distributed full-batch training.

As shown in Fig. 5, we further classify distributed full-batch training more specifically into two categories according to whether the workload is preset in the preprocessing phase, namely dispatch-workload-based execution and preset-workload-based execution, as shown in the second column of Table III. Their detailed introduction and analysis are presented in Sec. IV-A and Sec. IV-B.

B. Distributed Mini-batch Training

Similar to distributed full-batch training, distributed mini-batch training is the distributed implementation of GNN mini-batch training as in Fig. 4. It also needs to synchronize gradients prior to model parameter update, so a round of distributed mini-batch training includes three phases: sampling, model computation, and gradient synchronization. The model parameter update is included in the gradient synchronization phase.

Distributed mini-batch training parallelizes the training process by processing several mini-batches simultaneously, one for each computing node. The mini-batches can be sampled either by the computing node itself or by other devices, such as another node specifically for sampling. Each computing node performs forward propagation and backward propagation on its own mini-batch. Then, the nodes synchronize and accumulate the gradients, and update the model parameters accordingly. Such a process can be formulated by

$$\mathcal{W}_{i+1} = \mathcal{W}_i + \sum_{j=1}^{n} \nabla g_{i,j} \tag{5}$$

where $\mathcal{W}_i$ is the weight parameters of model in the ith round of computation, $\nabla g_{i,j}$ is the gradients generated in the backward propagation of the computing node j in the ith round of computation, and the n is the number of the computing nodes.

As shown in Fig. 5, we further classify it more specifically into two categories according to whether the sampling and model computation are decoupled, namely individual-sample-based execution and joint-sample-based execution, as shown in the second column of Table III. Their detailed introduction and analysis are presented in Sec. V-A and Sec. V-B.

C. Comparison Between Distributed Full-batch Training and Distributed Mini-batch Training

This subsection compares distributed full-batch training with distributed mini-batch training of GNNs. The major

	Distributed Full-batch Training	Distributed Mini-batch Training
Workflow Characteristic	Collaborative Computation with Workload Partition	Independent Computation with Periodic Synchronization
Independence of Computation	Low	High
Involvement of Entire Graph?	Yes	No
Memory Capacity Requirement	High	Low
Communication Volume	Large	Small
Prime Communication Content	Features of Neighboring Vertices	Mini-batches
Communication Irregularity	High	Low
Communication Uncertainty	High	Low
Communication Bandwidth Requirement	High	Low
Additional Overhead	-	An Extra Phase (Sampling Phase)
Primary Challenges	Workload Imbalance and Massive Transmissions	Insufficient Sampling Performance

differences are also summarized in Table II.

The workflow of distributed full-batch training is summarized as collaborative computation with workload partition. Since the computation in each round involves the entire graph, the computing nodes need to cache it locally, leading to a high memory capacity requirement [44]. Also, the communication volume of distributed full-batch training is large [40], [47]. In every round, the Aggregate function needs to collect the features of neighbors for each vertex, causing a large quantity of inter-node communication requests since the graph is partitioned and stored on different nodes. Considering that the communication is based on the irregular graph structure, the communication irregularity of distributed full-batch training is high [46]. Another characteristic of communication is high uncertainty. The time of generating the communication request is indeterminate, since each computing node sends communication requests according to the currently involved vertices in its own computing process. As a result, the main challenges of distributed full batch training are workload imbalance and massive transmissions [45]–[47].

In contrast, the workflow of distributed mini-batch training is summarized as independent computation with periodic synchronization. The major transmission content is the mini-batches, sent from the sampling node (or component) to the computing node (or component) responsible for the current mini-batch [51], [52], [83]. As a result, these transmissions have low irregularity and low uncertainty, as the direction and content of transmission are deterministic. Since the computation of each round only involves the mini-batches, it triggers less communication volume and requires less memory capacity [51]. However, the extra sampling phase may cause some new challenges. Since the computation of the sampling phase is irregular and requires access to the whole graph for neighbor information of a given vertex, it is likely to encounter the problem of insufficient sampling performance, causing the subsequent computing nodes (or components) to stall due to lack of input, resulting in a performance penalty [56], [83].

D. Other Information of Taxonomy

Table III provides a summary of the current studies on distributed GNN training using our proposed taxonomy. Except for the aforementioned classifications, we also add some supplemental information in the table to provide a comprehensive review of them.

Software frameworks. The software frameworks used by the various studies are shown in the third column of Table III. PyTorch Geometric (PyG) [84] and Deep Graph Library (DGL) are the most popular among them. In addition, there are many newly proposed software frameworks aiming at distributed training of GNNs, and many of them are the optimization version of PyG [84] or DGL [85]. A detailed introduction to the software frameworks of distributed GNN training is presented in Sec. VI.

Hardware platforms. Multi-CPU platform and multi-GPU platform are the most common hardware platforms of distributed GNN training, as shown in the fourth column of Table III. Multi-CPU platform usually refers to a network with multiple servers, which uses CPUs as the only computing component. On the contrary, in multi-GPU platforms, GPUs are responsible for the major computing work, while CPU(s) handle some computationally complex tasks, such as workload partition and sampling. A detailed introduction to the hardware platforms is presented in Sec. VII.

Year. The contribution of distributed GNN training began to emerge in 2019 and is now showing a rapid growth trend. This is because more attention is paid to it due to the high demand from industry and academia to shorten the training time of GNN model.

Code available. The last column of Table III simply records the open source status of the corresponding study on distributed GNN training for the convenience of readers.

IV. DISTRIBUTED FULL-BATCH TRAINING

This section describes GNN distributed full-batch training in detail. Our taxonomy classifies it into two categories ac-

TABLE III
SUMMARY OF RECENT STUDIES ON DISTRIBUTED GNN TRAINING.

Name	Taxonomy	Software Framework	Hardware Platform	GPU Types	Year	Code Available
Distributed Full-batch Training						
NeuGraph [44]	Dispatch-workload-based Execution	NeuGraph	multi-GPU	Tesla P100	2019	-
Roc [45]	Dispatch-workload-based Execution	Roc	multi-GPU	Tesla P100	2020	Yes
CAGNET [40]	Preset-workload-based Execution	PyTorch	multi-GPU	Tesla V100	2020	Yes
DGCL [46]	Preset-workload-based Execution	PyTorch	multi-GPU	Tesla V100	2021	-
DistGNN [47]	Preset-workload-based Execution	DGL	multi-CPU	-	2021	Yes
FlexGraph [37]	Preset-workload-based Execution	FlexGraph	multi-CPU	-	2021	-
MG-GCN [48]	Preset-workload-based Execution	MG-GCN	multi-GPU	Tesla V100/A100	2021	Yes
Dorylus [49]	Preset-workload-based Execution	Dorylus	multi-CPU	-	2021	Yes
SAR [50]	Preset-workload-based Execution	DGL	multi-CPU	-	2021	-
Distributed Mini-batch Training						
AliGraph [35]	Joint-sample-based Execution	AliGraph	multi-CPU	-	2019	-
DistDGL [38]	Joint-sample-based Execution	DGL	multi-CPU	-	2020	-
AGL [36]	Individual-sample-based Execution	AGL	multi-CPU	-	2020	-
PaGraph [51]	Joint-sample-based Execution	DGL	multi-GPU	GTX 1080Ti	2020	-
2PGraph [52]	Joint-sample-based Execution	PyTorch	multi-GPU	RTX 3090	2021	-
LLCG [53]	Joint-sample-based Execution	PyG	multi-GPU	RTX 8000	2021	Yes
GraphTheta [55]	Joint-sample-based Execution	GraphTheta	multi-CPU	-	2021	-
SALIENT [56]	Individual-sample-based Execution	PyG	multi-GPU	Tesla V100	2021	-
P3 [57]	Joint-sample-based Execution	DGL	multi-GPU	Tesla P100	2021	-
DistDGLv2 [54]	Joint-sample-based Execution	DistDGLv2	multi-GPU	Tesla T4	2022	-
DGTP [58]	Individual-sample-based Execution	DGL	multi-GPU	GTX 1080Ti	2022	-

cording to whether the workload is preset in the preprocessing phase, namely dispatch-workload-based execution and preset-workload-based execution, as shown in Fig. 6 (a).

A. Dispatch-workload-based Execution

The dispatch-workload-based execution of distributed full-batch training is illustrated in Fig. 6 (b). Its workflow, computational pattern, communication pattern, and optimization techniques are introduced in detail as follows.

1) Workflow: In the dispatch-workload-based execution, a leader and multiple workers are used to perform training. The leader stores the model parameters and the graph data, and is also responsible for scheduling: it splits the computing workloads into chunks, distributes them to the workers, and collects the intermediate results sent from the workers. It also processes these results and advances the computation. Note that, the *chunk* we use here is as a unit of workload.

2) Computational Pattern: The computational patterns of forward propagation and backward propagation are similar in dispatch-workload-based execution: the latter can be seen as the reversed version of the former. As a result, we only introduce forward propagation's computational pattern here for simplicity. The patterns of the two functions in forward propagation (Aggregate and Combine) differ a lot and will be introduced below respectively.

Aggregate function. The computational pattern of the Aggregate function is dynamic and irregular, making workload partition for this step a challenge. In the Aggregation step, each vertex needs to aggregate the features of its own neighbors. As a result, the computation of the Aggregation step relies heavily on the graph structure, which is irregular or even changeable. Thus, the number and memory location of neighbors vary significantly among vertices, resulting in the dynamic and irregular computational pattern [86], causing the poor workload predictability and aggravating the difficulty of workload partition.

Combine function. The computational pattern of the Combine function is static and regular, thus the workload partition for it is simple. The computation of the Combination step is to perform neural network operations on each vertex. Since the structure of neural networks is regular and these operations share the same weight parameters, the Combination step enjoys a regular computational pattern. Consequently, a simple partitioning method is sufficient to maintain workload balance, so it is relatively not a major consideration in dispatch-workload-based execution of GNN distributed full-batch training.

3) Communication Pattern: The majority of communication is the transmission of input data and output results between the leader and the workers [44]. Since the leader is responsible for workload distribution and intermediate result collection, it needs to communicate with multiple workers. Such a communication structure results in a one-to-many communication pattern, resulting in a possible bottleneck in the leader's communication path [44]. However, these communications are relatively regular since the distribution of

(a) Categories of GNN Distributed Full-batch Training

(b) Illustration of Dispatch-workload-based Execution
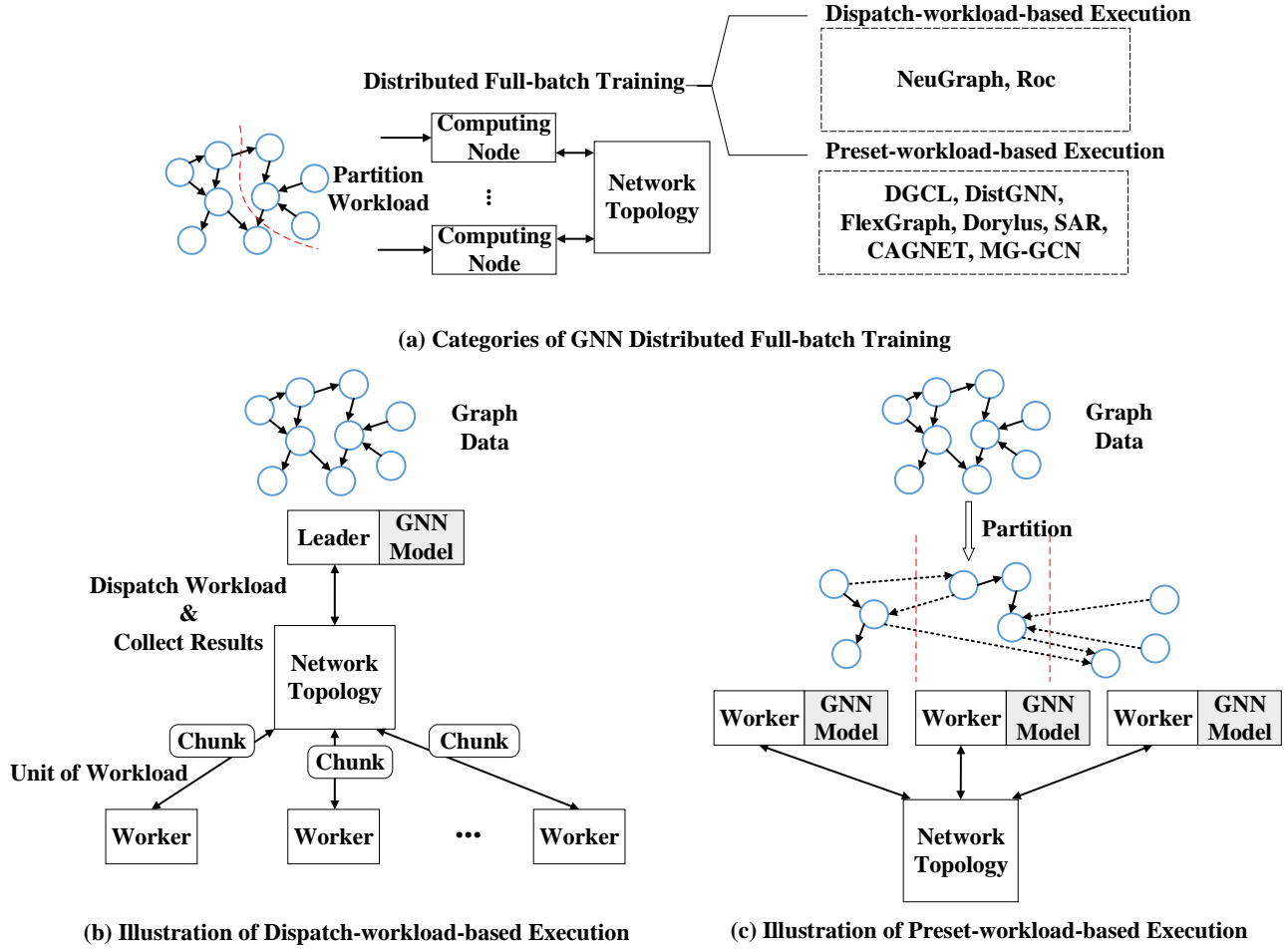

(c) Illustration of Preset-workload-based Execution

Fig. 6. GNN distributed full-batch training: (a) Categories of GNN distributed full-batch training; (b) Illustration of dispatch-workload-based execution; (c) Illustration of preset-workload-based execution.

tasks is controlled by the leader. Therefore, the communication path congestion can be avoided to some extent by optimized scheduling techniques.

4) Optimization Techniques: Next, we introduce the optimization techniques used to partition workload, balance workload, reduce transmission, and exploit parallelism for dispatch-workload-based execution. We classify them into four categories, including ❶ Vertex-centric Workload Partition, ❷ Balanced Workload Generation, ❸ Transmission Planning, and ❹ Feature-dimension Workload Partition. A summary for these categories is shown in Table IV. Note that the transmission reduction in this paper refers to a reduction in the amount of transmitted data or in transmission time.

❶ **Vertex-centric Workload Partition.** Vertex-centric workload partition refers to the technique of generating workload chunks by partitioning the graph or matrix from the perspective of vertices. Specifically, the graph is partitioned into a list of subgraphs according to the source vertex u and the destination vertex v of edge (u, v). Then the subgraphs are taken as the workload chunk, and the leader distributes them to each computing node for computation. This is a very common partitioning method for processing large-scale graphs in the traditional graph analytics [87]–[89].
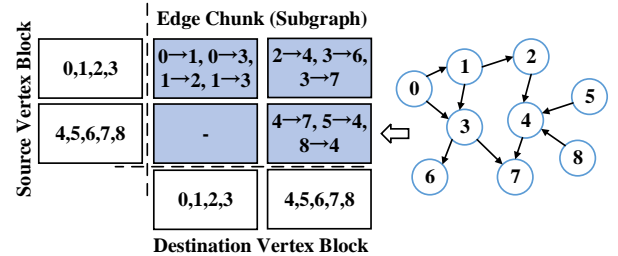
Fig. 7. Illustration of 2D graph partition.

Fig. 7 illustrates 2D graph partition, an typical example of graph partition. The vertices are firstly partitioned into P disjoint blocks. Then, we tile the edges into $P \times P$ chunks according to their source and destination vertices: in the yth chunk of the xth row, the source vertices of the edges all belong to x^{th} source vertex block, and the destination vertices all belong to y^{th} destination vertex block. This partitioning method works well for the Aggregation step, which needs to transfer the information of the source vertex to the destination vertex along the edge.

NeuGraph [44] adopts the 2D graph partitioning method. It chooses P as the minimum integer satisfying the requirement

TABLE IV

Optimization Technique	Function	Operation Object	Focus	Related Work
❶ Vertex-centric Workload Partition	Workload Partition	Graph or Matrix	Graph or Matrix Partition Strategy	NeuGraph [44], Roc [45]
❷ Balanced Workload Generation	Workload Balance	Partition Operation	Partition Strategy	Roc [45]
❸ Transmission Planning	Transmission Reduction	Transmission Operation	Transmission Strategy	Roc [45], NeuGraph [44]
❹ Feature-dimension Workload Partition	Parallelism Exploitation	Vertices' Feature	Feature Partition Strategy	NeuGraph [44]

to fit each chunk in the device memory of GPUs. Roc [45], on the other hand, uses the graph partition strategy proposed in [89], which can also be mentioned as 1D graph partition. Suppose there are now n computing nodes, and the number of vertices in the graph is T. Then select n-1 numbers, ranging from 1 to T, to split the vertices into n parts, that is, n subgraphs. In this way, each subgraph holds consecutively numbered vertices, which are stored in adjacent locations, and their in-edges, thereby improving the data access efficiency of computing nodes.

❷ **Balanced Workload Generation.** Workload balance is an extremely important optimization goal for dispatch-workload-based execution. Since the workloads are split and distributed to multiple computing nodes, the prerequisite of the continuation of computing is that all the computing nodes have already returned their intermediate results. If the workload is not evenly partitioned, the consequent lag of waiting will stall the training process. Therefore, it is necessary to carefully adjust the workload partition, so as to make the workload of each computing node as balanced as possible.

In response to that, Roc [45] proposes a linear regression cost model to produce balanced workloads in each round. The cost model is used to predict the computation time of a GNN layer on an arbitrary input, which could be the whole or any subset of an input graph.

❸ **Transmission Planning.** Planning data transmission is helpful to make full use of reusable data across computing components and nodes, thereby reducing data transmission. In dispatch-workload-based execution, the major transmission overhead is caused by the requirement to transmit the input data and intermediate results. Recent work focuses on harvesting two following optimization opportunities to reduce this overhead.

Avoid repeated transmission of overlapped data. The input data required by different computing tasks may overlap. As a result, caching these overlapped parts on computing nodes or components is a reasonable way to reduce transmission. Roc [45] formulates GPU memory management as a cost minimization problem. It uses a recursive dynamic programming algorithm to find the global optimal solution to decide which part of the data should be cached in GPU memory for reuse, according to the input graph, the GNN model, and device

information. This minimizes data transmission between the CPU and GPUs.

Rationally select the source of the transmission. The overhead of data transmission can be reduced by allowing computing components or nodes to send data, so every component or node can rationally select the source of the transmission. As a result, the design can reduce the overhead by decreasing the transmission distance. For instance, NeuGraph [44] employs a chain-based streaming scheduling scheme. The idea is to have one GPU (which already holds the data chunk) forward the data chunk to its neighbor GPU under the same PCIe switch, which can eliminate the bandwidth contention on the upper-level shared inter-connection link.

❹ **Feature-dimension Workload Partition.** Feature-dimension partition refers to the finer partition of the workload from the dimension of the vertex feature, to make full use of the parallel computing hardware in the computing node. In traditional graph analytics, the majority of the input is the structure data of graphs, such as edges. However, in GNNs, vertex features make up the majority of the dataset, while graph structure takes up only a tiny fraction of it. This is because the feature of each vertex in traditional graph analytics is usually just a scalar, while it is a long-length vector or even a large tensor in GNNs. For example, the feature dimension of a vertex can be 512, 1024, or even more [44], [83]. It is thus feasible to split the workload more finely in the feature dimension, thereby further improving the degree of computational parallelism.

To harvest the opportunity, NeuGraph [44] explores feature-level parallelism by splitting the feature of each vertex evenly into multiple parts, which partitions the workload in a more fine-grained way. It then assigns it to different GPU threads for corresponding computation. This method takes full advantage of the massive parallel computing resources of GPUs.

B. Preset-workload-based Execution

The preset-workload-based execution of distributed full-batch training is illustrated in Fig. 6 (c). Its workflow, computational pattern, communication pattern, and optimization techniques are introduced in detail as follows.

1) Workflow: Preset-workload-based execution involves multiple collaborative workers to perform training. The graph

is firstly split into subgraphs through the partition operation in the preprocessing phase. Then, each worker holds one subgraph and a replica of the model parameters. During training, each worker is responsible to complete the computing tasks of all the vertices in its subgraph. As a result, a worker needs to query the information from other workers to gather the information of neighboring vertices. However, the Combine function can be performed directly locally, as the model parameters are replicated, but this also means that gradient synchronization is required for each round to ensure the consistency of model parameters across the nodes.

2) Computational Pattern: The computational pattern of preset-workload-based execution differs significantly in different steps. In the Aggregation step, a node needs to query the neighbor information of vertices from other nodes, so only when each computing node cooperates efficiently can the data be supplied to the target computing node in a timely manner. However, in the Combination step, each computing node conducts the operation on the vertices in its own subgraph independently due to the local replica of model parameters. As a result, the Aggregation step is more prone to inefficiencies such as computational stagnation, and it is the key optimization point for preset-workload-based execution.

In addition, the preset workload has the benefit that the whole graph can be loaded at a time, as it is partitioned into small subgraphs which fit in the memory of computing nodes in the preprocessing phase. This means that it is more scalable than dispatch-workload-based execution when either adding more computing resources or increasing the size of the dataset. This avoids frequent accesses to graph data from low-speed storage such as hard disks, thereby ensuring the timely provision of data for high-speed computing.

3) Communication Pattern: Communication happens mostly in the Aggregation step and gradient synchronization. Due to the distributed storage of graph data, it encounters a lot of irregular transmissions during the Aggregation step when collecting the features of its neighboring vertices that are stored in other nodes due to graph partition [46]. Since the features of vertices are vectors or even tensors, the amount of data transmitted in the Aggregation step is large. Also, the communication is irregular due to the irregularity graph structure [86], [88], [90], which brings difficulty in optimizing connection. In contrast, the communication overhead of gradient synchronization is minuscule due to the small size of model parameters and the regular communication pattern. As a result, the communication between nodes in the Aggregation step is the main concern of preset-workload-based execution in distributed full-batch training.

4) Optimization Techniques: Here we introduce the optimization techniques used to balance workload, reduce transmission, and reduce memory pressure for preset-workload-based execution in detail. We classify them into four categories: ❶ Graph Pre-partition, ❷ Transmission Optimization, ❸ Delayed Aggregation, and ❹ Activation Rematerialization. A summary for these categories is shown in Table V.

❶ **Graph Pre-partition.** Graph pre-partition refers to partitioning the whole graph into several subgraphs according to the number of computing nodes, mainly to balance workload and reduce transmission [46]. This operation is conducted in the preprocessing phase. The two key principles in designing the partitioning algorithm are listed as follows.

First, in order to pursue workload balance, the subgraphs need to be similar in size. In preset-workload-based execution, each worker performs the computation of vertices within its own subgraph. Therefore, the size of the subgraph determines the workload of the worker. The main reference parameters are the number of vertices and edges in the subgraph. Since preset-workload-based execution has a gradient synchronization barrier, workload balance is very important for computing nodes to complete a round of computation at a similar time. Otherwise, some computing nodes will be idle, causing performance loss.

Second, minimizing the number of edge-cuts in the graph pre-partition can reduce communication overhead. It is inevitable to cut edges in the graph pre-partition, meaning that the source and destination vertices of an edge might be stored in different computing nodes. When the information of neighboring vertex is required during the computation of the Aggregation step, communication between workers is introduced. Therefore, reducing the number of edges cut can reduce communication overhead in the Aggregation step.

DGCL [46] uses METIS library [91] to partition the graph for both the above two targets. Dorylus [49] also uses an edge-cut algorithm [92] for workload balance. DistGNN [47] aims at the two targets too. However, it uses a vertex-cut based graph partition technique instead. This means distributing the edges among the partitions. Thus, each edge exists in only one partition, while a vertex can reside in multiple partitions. Any updates to such vertex must be synchronized to its replicas in other partitions. FlexGraph [37] partitions the graph in the manner of edge-cut too. Besides, it learns a cost function to estimate the training cost for the given GNN model. Using the estimated training cost, FlexGraph migrates the workload from overloaded partitions to underloaded ones to pursue workload balance.

❷ **Transmission Optimization.** Transmission optimization refers to adjusting the transmission strategy between computing nodes to reduce communication overhead. Due to the irregular nature of communication in preset-workload-based execution, the demand for transmission optimization is even stronger than above.

DGCL [46] provides a general and efficient communication library for distributed GNN training. It tailors the shortest path spanning tree algorithm to transmission planning, which jointly considers fully utilizing fast links, avoiding contention, and balancing workloads on different links.

Compared to traditional transmission planning, FlexGraph [37] takes a different approach to take advantage of the aggregation nature of the Aggregate function. FlexGraph [37] partially aggregates the features of neighboring vertices that co-locate at the same partition when possible, aiming to reduce the amount of data transmission and overlap partial aggregations and communications. When each computing node receives a neighbor information request from other nodes, it will first partially aggregate the neighbor information locally, and

Optimization Technique	Function	Operation Object	Focus	Related Work
❶ Graph Pre-partition	Workload Balance and Transmission Reduction	Entire Graph	Graph Partition Strategy	DGCL [46], Dorylus [49], FlexGraph [37], DistGNN [47]
❷ Transmission Optimization	Transmission Reduction	Transmission Data	Transmission Strategy	DGCL [46], FlexGraph [37]
❸ Delayed Aggregation	Transmission Reduction	Aggregation Operation	Aggregation Strategy	Dorylus [49], DistGNN [47]
❹ Activation Rematerialization	Memory Pressure Reduction	Intermediate Results	Intermediate Results Retrieve Strategy	SAR [50]

then send the partial result to the requesting computing node, instead of directly sending the initial neighbor information. The requesting computing node only needs to aggregate the received partial result with its local nodes to continue the computation. As a result, due to the reduction in data transmission and the overlap of computation and communication, the communication overhead is significantly reduced.

❸ **Delayed Aggregation.** By allowing the computing nodes to utilize old transmitted data in the Aggregation step, delayed aggregation can reduce the overhead of data transmission. Normally, there is no scope for intra-epoch overlap due to the dependence between consecutive phases in an epoch. Delayed aggregation solves the problem to overlap the computation and communication, allowing the model to use the previously transmitted data. However, we need to point out that in order to ensure convergence and final accuracy, the delayed aggregation is mainly based on bounded asynchronous training [49].

Dorylus [49] uses bounded asynchronous training on two synchronization points: the update of weight parameters in the backward propagation, and the neighbor aggregation in the Aggregation step. Bounded asynchronous training of GNNs is based on bounded staleness [93]–[96], an effective technique for mitigating the convergence problem by employing lightweight synchronization. The key policy is to restrict the number of iterations between the fastest worker and the slowest worker to not exceed more than a user-specified staleness threshold s, where s is a natural number. As long as the policy is not violated, there is no waiting time among workers.

DistGNN [47] proposes the delayed remote partial aggregates (DRPA) algorithm to overlap the communication with local computation, which is inter-epoch computation-communication overlap. In the algorithm, the set of vertices that may be queried by other computing nodes is partitioned into r subsets. For each epoch computation, only the data of one subset is transmitted. The transmitted data is not required to be received at this epoch, but after r epochs. This means that the computing nodes do not use the latest global data of vertices, but locally existing data of them. This algorithm allows communication to overlap with more computational processes, thereby reducing communication overhead.

❹ **Activation Rematerialization.** Activation rematerialization uses data retransmission and recomputation during the computation process to reduce the memory pressure of computing nodes caused by intermediate results or data.

In preset-workload-based execution, the graph data is stored in each worker in a distributed manner: if there are n workers, then each worker only needs to store $1/n$ of raw data initially. However, during the computation, each worker needs to store the information received from other workers. In forward propagation, each worker needs to query other workers to obtain the neighbor information of its local vertices, and the information needs to be stored for later use in backward propagation. As a result, the actual data stored by each worker during the computation is much larger than its initial size. In addition, its size is difficult to estimate and may lead to memory overflow problems.

Activation rematerialization, a widely-applied and mature technology in DNN, solves the problem by storing all activations during forward propagation. Activation is the output of each layer of neural networks, which means the representation of each vertex of each layer in GNNs. Its idea is to recompute or load the activations directly from disks during the backward propagation to reduce the pressure of memory [97], [98].

SAR [50] draws on the idea of activation rematerialization and proposes sequential aggregation and rematerialization for distributed GNN training. The specific execution flow is as follows. In forward propagation, each computing node only receives activation from one other computing node at a time. After the aggregation operation is completed, the activation is removed immediately. Then the computing node receives the activation from the next computing node and continues the aggregation operation. This makes the activation of each vertex only exist in the computing node where it is located, and there will be no replicas. In backward propagation, the computation is also performed sequentially as above. Each computing node transmits activation sequentially to complete the computation. Through this method, memory will not overflow as long as the memory capacity of the computing node is larger than the size of two subgraphs. This allows SAR to scale to arbitrarily large graphs by simply adding more workers.

V. DISTRIBUTED MINI-BATCH TRAINING

This section describes GNN distributed mini-batch training in detail. Our taxonomy classifies it into two categories

(a) Categories of GNN Distributed Mini-batch Training

(b) Illustration of Individual-sample-based Execution

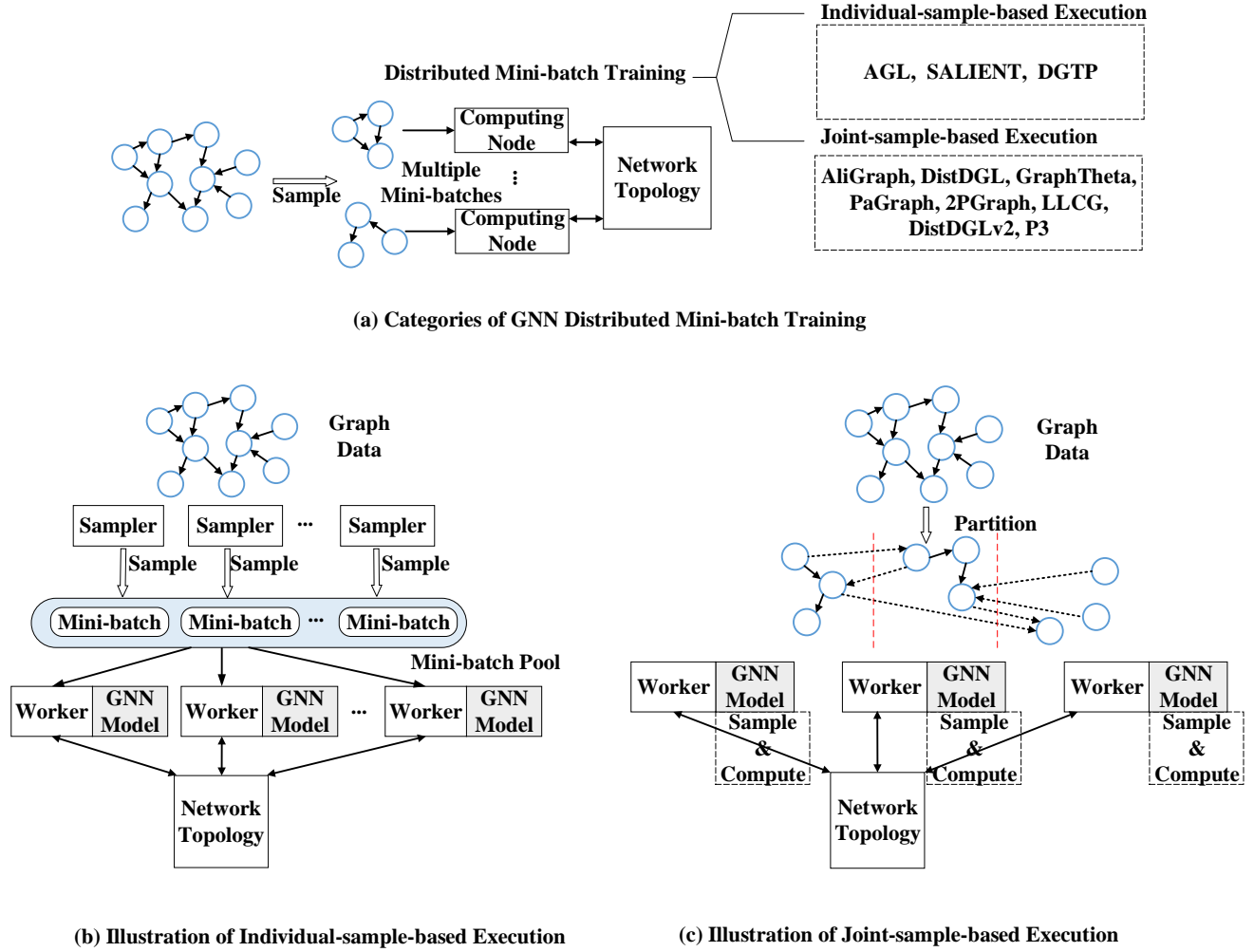**(c) Illustration of Joint-sample-based Execution**

Fig. 8. GNN distributed mini-batch training: (a) Categories of GNN distributed mini-batch training; (b) Illustration of individual-sample-based execution; (c) Illustration of joint-sample-based execution.

according to whether the sampling and model computation are decoupled, namely individual-sample-based execution and joint-sample-based execution, as illustrated in Fig. 8 (a).

A. Individual-sample-based Execution

The individual-sample-based execution of distributed mini-batch training is illustrated in Fig. 8 (b). Its workflow, computational pattern, communication pattern, and optimization techniques are introduced in detail as follows.

1) Workflow: The individual-sample-based execution involves multiple samplers and workers so that it decouples sampling from the model computation. The sampler first samples the graph data to generate a mini-batch, and then sends the generated mini-batch to the workers. The worker performs the computation of the mini-batch and conducts gradient synchronization with other workers to update the model parameters. By providing enough computing resources for the samplers to prepare mini-batches for the workers, the computation can be performed without stalls.

A more detailed description of the workflow follows. First, the samplers generate mini-batches by querying the graph structure. Since each worker requires one mini-batch per round of training, the samplers need to generate enough mini-batches in time. These mini-batches are transferred to the workers for subsequent computations. The workers perform forward propagation and backward propagation on their own received mini-batch and generate gradients. After that, gradient synchronization is conducted between workers to update model parameters.

2) Computational Pattern: The computational pattern in the individual-sample-based execution is dominated by the computation of the sampler and the workers, which are responsible for the sampling phase and other computational phases respectively. The sampling phase relies on the graph structure, which leads to irregularities in the computation. These irregularities are reflected in the uncertainty of the data, including its amount and storage address, making it difficult to estimate the computational efficiency of the samplers, as well as the efficiency of mini-batch generation. After that, the sampler sends the mini-batches to the workers, which perform model computation on the batches. In contrast to sampling, the computational efficiency of workers is easy to estimate, since

TABLE VI

Optimization Technique	Function	Operation Object	Focus	Related Work
❶ Parallel Mini-batch Generation	Mini-batch Generation	Generation Operation	Generation Parallel Strategy	SALIENT [56], AGL [36]
❷ Dynamic Mini-batch Allocation	Workload Balance	Mini-batch	Mini-batch Allocation Strategy	SALIENT [56], AGL [36]
❸ Mini-batch Transmission Pipelining	Transmission Reduction	Transmission Operation	Transmission Strategy	SALIENT [56], AGL [36]
❹ Parallel Aggregation with Edge Partitioning	Parallelism Exploitation	Aggregation Operation	Computation Strategy	AGL [36]

the amount of data and computation between mini-batches are similar. In addition, since the size of the GNN model is small, the gradient synchronization overhead is also small, which makes it generally not a bottleneck [38], [51]. As a result, the difference in the computational pattern between the samplers and workers makes the generation of mini-batch and the balance of consumption the focus of attention. The key optimization point is to accelerate the sampling process, so that it provides enough mini-batches in time to avoid stalls.

3) Communication Pattern: The main communication of individual-sample-based execution is the mini-batch transmission between samplers and workers, which is characterized by regular but frequent. Since the mini-batch consists of a fixed number of target vertices and their limited neighbors, the amount of data is consistent and small. In addition, its transmission target is determined. This makes the transmission regular. Due to the small amount of data in the mini-batch, the amount of computation is also small and it is easy to estimate the time cost required for computation. Due to the continuous consumption of mini-batches by workers, frequent mini-batch transfers are required for maintaining a timely supply of mini-batches.

4) Optimization Techniques: Here we introduce the optimization techniques used to generate mini-batch, balance workload, reduce transmission, and exploit parallelism for individual-sample-based execution in detail. We classify them into four categories: ❶ Parallel Mini-batch Generation, ❷ Dynamic Mini-batch Allocation, ❸ Mini-batch Transmission Pipelining, and ❹ Parallel Aggregation with Edge Partitioning. A summary for these categories is shown in Table VI.

❶ **Parallel Mini-batch Generation.** Parallel mini-batch Generation refers to parallelizing the generation of mini-batches to reduce the waiting time of the workers for mini-batches. It aims to speed up the sampling to provide the workers with sufficient mini-batches in time. As mentioned before, the computational pattern of the sampling is irregular, which is hard to be efficiently performed by GPUs. Therefore, CPUs are generally used to perform sampling [51], [52]. A widely-used solution is to parallelize the generation of mini-batches by using CPUs' multi-thread design.

SALIENT [56] parallelizes mini-batch generation by using the multi-thread technology of CPUs. It uses C++ threads to end-to-end implement mini-batch generation rather than Python threads to avoid Python's global interpreter lock, and thus improves the performance of mini-batch generation.

AGL [36], on the other hand, generates k-hop neighborhoods of vertices through preprocessing to simplify and speed up the generation of mini-batches. Here, the k-hop neighborhoods are the vertices that can be visited within the maximum k edges by the given vertex. AGL proposes a distributed pipeline to generate k-hop neighborhoods of each vertex based on message passing, and implements it with MapReduce infrastructure [99]. The generated k-hop neighborhoods information is stored in the distributed file system. Since it is only necessary to collect the k-hop neighborhoods information of the target vertices when a mini-batch is required, such a design greatly accelerates the sampling process. In order to limit the size of k-hop neighborhoods, AGL implements a set of sampling strategies to selectively collect neighbors.

❷ **Dynamic Mini-batch Allocation.** In dynamic mini-batch allocation, mini-batches are dynamically allocated to the workers to ensure that each computing node has mini-batch supply in time and alleviate workload imbalance. Usually, a well-designed allocator is used to collect the mini-batch generated by the samplers, and then supply the mini-batch in time according to the needs of the workers. In this way, the performance of sampling is determined by the efficiency of all samplers, instead of the slowest sampler. This can effectively avoid the situation that the computation is stalled due to the long sampling time of some slow samplers.

Meanwhile, dynamic mini-batch allocation helps to alleviate workload imbalance. Traditionally, the mini-batch static allocation is that each worker has one or several corresponding samplers, and its sampler is responsible for sampling and transferring mini-batches to it. Due to the irregular computational pattern of sampling, the computing time of a sampler is difficult to estimate, and it may lead to serious workload imbalance in static allocation, as the workers corresponding to some slow samplers have to wait for their mini-batches. In contrast, dynamic mini-batch allocation uniformly manages the workload and dispatches mini-batches to workers in time, thereby avoiding workload imbalance and achieving better performance.

SALIENT [56] implements dynamic mini-batch allocation by using a lock-free input queue. CPU responsible for sampling and GPU responsible for model computation are not

in a static correspondence. The number of each worker is sequentially stored in the queue. The mini-batch generated by each CPU will be assigned a destination worker number according to the number stored in the queue during the generation. After the generation, the mini-batch will be sent to the corresponding worker according to the destination number immediately. The lock-free input queue is a simple design that effectively avoids the problem faced by static allocation.

❸ **Mini-batch Transmission Pipelining.** Mini-batch transmission pipelining refers to pipelining the transmission of mini-batches and the model computation to reduce the overhead of data transmission.

Due to the decoupled nature of sampling and model computation in individual-sample-based execution, mini-batches need to be transferred from samplers to workers. The overhead of the transmission is high and even occupies up to 35% of the per-epoch time [56]. Fortunately, the mini-batch transmission and model computation do not require strict sequential execution. Thus, the mini-batch transmission and model computation pipelining can be implemented to reduce the mini-batch transmission overhead. This reduces the occurrence of worker stagnation due to the failure of the mini-batch to be supplied in time.

In SALIENT [56], the samplers are CPUs and the workers are GPUs, so mini-batches need to be transferred from CPUs to GPUs. To pipeline mini-batch transmission and model computation, it uses different GPU threads to deal with each of them respectively. AGL [36] uses the idea of pipelining to reduce the data transmission too.

❹ **Parallel Aggregation with Edge Partitioning.** Parallel aggregation with edge partitioning refers to partitioning the edges in mini-batches according to their destination vertices and assigning each partition to an individual thread to accomplish aggregation in parallel.

For the Aggregation operation of GNNs, the neighboring vertex's feature needs to be aggregated for each vertex. It means the computation is actually determined by the edges. AGL [36] proposes edge partitioning techniques to accomplish parallel aggregation according to this property. In each mini-batch, AGL partitions the edges into several partitions according to their destination vertices ensuring that the edges with the same destination vertex are in the same partition. Then these partitions are handled by multiple threads independently. As there are no data conflicts between these threads, the Aggregation operation can be accomplished effectively in parallel. In addition, the workload balance between these threads is guaranteed. It is because the mini-batch generated by sampling ensures that each vertex has a similar number of neighboring vertex, resulting in a similar number of edges per edge partition.

B. Joint-sample-based Execution

The joint-sample-based execution of distributed mini-batch training is illustrated in Fig. 8 (c). Its workflow, computational pattern, communication pattern, and optimization techniques are introduced in detail as follows.

1) Workflow: In the joint-sample-based execution, multiple collaborative workers are used to perform training. The graph is split into subgraphs through the partition in the preprocessing. Each worker holds one subgraph and a replica of the model parameters. The workers sample their own subgraphs to generate mini-batches and perform forward propagation as well as backward propagation on the mini-batches to obtain gradients. Then they update the model synchronously through communication with other workers.

A more detailed description of the workflow follows. First, in order to make the computation of each worker more independent, a well-design graph partition algorithm is used to split the original graph into multiple subgraphs in the preprocessing. Each subgraph is assigned to a worker, and the worker samples its local subgraph, which means the target vertex chosen for sampling is chosen only from its own subgraph. However, the worker may need to query other workers for the neighborhoods of target vertices in the sampling. Second, the worker performs model computation on the mini-batch generated by itself. Through forward and backward propagation, each worker produces gradients and conducts gradient synchronization together to update model parameters. Note that the sampling and model computation is conducted on a single worker. This feature makes the computation of each computing node more independent, which greatly reduces the demand for data transmission.

2) Computational Pattern: In the joint-sample-based execution, the computation of each computing node is relatively independent. The computation of each computing node includes two parts: sampling and model computation. In addition, gradient synchronization between computing nodes is required for updating model parameters. In the sampling phase, each computing node mainly uses its own local data, and there may be a need to access remote data for neighbor information of vertices. The mini-batch generated by sampling is directly consumed by itself for model computation. For such a computing pattern, the main concern is how to make the computation of computing nodes more independent, so as to obtain better performance.

3) Communication Pattern: There are three communication requirements in joint-sample-based execution, each with a different communication pattern. The first exists in the sampling phase, and its communication pattern is irregular. In the joint-sample-based execution, each computing node selects the target vertices of the mini-batch from its own subgraph. Then the computing node needs to sample the neighbors of these target vertices. Since the graph is partitioned, it may be necessary to query other computing nodes to obtain the neighbor information of these target vertices [38]. As the communication is determined by the graph structure and the graph itself has an irregular connection pattern, its communication pattern is irregular.

The second communication requirement is in the gradient synchronization phase. After sampling and model computation, the computing nodes need to synchronize to update the model parameters. Unlike DNNs, the size of model parameters of GNNs is much smaller, since GNNs have few layers and share weights across all vertices. As a result, it is expected

Optimization Technique	Function	Operation Object	Focus	Related Work
❶ Locality-aware Partition	Transmission Reduction	Entire Graph	Graph Partition Strategy	PaGraph [51], 2PGraph [52], DistDGL [38], AliGraph [35]
❷ Partition with Overlap	Transmission Reduction	Neighboring Vertices	Duplicate Strategy	PaGraph [51], 2PGraph [52]
❸ Independent Execution with Refinement	Waiting Time Reduction	Computing Fashion	Computing Strategy	LLCG [53]
❹ Frequently-used Data Caching	Transmission Reduction	Frequently-used Data	Caching Strategy	AliGraph [35], PaGraph [51], 2PGraph [52]

that the gradient synchronization phase occupies a small part of the execution time [38], [51]. However, synchronization with the nature of a fence requires all computing nodes to complete the computation before it can start, so it is highly sensitive to workload imbalance. The low performance of a single computing node will cause other computing nodes to wait, which will result in a large time overhead for this part of the communication [83].

The third communication requirement only arises when the sampling and model computation of a single worker are performed on different computing components, such as CPUs and GPUs on a single server [51], [52]. Its communication pattern is regular but with high redundancy. The existence of the same vertex in different mini-batches is called inter-mini-batch redundancy, which leads to wasted transmission. This redundancy is exacerbated by the fact that each computing node in the joint-sample-based execution mainly samples on its own subgraph: compared to the sampling on the entire graph, the sampling on the same subgraph exhibits a higher probability of sampling the same vertices for different mini-batches [51]. As a result, reducing transmission overhead by eliminating this redundancy is an effective optimization point.

4) Optimization Techniques: Here we introduce the optimization techniques used to reduce transmission overhead and waiting overhead for joint-sample-based execution in detail. We categorize them into four categories, including ❶ Locality-aware Partition, ❷ Partition with Overlap, ❸ Independent Execution with Refinement, and ❹ Frequently-used Data Caching. A summary for these categories is shown in Table VII.

❶ **Locality-aware Partition.** Locality-aware partition refers to partitioning the graph into subgraphs with good locality, that is, vertices and their neighbors have a high probability to be in the same subgraph, so most of the data required for sampling is local to the computing node. This partition is conducted in the preprocessing phase.

Locality-aware partition aims to make the computation of the nodes more independent by reducing the communication between them. According to the workflow of mini-batch training, the model computation is restricted in using the mini-batch data and does not involve access to the entire raw graph data. That means the access to the raw graph data is almost completely in the sampling phase. By focusing on the locality

of the graph, each vertex and its neighbors are clustered into a subgraph as much as possible. This makes the workers mainly access their own subgraph during the sampling phase, reducing the remote queries of neighboring vertices, and thus improving the independence of each worker's computation.

The graph partition here is very different from the one of distributed full-batch training mentioned earlier in the paper, which is mainly reflected in its purpose. The graph partition in the preset-workload-based execution of distributed full-batch training focuses more on workload balance due to its significant impact on performance. Because a large amount of information of the subgraph is related to the execution time, it is necessary to take various factors into account to guide the graph partition, such as the numbers of vertices and edges. In contrast, workload balance is not the critical issue for graph partition in distributed mini-batch training. Because the computation amount of the subgraph is mainly determined by the number of target vertices contained in it, it is only necessary to ensure that the number of target vertices in each subgraph tends to be consistent, which is simple but effective. Such characteristics cause the two graph partitions to be quite different, which in turn leads to different techniques used.

To implement locality-aware partition, PaGraph [51] proposes a formula to guide the graph partition. Its partition algorithm scans the whole target vertex set, and iteratively assigns the scanned vertex to one of K partitions according to the score computed from the formula:

$$score_{v_t}^{(i)} = |TV_i \cap IN(V_t)| \cdot \frac{(TV_{avg} - |TV_i|)}{|PV_i|} \quad (6)$$

where TV_i denotes the train vertex set already assigned to the i-th partition, $IN(V_t)$ denotes the L-hop (L is the layer number of GNN model) incoming neighbor set of the target vertex v_t, and PV_i denotes the total number of vertices in the i-th partition. TV_{avg} denotes the expected number of target vertices in the final i-th partition, which is typically set as the total number of vertices $|TV_i|$ divided by the partition number K for workload balance. This formula fully considers the distribution of the vertices' neighbors when splitting them into each subgraph, thus making great use of the locality of the graph.

Other researches mainly use existing graph partition algorithms to harvest the locality of the graph. 2PGraph [52]
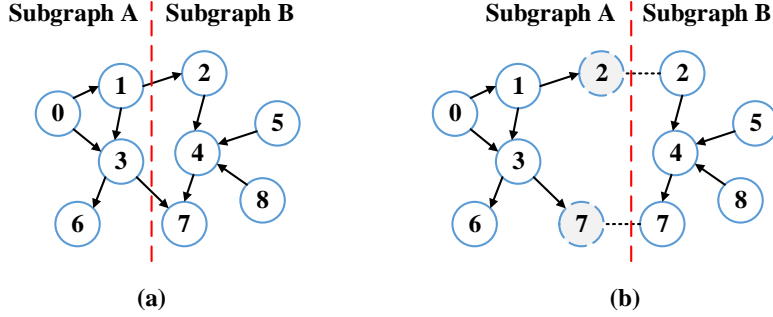
Fig. 9. Illustration of graph partition with overlap: (a) Graph partition with non-overlap where no duplicate method is used; (b) Graph partition with overlap. The duplicate range is 1-hop neighbors. The vertices in white are original target vertices, while the vertices in grey are duplicated, referred to as mirror vertices.

uses cluster-based graph partition algorithm [100], [101] in its graph partition. DistDGL [38] takes advantage of the multi-constraint mechanism in METIS [100] for graph partition. AliGraph [35] implements four graph partition algorithms in its system and suggests that the four algorithms are suitable for different circumstances. They are METIS [91] for processing sparse graphs, vertex and edge cut partitions [102] for processing dense graphs, 2-D partition [103] for systems with a fixed number of workers, and streaming-style partition strategy [104] for graphs with frequently edge updates.

❷ **Partition with Overlap.** Partition with overlap refers to duplicating the neighboring vertices of the target vertex when partitioning the graph, thereby reducing or even completely eliminating the data transmission between computing nodes in the sampling phase.

For comparison, we first review the general graph partitioning method used in traditional graph analytics, i.e., partition with non-overlap which means no duplicate method is used. An example of partition with non-overlap is demonstrated in Fig. 9 (a). The original graph is split into two subgraphs, denoted as A and B. The vertices held by each subgraph are disjoint. This has the benefit of linearly reducing the requirement of memory capacity when training large graphs. In the case of n workers, each worker only needs to hold $1/n$ of the original graph data. However, partition with non-overlap causes some vertices' neighbors to be stored in remote workers, resulting in the need for remote queries during the sampling phase. Intensive transmissions would lead to inefficiencies in the parallel execution of distributed training.

To reduce the amount of transmission in the sampling phase, partition with overlap is proposed. An example of partition with overlap is illustrated in Fig. 9 (b), whose duplicate range is 1-hop neighbors. The original graph is split into two subgraphs with duplicate vertices. Vertex 2 and 7 are duplicated as they are the 1-hop neighboring vertex of target vertices in subgraph A. This method takes advantage of the characteristics of GNN mini-batch training. When the mini-batch is sampled, only the L-hop neighboring vertices of the target vertices are needed, where L is the number of GNN layers. Therefore, by duplicating the features of its neighboring vertices, the transmission overhead in the sampling phase

can be reduced. These duplicate vertices are regarded as mirror vertices, which do not participate in training as target vertices [102]. If the L-hop neighbors are fully duplicated, the transmission requirements during the sampling phase can be completely eliminated. Note that duplicating more vertices causes an increase in the memory capacity requirements of the computing nodes, so there is a trade-off in how many neighboring vertices are duplicated.

Both PaGraph [51] and 2PGraph [52] use this method to make the computation of each worker more independent. DGCL [46] defines a replication factor, which is computed from the total number of vertices (including the original and mirror vertices) kept by all workers divided by the number of vertices in the original graph. In its experiments, for the dense Reddit dataset [13] with 16 GPU workers and 2-hop duplicate range, the replication factor reach 15. In contrast, for the sparser Web-Google dataset [105] with the same setting, the replication factor is much smaller which is 2.5. It suggests that sparse graphs are more suitable for the duplicate method as they have less memory capacity overhead.

❸ **Independent Execution with Refinement.** Independent execution with refinement refers to letting each worker compute and update the model parameters independently, and then periodically averages the model parameters using additional refine operation to release the potential of parallel computation while ensuring high accuracy.

Although the workers of joint-sample-based execution mainly perform computations on their own local subgraph, there are still interactions between them to advance the training process, including the sampling phase and the gradient synchronization phase. The limitations of these two phases cause each worker's execution to be affected by the state of other workers, resulting in idle wait time.

In order to achieve more independent computation of each computing node, LLCG [53] proposes a method of independent execution with refinement. In its implementation, each worker first performs sampling and corresponding model computation independently on their own subgraphs. That is to say, the data of remote workers is not accessed at all, and each worker only updates its own model parameters locally. A parameter server then periodically collects the model param-

eters from each worker, and performs the average operation of the parameters. Finally, the parameter server refines the average result, that is, it samples the mini-batch on the whole graph and conducts model computation to update the model parameters of every worker. This is due to the fact that each worker lacks the structural information of the whole graph, which may lead to low accuracy. Through this method, the computation of each worker is more independent while the refinement on model parameters ensures the high accuracy of GNN model.

❹ **Frequently-used Data Caching.** Frequently-used data caching refers to caching the information of frequently used vertices from other computing nodes, including neighbors or features, to reduce the overhead of data transmission.

To reduce the transmission overhead of obtaining the neighboring vertices in the sampling phase, AliGraph [35] proposes to cache the outgoing neighbors of important vertices wherever important vertices are located. Important vertices are identified by the following formula.

$$Imp^k(v) = \frac{Nin^k(v)}{Nout^k(v)} > U^k \qquad (7)$$

where $Imp^k(v)$ denotes the importance score of the vertex v, $Nin^k(v)$ and $Nout^k(v)$ represents the number of k-hop incoming and outgoing neighbors of the vertex v, and U^k is a user-specified threshold. When the $Imp^k(v)$ exceeds U^k, the vertex v is considered to be an important vertex. By caching the outgoing neighbors of important vertex v, the transmission overhead derived from the queries of other vertices via v can be reduced.

Similarly, the frequently-used data caching technique can also reduce the transmission of mini-batches between CPUs and GPUs. It is the third communication requirement of joint-sample-based execution which occurs when CPUs and GPUs are responsible for sampling and model computation respectively. Due to the benefit of mini-batch training, each GPU only needs to load one mini-batch of data per round of computation, and it is possible to load cached data into the free memory during model computation. Thus, by caching node features and combining them into mini-batches on GPUs, the amount of mini-batch transmission can be reduced.

PaGraph [51] uses a very simple caching strategy. It sorts vertices according to their degree from largest to smallest and then caches them in order, where the degree is the number of incoming edges of a vertex. The higher the degree, the higher the cache priority. This is a simple but effective method: when a vertex has a larger degree, this vertex is more likely to be a neighbor of other target vertices. As for the proportion of cached vertices, it is determined by measuring the free memory capacity of GPUs during the pre-run. 2PGraph [52] proposes GNN Layer-aware Caching. It adopts a sampling method that uses a fixed order of target vertices, which allows it to schedule the vertices to train next. By pre-caching all of the target vertices' L-hop neighbors into GPU memory before the computation, the time overhead of data transmission can be almost completely eliminated. However, fixing the order of target vertices may compromise the model accuracy, so it proposes to periodically permute the order to reduce this effect.

VI. SOFTWARE FRAMEWORKS FOR DISTRIBUTED GNN TRAINING

In this section, we introduce the software frameworks of distributed GNN training. Table VIII provides a summary of these frameworks and their supported attributes.

Basic Software Frameworks. PyG [84] and DGL [85] are the two most popular software frameworks in the GNN community. PyG [84] is a geometric deep learning extension library for PyTorch to enable deep learning on irregular structure data such as graphs. It supports both CPU and GPU computing, providing convenience for using GPU to accelerate the computing process. Through the message-passing application programming interface (API), it is easy to express various GNN models, as neighbor aggregation is a kind of message propagation. DGL [85] is a framework specialized for deep learning models on graphs. It abstracts the computation of GNNs into a few user-configurable message passing primitives, thus helping users express GNNs more conveniently. It achieves good performance by exploring a wide range of parallelization strategies. It also supports both CPU and GPU computing.

Due to their open source and easy-to-use nature, more and more software frameworks are building on top of them and propose optimizations for distributed GNN training, such as DistGNN [47], SAR [50], DistDGL [38], PaGraph [51], LLCG [53], DistDGLv2 [54], SALIENT [56], P3 [106], and so on. Next, we introduce the software frameworks dedicated to distributed full-batch training and distributed mini-batch training.

Software Frameworks Dedicated to Distributed Full-batch Training. The software frameworks dedicated to distributed full-batch training includes NeuGraph [44], Roc [45], FlexGraph [37], MG-GCN [48], and Dorylus [49].

NeuGraph [44] is a distributed GNN training software framework proposed in 2019, using multi-GPU hardware platform. It is categorized as the dispatch-workload-based execution of distributed full-batch training. It proposes SAGA-NN, an abstract model for the programming of GNN operations which splits each layer of model computation into four stages: Scatter, ApplyEdge, Gather, and ApplyVertex. SAGA-NN is named after these stages. The Scatter stage means the vertices scatter their features to their output edges, while the Gather stage means the vertices gather the value from their input edges. There are two user-defined functions used in the ApplyEdge stage and ApplyVertex stage, for users to declare neural network computations on edges and vertices respectively. Through the abstraction of SAGA-NN, users can easily express various GNN models and execute them in a parallelized way. NeuGraph optimizes the training process based on this abstract model, using techniques including vertex-centric workload partition, transmission planning, and feature-dimension workload partition.

Roc [45] is a distributed multi-GPU software framework for fast GNN training and inference, proposed in 2020. It is categorized as the dispatch-workload-based execution of distributed full-batch training. Its optimization of distributed training mainly focuses on balanced workload generation

TABLE VIII
Software frameworks for distributed GNN training.

Name	Multi-CPU	Multi-GPU	Distributed Mini-batch Training	Distributed Full-batch Training	Homogeneous Graph	Heterogeneous Graph	Static Graph	Dynamic Graph
PyG [84]	✓	✓	✓	✓	✓	✓	✓	✓
DGL [85]	✓	✓	✓	✓	✓	✓	✓	✓
NeuGraph [44]		✓		✓	✓		✓	
Roc [45]		✓		✓	✓		✓	
FlexGraph [37]	✓			✓	✓	✓	✓	
MG-GCN [48]		✓		✓	✓		✓	
Dorylus [49]	✓			✓	✓		✓	
AliGraph [35]	✓		✓		✓	✓	✓	✓
AGL [36]	✓		✓		✓		✓	
GraphTheta [55]	✓		✓	✓	✓		✓	

and transmission planning. In terms of balanced workload generation, an online linear regression cost model is proposed to achieve efficient graph partition. The cost model is being tuned by collecting runtime data. According to this cost model, the training resources and time required for the subgraph can be estimated to guide the workload partition for workload balance. In terms of transmission planning, a recursive dynamic programming algorithm is introduced to find the global optimal solution to decide which part of the data should be cached in GPU memory for reuse.

FlexGraph [37] is a distributed multi-CPU training software framework proposed in 2021. It is categorized as the preset-workload-based execution of distributed full-batch training. In order to express more kinds of GNNs including GNNs for heterogeneous graphs, it proposes a novel programming abstraction, namely NAU. NAU splits the computation of one GNN layer into three stages including NeighborSelection, Aggregation, and Update stages, each with a user-defined function. Based on NAU, FlexGraph optimizes the training process using techniques including graph pre-partition and partial aggregation. In terms of graph pre-partition, it introduces a cost model to estimate the runtime overhead of the workload, so as to guide the workload partition for workload balance. In terms of partial aggregation, FlexGraph partially aggregates the features of vertices collocated at the same partition when possible. In addition, it overlaps partial aggregations and communication to reduce the transmission overhead.

MG-GCN [48] is a distributed multi-GPU training software framework proposed in 2021. It is categorized as the preset-workload-based execution of distributed full-batch training. It focuses on the efficient parallelization of the sparse matrix-matrix multiplication (SpMM) kernel on multi-GPU hardware platform. It uses matrix partitioning method to distribute raw data to multiple GPUs, and each GPU is responsible for completing the workload of its own local matrix. It involves the efficient reuse of memory buffers to reduce the memory footprint of training GNN models, and overlaps communication and computation to reduce communication overhead. Specifically, the memory buffer in the computing node is used to cache the data reused by the forward propagation and backward propagation, thereby reducing data transmission. As for the communication and computation overlap, it uses two GPU streams for computation and communication, respectively.

Dorylus [49] is a distributed multi-CPU training software framework proposed in 2021. It is categorized as the preset-workload-based execution of distributed full-batch training. Its main focus is on how to train GNNs at a low cost, so it adopts serverless computing. Serverless computing refers to "cloud function" threads, such as AWS Lambda and Google Cloud Functions, that can be used massively in parallel at an extremely low price. The hardware platform of Dorylus consists of CPUs and serverless threads. CPUs mainly perform the Aggregation operation, while the serverless threads are used for the Combination operation due to more regular computation and simpler workload partition in the Combination operations. It adopts a fine-grained workload partition to adapt to the situation that the available hardware resources of serverless threads are quite limited. In addition, asynchronous training is used to make full use of computing resources and reduce stagnation.

Software Frameworks Dedicated to Distributed Mini-batch Training. The software frameworks dedicated to distributed mini-batch training includes AliGraph [35] and AGL [36].

AliGraph [35] is a distributed multi-CPU training framework proposed in 2019. It is categorized as the joint-sample-based execution of distributed mini-batch training. It supports not only GNNs for homogeneous graphs and static graphs, but also GNNs for heterogeneous graphs and dynamic graphs. In terms of storage, it adopts a graph partitioning method to store graph data in a distributed manner. The structure and features of the subgraph in each computing node are stored separately. In addition, two caches are added for the features of vertices and edges. Furthermore, it proposes a caching strategy

to reduce the communication overhead between computing nodes, that is, each computing node caches the outgoing neighbors of frequently-used vertices.

AGL [36] is a distributed multi-CPU training software framework proposed in 2020. It is categorized as the individual-sample-based execution of distributed mini-batch training. To speed up the sampling process, it introduces a distributed pipeline to generate k-hop neighborhood in the spirit of message passing, which is implemented with MapReduce infrastructure [99]. In this way, in the sampling phase, mini-batch data can be rapidly generated by collecting the k-hop neighbors of the target vertices. In the training phase, the computing nodes are partitioned into workers and parameter servers. The workers perform the model computation on the mini-batches, while the parameter servers maintain the current version of the model parameters. It also uses the commonly used optimization techniques for better efficiency, such as the transmission pipeline.

Software Frameworks Dedicated to Both Distributed Full-batch Training and Mini-batch Training. There exists a software framework dedicated to both distributed full-batch training and distributed mini-batch training.

GraphTheta [55] is a distributed multi-CPU training software framework proposed in 2021. It supports three training methods: mini-batch, full-batch, and cluster-batch training. The cluster-batch training methods is proposed by Chiang [80] in 2019. It first partitions a large graph into a set of smaller clusters. Then, it generates a batch of data either based on one cluster or a combination of multiple clusters. Apparently, cluster-batch restricts the neighbors of a target vertex into only one cluster, which is equivalent to conducting a globalized convolution on a cluster of vertices. There is a parameter server in GraphTheta, which is responsible for managing multi-version model parameters. The worker obtains the model parameters from the parameter server and transfers the generated gradient back to the parameter server for the update of model parameters. Multi-version parameter management makes it possible to train GNNs asynchronously as well as synchronously.

VII. Hardware Platforms for Distributed GNN Training

In this section, we introduce the hardware platforms for distributed GNN training. At present, it is mainly categorized into two types: multi-CPU hardware platform and multi-GPU hardware platform. A summary of the hardware platform for distributed GNN training is shown in Table IX. Next, we introduce them separately.

A. Multi-CPU Hardware Platform

The structure diagram of multi-CPU hardware platform is illustrated in Fig. 10 (a). It is mainly composed of multiple computing nodes. Each computing node has one or multiple CPUs. CPUs in each computing node communicate with each other through the network. The graph data is generally stored in each computing node in a distributed storage manner. Each computing node performs computations mainly on its own local data. When remote data is required, it queries other computing nodes through the network to obtain the required data.

The main advantage of multi-CPU hardware platform is the good scalability due to the large memory [38], [47]. By increasing the number of computing nodes, large graphs can be maintained completely in memory. This makes it possible to avoid data access to high-latency storage components, such as hard disks, thereby ensuring the timely supply of data. This characteristic makes it ideal for preset-workload-based execution of distributed full-batch training [37], [47] and joint-sample-based execution of distributed mini-batch training [35], [38], [55]. The graph is partitioned and distributed to different computing nodes for storage. Each computing node is responsible for the workload of its own subgraph. This results in computing nodes using mostly local data for computation, although additional communication is also involved. By further optimizing the communication overhead, multi-CPU platforms can achieve good scalability.

The main disadvantage of multi-CPU hardware platform is the limited computing resource in each computing node. CPU is a latency-oriented architecture with limited computing resources [107]. This makes it less friendly to neural network operations that incur a lot of computation [108], [109]. This problem is even more pronounced when using distributed mini-batch training. In this case, CPUs perform both the sampling and model computation, leading to a shortage of computing resources.

B. Multi-GPU Hardware Platform

The structure diagram of multi-GPU hardware platform is illustrated in Fig. 10 (b). Graph data is generally stored in a distributed manner on multiple computing nodes. Each computing node consists of CPUs and GPUs. Computing nodes are connected through high-speed network interfaces. For example, GPUs communicate with each other over PCIe under the same computing node and over InfiniBand across different computing nodes. GPUs can also be interconnected through NVLink and NVSwitch for higher-speed inter-GPU communication.

The main advantage of multi-GPU hardware platform is the rich computing resources. Compared with CPUs, GPUs have more computing resources that can be used for large-scale parallel computing. This is because GPU is a throughput-oriented architecture which focuses on improving the throughput by using massive computing resources [107]–[109]. In view of the development of DNN, both sparse matrix multiplication and dense matrix multiplication in GPU have been fully developed, which facilitates the model computation of GNN. Due to its limited memory capacity, it is not suitable for distributed full-batch training. Otherwise, it will result in a large amount of data transmission. Currently, distributed mini-batch training is usually performed on multi-GPU hardware platform. The most popular way to perform distributed mini-batch training on multi-GPU hardware platform is that CPUs are responsible for the sampling phase and GPUs are responsible for the model computation. This is because the computational pattern of the
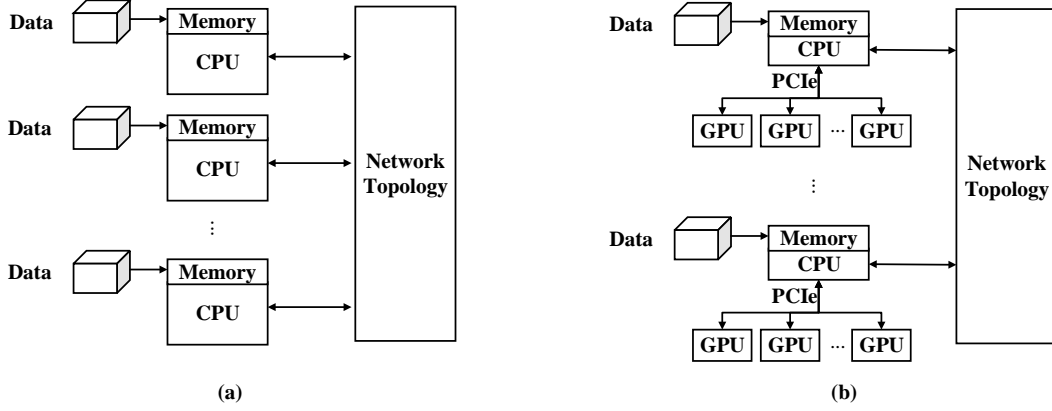
Fig. 10. Hardware platforms for distributed GNN training: (a) Multi-CPU hardware platform; (b) Multi-GPU hardware platform.

TABLE IX
SUMMARY OF HARDWARE PLATFORMS FOR DISTRIBUTED GNN TRAINING.

Hardware Platform	Main Advantages	Main Disadvantages	Suitable Training Method
Multi-CPU	Rich Memory Resource and Good Scalability	Limited Computing Resource	Distributed Full-batch Training
Multi-GPU	Rich Computing Resource	Limited Memory Resource, Resource Contention, and High Transmission Overhead	Distributed Mini-batch Training

sampling phase is irregular and CPUs are more suitable for the irregular computational pattern than GPUs [88], [90].

The main disadvantages of multi-GPU hardware platform are resource contention issues and expensive data transmission overhead. Since there are generally multiple GPUs on a computing node, these GPUs share the computing node's resources, including CPU sampling threads and network interface [44], [83]. The competition for these resources makes multiple GPUs hard to obtain sufficient input data in time. This causes the occurrence of computational stagnation, resulting in poor scalability, manifested by a large gap between the actual speedup and the ideal [83]. The expensive data transmission overhead is due to the tight memory resources of GPUs. When using a multi-GPU hardware platform for distributed full-batch training, it needs to continuously transmit input data and output data [44]. When performing distributed mini-batch training, it is also necessary to continuously transmit the mini-batch data sampled by CPUs [51]. This problem is further exacerbated when the graph is large.

C. Multi-CPU Hardware Platform V.S. Multi-GPU Hardware Platform

The summary of the comparison between multi-CPU hardware platform and multi-GPU hardware platform is shown in Table IX. Multi-CPU and multi-GPU hardware platforms have their own advantages and are suitable for different scenarios. The different characteristics of the two also lead to different optimization directions. Multi-CPU hardware platform is more suitable for distributed full-batch training. The entire graph can be cached in the memory, which leads to good scalability.

Multi-GPU hardware platform is more suitable for distributed mini-batch training given its limited memory resources. However, when the graph size is small and there are high-speed interconnections between GPUs, such as NVLink, distributed full-batch training using multi-GPU hardware platform can achieve good performance [45].

VIII. COMPARISON TO DISTRIBUTED DNN TRAINING

In this section, we highlight the characteristics of distributed computing of GNNs by comparing the distributed training of GNNs with that of DNNs. We first introduce the two categories of distributed DNN training: data parallelism and model parallelism. Among them, for GNN distributed full-batch training, its operation mode is similar to the model parallelism of DNNs. GNN distributed mini-batch training is similar to the data parallelism of DNNs. The two pairs are compared separately below. A summary of the comparison between distributed GNN training and distributed DNN training is shown in Table X. Next, we introduce them separately.

A. Brief Introduction to Distributed DNN Training

Distributed DNN training is categorized into model parallelism and data parallelism [110]. *In model parallelism, the model is split into several parts and different computing nodes compute each part.* AlexNet [111] is the first one to use model parallelism, because its model size could not fit in one GPU at that time. Therefore, the model is split into two parts, which are then computed by two GPUs together.

In data parallelism, each computing node holds a copy of the model parameters, and the input dataset is distributed

	Comparison Object	Distributed Full-batch Training	DNN Model Parallelism
Similarity	Perspective of Parallelism Exploitation	Model	
	Fashion of Parallelism Exploitation	Partitioning Workload of Model Computation	
Difference	Communication Uncertainty	High	Low
	Transmission Planing Complexity	High	Low
	Probability of Communication Congestion	High	Low
	Probability of Computation Stagnation	High	Low

	Comparison Object	Distributed Mini-batch Training	DNN Data Parallelism
Similarity	Perspective of Parallelism Exploitation	Input Data	
	Fashion of Parallelism Exploitation	Partitioning Raw Data	
Difference	Source of Input Data	Generated by Sampling Phase	Directly Loaded From Raw Data
	Main Phase of Inter-node Communication	Sampling	Gradient Synchronization
	Communication Uncertainty	High	Low
	Transmission Planing Complexity	High	Low
	Probability of Communication Congestion	High	Low
	Probability of Computation Stagnation	High	Low

to each computing node. Each computing node conducts the training process completely using its own local data and uses the generated gradients to update the model parameters with other computing nodes together. Data parallelism is classified into synchronous training and asynchronous training. However, compared with synchronous distributed training, asynchronous distributed training has lower final accuracy, and sometimes non-convergence may occur [110]. In synchronous distributed training, after each computing node completes one round of training on a small piece of data, the system starts to collect gradients, which are used to update the model parameters uniformly. In this way, each round of training in each computing node is carried out with the same model parameters, which is equivalent to the case of one computing node.

B. Distributed Full-batch Training V.S. DNN Model Parallelism

Similarities. Both of them distribute the workload of model computation to different computing nodes from the perspective of the model. Each computing node needs to transfer the intermediate results of model computation to each other and cooperate to complete a round of training. In DNN model parallelism, DNN model computation is partitioned into different parts and distributed to different computing nodes [111]. Each computing node may be responsible for executing several layers of DNN, or a set of operations within each layer. Similarly, GNN distributed full-batch training also distributes the workload of model computation to different computing nodes from the perspective of the model [40], [44], [45], [48]. Since the operation of GNN model computation is to operate on vertices, including the Aggregation and Combination operations, the workload is generally distributed to different computing nodes by partitioning the graph. Each computing node is responsible for completing the Aggregation and Combination operation of its assigned vertices and communicating with other computing nodes to jointly advance the computation [37], [46], [47].

Differences. DNN model parallelism has low data communication uncertainty, while GNN distributed full-batch training exhibits high data communication uncertainty. In DNN model parallelism, most of the transmitted data is intermediate data generated in the computation. Since the transmission of intermediate data directly affects the execution of other computing nodes, there is a strong execution dependence among computing nodes. Due to the high certainty of data transmission in DNN model parallelism, which is reflected in the fixed data volume and transmission target, the computation and data transmission can be overlapped by reasonable planning to reduce the communication overhead. However, in GNN distributed full-batch training, the data communication uncertainty is high. This is because the data transmitted between each computing node is mainly the feature of the neighboring vertex needed in the Aggregation step. Due to the irregular connection pattern of the graph itself and the fluctuation of the computational efficiency of each computing node, there are great uncertainties in data transmission [46]. Therefore, the congestion of the communication network can easily occur, which in turn leads to stagnation of computation [112].

C. Distributed Mini-batch Training V.S. DNN Data Parallelism

Similarities. Both of them use small pieces of input data per round of training to update the model parameters. The computing nodes use different small pieces of input data for

computation, and then jointly update the model parameters. In DNN data parallelism, each computing node holds a portion of the original data, typically stored in its memory. In each round of computation, a small piece of data is taken from its own local data and computed. Then, using the generated gradients, the model parameters are updated asynchronously or synchronously. In GNN distributed mini-batch training, the training of each computing node also uses independent input data, that is, the mini-batch generated by sampling [51], [52].

Differences. The input data required for each round of GNN distributed mini-batch training need to be generated by the sampling, while DNN data parallelism is loaded directly from memory. In DNN data parallelism, each computing node performs computations on its own local data [110]. It does not need to communicate with other computing nodes except for the stage of updating model parameters. In GNN distributed mini-batch training, a large number of communication requirements are required in the sampling phase, including querying graph data and transmitting mini-batch data. As graphs generally exhibit an irregular connection pattern, the communication in the sampling phase is highly irregular, resulting in high communication uncertainty and planning complexity. This leads to the fact that distributed GNN training may not be able to provide input data in time due to the inefficiency of the sampling, resulting in computational stagnation. This makes it necessary to optimize the sampling stage to reduce the communication overhead when conducting GNN distributed mini-batch training [51], [52].

IX. SUMMARY AND DISCUSSION

This section summarizes the aforementioned details of distributed GNN training, and discusses several interesting issues and opportunities in this field.

The main focus of optimizing distributed full-batch training is to reduce its communication overhead [37], [40], [44]–[47], [49]. In this training method, each computing node inevitably needs to communicate with other nodes for information about the entire graph structure, so that it can finish operations which require graph data that are not stored in the local split, such as aggregating information of neighboring nodes which are distributed in other computing components. As a result, reducing communication overhead, mainly by reasonably splitting the graph and planning the data transmission, can greatly improve computational efficiency.

By contrast, the main focuses of optimizing distributed mini-batch training are to speed up the sampling phase and to reduce the mini-batch transmission overhead [35], [36], [38], [51], [52], [56]. Currently, it is popular to perform distributed mini-batch training in which CPUs are responsible for sampling and GPUs are responsible for GNN model computation. The sampling capability of CPUs is insufficient, producing a lot of idleness and limiting the utilization of GPUs due to insufficient data supply. By accelerating sampling and optimizing data transmission, the utilization of computing components can be improved, thereby improving efficiency.

The two methods show their advantages in different aspects respectively. In current designs of distributed training of GNNs, the adoption of the mini-batch method is gradually increasing due to its advantages of a simpler implementation and less memory pressure. Distributed full-batch training, on the other hand, requires programmers to put more effort into programming and design, so it is more difficult to optimize. However, some research shows that the final precision of distributed full-batch training can be higher [45].

Next, we discuss several interesting issues and opportunities in this field.

A. Quantitative Analysis of Performance Bottleneck

As mentioned before, the performance of distributed GNN training is primarily hindered by the workload imbalance across different computing nodes, irregular transmission between computing nodes, etc. Although these performance bottlenecks are well-known in the GNN community, the quantitative analysis of these bottlenecks, which is important to guide the optimizations for distributed GNN training, is still obscure.

Recently, a lot of characterization efforts for GNN training or inference have been conducted on a single computing node or a single GPU [113]–[118]. For example, Yan et al. [113] quantitatively disclose the computational pattern of the inference on a single GPU. Zhang et al. [114] quantitatively characterize the training of a large portion of GNN variants concerning general-purpose and application-specific hardware architectures. However, there are few characterization efforts on distributed GNN training [116].

Diagnosing performance issues without a quantitative understanding of the bottlenecks can cause misdiagnosis: A potentially expensive effort that involves optimizing something that is not the real problem. Although Lin et al. [116] quantitatively characterize the end-to-end execution of distributed GNN training, it is not enough for the long-term development of distributed GNN training. We believe that the quantitative analysis of performance bottleneck will be one of the main directions in the field of distributed GNN training.

B. Performance Benchmark

Despite the large quantity of performance benchmarks presented to fairly evaluate and compare the performance of software frameworks and hardware platforms for DNNs, there are few performance benchmarks for distributed GNN training. Most of the recent benchmarks for GNNs mainly focus on evaluating the prediction accuracy of trained GNN models. For example, Open Graph Benchmark (OGB) [39] provides a set of large-scale real-world benchmark datasets to facilitate the graph machine learning research. However, current benchmarks focus mainly on the accuracy, instead of performance. As a result, standardized performance benchmark suites with typical, representative, and widely-acknowledged workloads are needed for the industry and academia to comprehensively compare the performance of distributed GNN training systems.

In the field of DNNs, MLperf [119], an industry standard benchmark suite for machine learning performance, has made a great contribution to the development of software frameworks and systems. Therefore, we believe that performance

benchmark suites for GNNs similar to MLperf are vital to drive the rapid development of GNNs.

C. Distributed Training on Extreme-scale Hardware Platform

For distributed GNN training on large-scale graphs or even extreme-scale graphs, the performance can be greatly improved by using the extreme-scale hardware platform. The scale of the graph data has reached a staggering magnitude and is growing at an exaggerated rate. For example, Sogou [120] graph dataset has 12 trillion edges and Kronecker2 [121] graph dataset has 70 trillion edges. The training on graphs in this order of magnitude puts a very high requirement on the computing resources and memory resources of the computing system. Only by using the extreme-scale hardware platform can the training task be completed in a reasonable time. In this respect, the research of graph processing and distributed DNN training on the extreme-scale hardware platform can be used for reference. ShenTu [120] realizes processing graph with trillion edges in seconds by using petascale computing system. It suggests that the extreme-scale hardware platform can be used to accelerate distributed GNN training, which needs to explore and establish more hardware platform support for distributed GNN training.

D. Domain-specific Distributed Hardware Platform

There are many single-node domain-specific hardware platforms designed for GNNs. However, very few efforts are conducted on distributed hardware platforms. Recently, many single-node domain-specific hardware accelerators designed for GNNs have achieved significant improvement in performance and efficiency compared with the single-GPU platform. For example, HyGCN [86] proposes a hybrid architecture consisting of the Aggregation engine and Combination engine for the acceleration of the Aggregation operation and Combination operation respectively. Compared with NVIDIA V100 GPU, HyGCN achieves an average $6.5\times$ speedup with $10\times$ energy reduction, respectively. However, the single-node platform is not sufficient to handle the rapid development of GNNs since the ever-growing scale of graphs dramatically increases the training time, generating the demand for distributed GNN hardware platform. In response to that, MultiGCN [112] proposes a distributed hardware platform for the acceleration of GNN inference, which consists of multiple computing nodes to perform the inference in parallel. MultiGCN achieves $2.5\sim8\times$ speedup over the state-of-the-art multi-GPU software framework. But only a single work is not enough. We have witnessed that TPU-Pod [122], a domain-specific supercomputer for training DNNs, greatly accelerates the deployment of large-scale DNN models. Therefore, we believe that a domain-specific distributed hardware platform for GNNs similar to TPU-Pod is time to present, which is vital to drive the rapid development and deployment of GNNs.

E. General Communication Library for Distributed GNN Training

The variety of hardware platforms and irregular communication patterns of distributed GNN training lead to strong de-

mand for a general communication library to ease the programming effort on transmission planning while achieving high efficiency. The communication characteristics of hardware platforms are various due to the differences in the number of computing nodes, network topology, and communication bandwidth as well as communication latency of interconnection interface. In order to better adapt to the various transmission requirements of distributed GNN training on various hardware platforms, it is urgent to develop a general communication library which adapts to different network topologies, as well as various interconnection interfaces of different communication bandwidths and latency, so as to ease the implementation of novel communication patterns while ensuring efficiency.

X. CONCLUSION

In this paper, we comprehensively review distributed training of GNNs. After investigating recent efforts on distributed GNN training, we classify them in more detail according to their workflow, including the dispatch-workload-based execution as well as preset-workload-based execution of distributed full-batch training, and the individual-sample-based execution as well as joint-sample-based execution of distributed mini-batch training. Each taxonomy's workflow, computational pattern, and communication pattern are summarized, and various optimization techniques are further introduced to facilitate the understanding of recent research status, and to enable readers to quickly build up a big picture of distributed training of GNNs. In addition, this paper introduces the training software and hardware platforms, and then contrasts distributed GNN training with distributed DNN training. In the end, we provide several discussions about interesting issues and opportunities of distributed GNN training.

The emergence of distributed GNN training has successfully expanded the usage of GNNs on large-scale graphs, making it a powerful tool when learning from real-world associative relations. We optimistically look forward to new research which can further optimize this powerful tool on large-scale graph data and bring its application into the next level.

REFERENCES

[1] D. B. West *et al.*, *Introduction to graph theory*. Prentice hall Upper Saddle River, 2001, vol. 2.
[2] J. A. Bondy, U. S. R. Murty *et al.*, *Graph theory with applications*. Macmillan London, 1976, vol. 290.
[3] S. Wasserman, K. Faust *et al.*, "Social network analysis: Methods and applications," 1994.
[4] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 591–600. [Online]. Available: https://doi.org/10.1145/1772690.1772751

[5] J. J. McAuley and J. Leskovec, "Learning to discover social circles in ego networks," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., 2012, pp. 548–556. [Online]. Available: https://proceedings.neurips.cc/paper/2012/hash/7a614fd06c325499f1680b9896beedeb-Abstract.html

[6] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1247–1250.

[7] M. Ashburner, C. A. Ball, J. A. Blake, D. Botstein, H. Butler, J. M. Cherry, A. P. Davis, K. Dolinski, S. S. Dwight, J. T. Eppig *et al.*, "Gene ontology: tool for the unification of biology," *Nature genetics*, vol. 25, no. 1, pp. 25–29, 2000.

[8] G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[9] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 1, pp. 249–270, 2022. [Online]. Available: https://doi.org/10.1109/TKDE.2020.2981333

[10] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Networks Learn. Syst.*, vol. 32, no. 1, pp. 4–24, 2021. [Online]. Available: https://doi.org/10.1109/TNNLS.2020.2978386

[11] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020. [Online]. Available: https://doi.org/10.1016/j.aiopen.2021.01.001

[12] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[13] W. L. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 1024–1034. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/5dd9db5e033da9c6fb5ba83c7a7ebea9-Abstract.html

[14] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.

[15] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl

[16] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 5171–5181. [Online]. Available: https://proceedings.neurips.cc/paper/2018/hash/53f0d7c537d99b3824f0f99d62ea2428-Abstract.html

[17] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, S. A. McIlraith and K. Q. Weinberger, Eds. AAAI Press, 2018, pp. 4438–4445. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17146

[18] M. Schlichtkrull, T. N. Kipf, P. Bloem, I. v. d. Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European semantic web conference*. Springer, 2018, pp. 593–607.

[19] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing [review article]," *IEEE Comput. Intell. Mag.*, vol. 13, no. 3, pp. 55–75, 2018. [Online]. Available: https://doi.org/10.1109/MCI.2018.2840738

[20] M. Wu, S. Pan, X. Zhu, C. Zhou, and L. Pan, "Domain-adversarial graph neural networks for text classification," in *2019 IEEE International Conference on Data Mining, ICDM 2019, Beijing, China, November 8-11, 2019*, J. Wang, K. Shim, and X. Wu, Eds. IEEE, 2019, pp. 648–657. [Online]. Available: https://doi.org/10.1109/ICDM.2019.00075

[21] J. Bastings, I. Titov, W. Aziz, D. Marcheggiani, and K. Sima'an, "Graph convolutional encoders for syntax-aware neural machine translation," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, Sep. 2017, pp. 1957–1967. [Online]. Available: https://www.aclweb.org/anthology/D17-1209

[22] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, Y. Guo and F. Farooq, Eds. ACM, 2018, pp. 974–983. [Online]. Available: https://doi.org/10.1145/3219819.3219890

[23] W. Fan, Y. Ma, Q. Li, Y. He, Y. E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, L. Liu, R. W. White, A. Mantrach, F. Silvestri, J. J. McAuley, R. Baeza-Yates, and L. Zia, Eds. ACM, 2019, pp. 417–426. [Online]. Available: https://doi.org/10.1145/3308558.3313488

[24] C. Ma, L. Ma, Y. Zhang, J. Sun, X. Liu, and M. Coates, "Memory augmented graph neural networks for sequential recommendation," in *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 5045–5052. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/5945

[25] X. Chen, L.-J. Li, L. Fei-Fei, and A. Gupta, "Iterative visual reasoning beyond convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7239–7248.

[26] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi *et al.*, "A graph placement methodology for fast chip design," *Nature*, vol. 594, no. 7862, pp. 207–212, 2021.

[27] H. Wang, K. Wang, J. Yang, L. Shen, N. Sun, H.-S. Lee, and S. Han, "Gcn-rl circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[28] Y. Ma, H. Ren, B. Khailany, H. Sikka, L. Luo, K. Natarajan, and B. Yu, "High performance graph convolutional networks with applications in testability analysis," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[29] L. Oliver *et al.* Traffic prediction with advanced graph neural networks. [Online]. Available: https://deepmind.com/blog/article/traffic-prediction-with-advanced-graph-neural-networks

[30] C. Chen, K. Li, S. G. Teo, X. Zou, K. Wang, J. Wang, and Z. Zeng, "Gated residual recurrent graph neural networks for traffic prediction," in *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2019, pp. 485–492. [Online]. Available: https://doi.org/10.1609/aaai.v33i01.3301485

[31] M. Li and Z. Zhu, "Spatial-temporal fusion graph neural networks for traffic flow forecasting," in *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 2021, pp. 4189–4196. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/16542

[32] X. Geng, Y. Li, L. Wang, L. Zhang, Q. Yang, J. Ye, and Y. Liu, "Spatiotemporal multi-graph convolution network for ride-hailing demand forecasting," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 3656–3663.

[33] A. Li, Z. Qin, R. Liu, Y. Yang, and D. Li, "Spam review detection with graph convolutional networks," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019, pp. 2703–2711.

[34] A. Fout, J. Byrd, B. Shariat, and A. Ben-Hur, "Protein interface prediction using graph convolutional networks," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach,

R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 6530–6539. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/f507783927f2ec2737ba40afbd17efb5-Abstract.html

[35] 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.*, vol. 12, no. 12, pp. 2094–2105, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p2094-zhu.pdf

[36] 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.*, vol. 13, no. 12, pp. 3125–3137, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p3125-zhang.pdf

[37] L. Wang, Q. Yin, C. Tian, J. Yang, R. Chen, W. Yu, Z. Yao, and J. Zhou, "Flexgraph: a flexible and efficient distributed framework for GNN training," in *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, A. Barbalace, P. Bhatotia, L. Alvisi, and C. Cadar, Eds. ACM, 2021, pp. 67–82. [Online]. Available: https://doi.org/10.1145/3447786.3456229

[38] 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*. IEEE, 2020, pp. 36–44. [Online]. Available: https://doi.org/10.1109/IA351965.2020.00011

[39] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/fb60d411a5c5b72b2e7d3527cfc84fd0-Abstract.html

[40] A. Tripathy, K. A. Yelick, and A. Buluç, "Reducing communication in graph neural network training," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, C. Cuicchi, I. Qualters, and W. T. Kramer, Eds. IEEE/ACM, 2020, p. 70. [Online]. Available: https://doi.org/10.1109/SC41405.2020.00074

[41] J. Huang, W. Qin, X. Wang, and W. Chen, "Survey of external memory large-scale graph processing on a multi-core system," *J. Supercomput.*, vol. 76, no. 1, pp. 549–579, 2020. [Online]. Available: https://doi.org/10.1007/s11227-019-03023-0

[42] J. Huang, H. Wang, X. Fei, X. Wang, and W. Chen, "$tc-stream$tc-stream: Large-scale graph triangle counting on a single machine using gpus," *IEEE Trans. Parallel Distributed Syst.*, vol. 33, no. 11, pp. 3067–3078, 2022. [Online]. Available: https://doi.org/10.1109/TPDS.2021.3135329

[43] "Global social media ranking— statistic." [Online]. Available: https://www.statista.com/statistics

[44] 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 *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafrir, Eds. USENIX Association, 2019, pp. 443–458. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/ma

[45] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the accuracy, scalability, and performance of graph neural networks with roc," in *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, I. S. Dhillon, D. S. Papailiopoulos, and V. Sze, Eds. mlsys.org, 2020. [Online]. Available: https://proceedings.mlsys.org/book/300.pdf

[46] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu, "DGCL: an efficient communication library for distributed GNN training," in *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, A. Barbalace, P. Bhatotia, L. Alvisi, and C. Cadar, Eds. ACM, 2021, pp. 130–144. [Online]. Available: https://doi.org/10.1145/3447786.3456233

[47] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. D. Kalamkar, N. K. Ahmed, and S. Avancha, "Distgnn: scalable distributed training for large-scale graph neural networks," in *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*, B. R. de Supinski, M. W. Hall, and T. Gamblin, Eds. ACM, 2021, pp. 76:1–76:14. [Online]. Available: https://doi.org/10.1145/3458817.3480856

[48] M. F. Balin, K. Sancak, and Ü. V. Çatalyürek, "MG-GCN: scalable multi-gpu GCN training framework," *CoRR*, vol. abs/2110.08688, 2021. [Online]. Available: https://arxiv.org/abs/2110.08688

[49] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim, and G. H. Xu, "Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads," in *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, A. D. Brown and J. R. Lorch, Eds. USENIX Association, 2021, pp. 495–514. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/thorpe

[50] H. Mostafa, "Sequential aggregation and rematerialization: Distributed full-batch training of graph neural networks on large graphs," *CoRR*, vol. abs/2111.06483, 2021. [Online]. Available: https://arxiv.org/abs/2111.06483

[51] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, "Pagraph: Scaling GNN training on large graphs via computation-aware caching," in *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, R. Fonseca, C. Delimitrou, and B. C. Ooi, Eds. ACM, 2020, pp. 401–415. [Online]. Available: https://doi.org/10.1145/3419111.3421281

[52] L. Zhang, Z. Lai, S. Li, Y. Tang, F. Liu, and D. Li, "2pgraph: Accelerating GNN training over large graphs on GPU clusters," in *IEEE International Conference on Cluster Computing, CLUSTER 2021, Portland, OR, USA, September 7-10, 2021*. IEEE, 2021, pp. 103–113. [Online]. Available: https://doi.org/10.1109/Cluster48925.2021.00036

[53] M. Ramezani, W. Cong, M. Mahdavi, M. T. Kandemir, and A. Sivasubramaniam, "Learn locally, correct globally: A distributed algorithm for training graph neural networks," *CoRR*, vol. abs/2111.08202, 2021. [Online]. Available: https://arxiv.org/abs/2111.08202

[54] D. Zheng, X. Song, C. Yang, D. LaSalle, and G. Karypis, "Distributed hybrid CPU and GPU training for graph neural networks on billion-scale heterogeneous graphs," in *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*, A. Zhang and H. Rangwala, Eds. ACM, 2022, pp. 4582–4591. [Online]. Available: https://doi.org/10.1145/3534678.3539177

[55] H. Li, Y. Liu, Y. Li, B. Huang, P. Zhang, G. Zhang, X. Zeng, K. Deng, W. Chen, and C. He, "Graphtheta: A distributed graph neural network learning system with flexible training strategy," *CoRR*, vol. abs/2104.10569, 2021. [Online]. Available: https://arxiv.org/abs/2104.10569

[56] T. Kaler, N. Stathas, A. Ouyang, A. Iliopoulos, T. B. Schardl, C. E. Leiserson, and J. Chen, "Accelerating training and inference of graph neural networks with fast sampling and pipelining," *CoRR*, vol. abs/2110.08450, 2021. [Online]. Available: https://arxiv.org/abs/2110.08450

[57] S. Gandhi and A. P. Iyer, "P3: distributed deep graph learning at scale," in *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, A. D. Brown and J. R. Lorch, Eds. USENIX Association, 2021, pp. 551–568. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/gandhi

[58] Z. Luo, Y. Bao, and C. Wu, "Optimizing task placement and online scheduling for distributed GNN training acceleration," in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications, London, United Kingdom, May 2-5, 2022*. IEEE, 2022, pp. 890–899. [Online]. Available: https://doi.org/10.1109/INFOCOM48880.2022.9796910

[59] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón, "Computing graph neural networks: A survey from algorithms to accelerators," *ACM Comput. Surv.*, vol. 54, no. 9, pp. 191:1–191:38, 2022. [Online]. Available: https://doi.org/10.1145/3477141

[60] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. F. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, Ç. Gülçehre, H. F. Song, A. J. Ballard, J. Gilmer, G. E. Dahl, A. Vaswani, K. R. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, "Relational inductive biases, deep learning, and graph networks," *CoRR*, vol. abs/1806.01261, 2018. [Online]. Available: http://arxiv.org/abs/1806.01261

[61] I. Chami, S. Abu-El-Haija, B. Perozzi, C. Ré, and K. Murphy, "Machine learning on graphs: A model and comprehensive taxonomy," *CoRR*, vol. abs/2005.03675, 2020. [Online]. Available: https://arxiv.org/abs/2005.03675

[62] L. C. Lamb, A. S. d'Avila Garcez, M. Gori, M. O. R. Prates, P. H. C. Avelar, and M. Y. Vardi, "Graph neural networks meet neural-symbolic

computing: A survey and perspective," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, C. Bessiere, Ed. ijcai.org, 2020, pp. 4877–4884. [Online]. Available: https://doi.org/10.24963/ijcai.2020/679

[63] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. v. d. Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European semantic web conference*. Springer, 2018, pp. 593–607.

[64] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[65] S. Yan, Y. Xiong, and D. Lin, "Spatial temporal graph convolutional networks for skeleton-based action recognition," in *Thirty-second AAAI conference on artificial intelligence*, 2018.

[66] Z. Wu, S. Pan, G. Long, J. Jiang, X. Chang, and C. Zhang, "Connecting the dots: Multivariate time series forecasting with graph neural networks," in *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, R. Gupta, Y. Liu, J. Tang, and B. A. Prakash, Eds. ACM, 2020, pp. 753–763. [Online]. Available: https://doi.org/10.1145/3394486.3403118

[67] M. Wu, S. Pan, L. Du, and X. Zhu, "Learning graph neural networks with positive and unlabeled nodes," *ACM Trans. Knowl. Discov. Data*, vol. 15, no. 6, pp. 101:1–101:25, 2021. [Online]. Available: https://doi.org/10.1145/3450316

[68] S. Zhu, C. Zhou, S. Pan, X. Zhu, and B. Wang, "Relation structure-aware heterogeneous graph neural network," in *2019 IEEE International Conference on Data Mining, ICDM 2019, Beijing, China, November 8-11, 2019*, J. Wang, K. Shim, and X. Wu, Eds. IEEE, 2019, pp. 1534–1539. [Online]. Available: https://doi.org/10.1109/ICDM.2019.00203

[69] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl

[70] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ

[71] J. Chen, T. Ma, and C. Xiao, "Fastgcn: Fast learning with graph convolutional networks via importance sampling," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=rytstxWAW

[72] W. Huang, T. Zhang, Y. Rong, and J. Huang, "Adaptive sampling towards fast graph representation learning," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 4563–4572. [Online]. Available: https://proceedings.neurips.cc/paper/2018/hash/01eee509ee2f68dc6014898c309e86bf-Abstract.html

[73] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. K. Prasanna, "Graphsaint: Graph sampling based inductive learning method," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=BJe8pkHFwS

[74] J. Dong, D. Zheng, L. F. Yang, and G. Karypis, "Global neighbor sampling for mixed CPU-GPU training on giant graphs," in *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, F. Zhu, B. C. Ooi, and C. Miao, Eds. ACM, 2021, pp. 289–299. [Online]. Available: https://doi.org/10.1145/3447548.3467437

[75] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *19th International Conference on Computational Statistics, COMPSTAT 2010, Paris, France, August 22-27, 2010 - Keynote, Invited and Contributed Papers*, Y. Lechevallier and G. Saporta, Eds. Physica-Verlag, 2010, pp. 177–186. [Online]. Available: https://doi.org/10.1007/978-3-7908-2604-3_16

[76] X. Liu, M. Yan, L. Deng, G. Li, X. Ye, and D. Fan, "Sampling methods for efficient training of graph convolutional networks: A survey," *IEEE CAA J. Autom. Sinica*, vol. 9, no. 2, pp. 205–234, 2022. [Online]. Available: https://doi.org/10.1109/JAS.2021.1004311

[77] H. Dai, Z. Kozareva, B. Dai, A. J. Smola, and L. Song, "Learning steady-states of iterative algorithms over graphs," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, ser. Proceedings of Machine Learning Research, J. G. Dy and A. Krause, Eds., vol. 80. PMLR, 2018, pp. 1114–1122. [Online]. Available: http://proceedings.mlr.press/v80/dai18a.html

[78] J. Chen, J. Zhu, and L. Song, "Stochastic training of graph convolutional networks with variance reduction," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, ser. Proceedings of Machine Learning Research, J. G. Dy and A. Krause, Eds., vol. 80. PMLR, 2018, pp. 941–949. [Online]. Available: http://proceedings.mlr.press/v80/chen18p.html

[79] D. Zou, Z. Hu, Y. Wang, S. Jiang, Y. Sun, and Q. Gu, "Layer-dependent importance sampling for training deep and large graph convolutional networks," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 11 247–11 256. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/91ba4a4478a66bee9812b0804b6f9d1b-Abstract.html

[80] W. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C. 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 2019, Anchorage, AK, USA, August 4-8, 2019*, A. Teredesai, V. Kumar, Y. Li, R. Rosales, E. Terzi, and G. Karypis, Eds. ACM, 2019, pp. 257–266. [Online]. Available: https://doi.org/10.1145/3292500.3330925

[81] J. Bai, Y. Ren, and J. Zhang, "Ripple walk training: A subgraph-based training framework for large and deep graph neural network," in *International Joint Conference on Neural Networks, IJCNN 2021, Shenzhen, China, July 18-22, 2021*. IEEE, 2021, pp. 1–8. [Online]. Available: https://doi.org/10.1109/IJCNN52387.2021.9533204

[82] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. K. Prasanna, "Accurate, efficient and scalable graph embedding," in *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*. IEEE, 2019, pp. 462–471. [Online]. Available: https://doi.org/10.1109/IPDPS.2019.00056

[83] H. Lin, M. Yan, X. Yang, M. Zou, W. Li, X. Ye, and D. Fan, "Characterizing and understanding distributed GNN training on gpus," *IEEE Comput. Archit. Lett.*, vol. 21, no. 1, pp. 21–24, 2022. [Online]. Available: https://doi.org/10.1109/LCA.2022.3168067

[84] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *CoRR*, vol. abs/1903.02428, 2019. [Online]. Available: http://arxiv.org/abs/1903.02428

[85] 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*, vol. abs/1909.01315, 2019. [Online]. Available: http://arxiv.org/abs/1909.01315

[86] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A GCN accelerator with hybrid architecture," in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*. IEEE, 2020, pp. 15–29. [Online]. Available: https://doi.org/10.1109/HPCA47549.2020.00012

[87] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 2016, pp. 56:1–56:13. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783759

[88] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, pp. 615–628. [Online]. Available: https://doi.org/10.1145/3352460.3358318

[89] Z. Jia, Y. Kwon, G. M. Shipman, P. S. McCormick, M. Erez, and A. Aiken, "A distributed multi-gpu system for fast graph processing," *Proc. VLDB Endow.*, vol. 11, no. 3, pp. 297–310, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol11/p297-jia.pdf

[90] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao, "Tigr: Transforming irregular

graphs for gpu-friendly graph processing," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 622–636, 2018.

[91] G. Karypis and V. Kumar, "Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0," *Applied Physics Letters, Volume 97, Issue 12, id. 124101 (3 pages)(2010)*, 1995.

[92] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, K. Keeton and T. Roscoe, Eds. USENIX Association, 2016, pp. 301–316. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu

[93] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Exploiting bounded staleness to speed up big data analytics," in *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, G. Gibson and N. Zeldovich, Eds. USENIX Association, 2014, pp. 37–48. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/cui

[94] B. Recht, C. Ré, S. J. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. C. N. Pereira, and K. Q. Weinberger, Eds., 2011, pp. 693–701. [Online]. Available: https://proceedings.neurips.cc/paper/2011/hash/218a0aefd1d1a4be65601cc6ddc1520e-Abstract.html

[95] X. Zhao, A. An, J. Liu, and B. X. Chen, "Dynamic stale synchronous parallel distributed training for deep learning," in *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*. IEEE, 2019, pp. 1507–1517. [Online]. Available: https://doi.org/10.1109/ICDCS.2019.00150

[96] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing, "More effective distributed ML via a stale synchronous parallel parameter server," in *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, C. J. C. Burges, L. Bottou, Z. Ghahramani, and K. Q. Weinberger, Eds., 2013, pp. 1223–1231. [Online]. Available: https://proceedings.neurips.cc/paper/2013/hash/b7bb35b9c6ca2aee2df08cf09d7016c2-Abstract.html

[97] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *CoRR*, vol. abs/1604.06174, 2016. [Online]. Available: http://arxiv.org/abs/1604.06174

[98] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, K. Keutzer, I. Stoica, and J. Gonzalez, "Checkmate: Breaking the memory wall with optimal tensor rematerialization," in *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, I. S. Dhillon, D. S. Papailiopoulos, and V. Sze, Eds. mlsys.org, 2020. [Online]. Available: https://proceedings.mlsys.org/book/320.pdf

[99] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[100] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998. [Online]. Available: https://doi.org/10.1137/S1064827595287997

[101] I. S. Dhillon, Y. Guan, and B. Kulis, "Weighted graph cuts without eigenvectors A multilevel approach," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 11, pp. 1944–1957, 2007. [Online]. Available: https://doi.org/10.1109/TPAMI.2007.1115

[102] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, C. Thekkath and A. Vahdat, Eds. USENIX Association, 2012, pp. 17–30. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[103] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2d graph partitioning," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, W. Gropp and S. Matsuoka, Eds. ACM, 2013, pp. 50:1–50:12. [Online]. Available: https://doi.org/10.1145/2503210.2503293

[104] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012*, Q. Yang, D. Agarwal, and J. Pei, Eds. ACM, 2012, pp. 1222–1230. [Online]. Available: https://doi.org/10.1145/2339530.2339722

[105] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Math.*, vol. 6, no. 1, pp. 29–123, 2009. [Online]. Available: https://doi.org/10.1080/15427951.2009.10129177

[106] H. Liu, S. Lu, X. Chen, and B. He, "G3: when graph neural networks meet parallel graph processing systems on gpus," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 2813–2816, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p2813-liu.pdf

[107] M. Garland and D. B. Kirk, "Understanding throughput-oriented architectures," *Communications of the ACM*, vol. 53, no. 11, pp. 58–66, 2010.

[108] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE micro*, vol. 28, no. 2, pp. 39–55, 2008.

[109] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[110] K. S. Chahal, M. S. Grover, K. Dey, and R. R. Shah, "A hitchhiker's guide on distributed training of deep neural networks," *J. Parallel Distributed Comput.*, vol. 137, pp. 65–76, 2020. [Online]. Available: https://doi.org/10.1016/j.jpdc.2019.10.004

[111] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., 2012, pp. 1106–1114. [Online]. Available: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

[112] G. Sun, M. Yan, D. Wang, H. Li, W. Li, X. Ye, D. Fan, and Y. Xie, "Multi-node acceleration for large-scale gcns," *IEEE transactions on computers*, 2022.

[113] M. Yan, Z. Chen, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Characterizing and understanding gcns on gpu," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 22–25, 2020.

[114] Z. Zhang, J. Leng, L. Ma, Y. Miao, C. Li, and M. Guo, "Architectural implications of graph neural networks," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 59–62, 2020.

[115] M. Yan, M. Zou, X. Yang, W. Li, X. Ye, D. Fan, and Y. Xie, "Characterizing and understanding hgnns on gpus," *IEEE Computer Architecture Letters*, pp. 1–4, 2022.

[116] H. Lin, M. Yan, X. Yang, M. Zou, W. Li, X. Ye, and D. Fan, "Characterizing and understanding distributed gnn training on gpus," *IEEE Computer Architecture Letters*, vol. 21, no. 1, pp. 21–24, 2022.

[117] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen, "Understanding and bridging the gaps in current GNN performance optimizations," in *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, J. Lee and E. Petrank, Eds. ACM, 2021, pp. 119–132. [Online]. Available: https://doi.org/10.1145/3437801.3441585

[118] H. Zhang, Z. Yu, G. Dai, G. Huang, Y. Ding, Y. Xie, and Y. Wang, "Understanding GNN computational graph: A coordinated computation, io, and memory perspective," *CoRR*, vol. abs/2110.09524, 2021. [Online]. Available: https://arxiv.org/abs/2110.09524

[119] P. Mattson, V. J. Reddi, C. Cheng, C. Coleman, G. Diamos, D. Kanter, P. Micikevicius, D. Patterson, G. Schmuelling, H. Tang, G.-Y. Wei, and C.-J. Wu, "Mlperf: An industry standard benchmark suite for machine learning performance," *IEEE Micro*, vol. 40, no. 2, pp. 8–16, 2020.

[120] H. Lin, X. Zhu, B. Yu, X. Tang, W. Xue, W. Chen, L. Zhang, T. Hoefler, X. Ma, X. Liu, W. Zheng, and J. Xu, "Shentu: processing multi-trillion edge graphs on millions of cores in seconds," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE / ACM, 2018, pp. 56:1–56:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291731

[121] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication," in *Knowledge Discovery in Databases: PKDD 2005, 9th European Conference on Principles and Practice of Knowledge Discovery in Databases, Porto, Portugal, October*

3-7, 2005, Proceedings, ser. Lecture Notes in Computer Science, A. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, Eds., vol. 3721. Springer, 2005, pp. 133–145. [Online]. Available: https://doi.org/10.1007/11564126_17

[122] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, 2020.

XI. Biography Section

Haiyang Lin received the B.S. degree from Shanghai Jiaotong University, Shanghai, China, in 2018, and currently working toward the Ph.D. degree at Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His current research interests include distributed training, graph-based hardware accelerator, and high-throughput computer architecture.

Mingyu Yan received the Ph.D. degree from University of Chinese Academy of Sciences, Beijing, China in 2020. He is currently an Associate Professor at Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His current research interests include graph-based machine learning, domain-specific hardware architecture for graph-based machine learning, domain-specific distributed system for graph-based machine learning, and high-throughput computer architecture. To date, Dr. Yan has published over 20 research papers in top-tier journals and conferences, including the MICRO, HPCA, IJCAI, DAC, ICCAD, IEEE Transactions on Computers (TC), IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), IEEE/CAA Journal of Automatica Sinica (IEEE/CAA JAS), and IEEE Journal of Selected Topics in Signal Processing (IEEE J-STSP).

Xiaochun Ye received the Ph.D. degree in computer architecture from Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2010. He is currently a Professor at Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His main research interest is domain-specific hardware architecture for graph-based machine learning and high-throughput computer architecture. To date, Dr. Ye has published over 90 research papers in top-tier journals and conferences, the MICRO, HPCA, IJCAI, PACT, IPDPS, DAC, ICCAD, DATE, HOT CHIPS, ICCV, IEEE Transactions on Computers (TC), IEEE/CAA Journal of Automatica Sinica (IEEE/CAA JAS), and IEEE Micro.
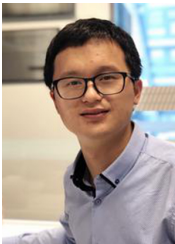
Dongrui Fan received the Ph.D. degree in computer architecture from Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2005. He is currently a Professor and Ph.D. Supervisor at Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His main research interests include high-throughput computer architecture, high-performance computer architecture, and low-power design. To date, Dr. Fan has published over 140 research papers in top-tier journals and conferences, including the IEEE Transactions on Parallel and Distributed Systems (TPDS), IEEE Transactions on Computers (TC), IEEE micro, Journal of Parallel and Distributed Computing (JPDC), MICRO, HPCA, PPoPP, IJCAI, DAC, ICCAD, DATE, ICCV, PACT, HOT CHIPS. He has been a member of the program committee of many important academic conferences, including MICRO, HPCA, etc.

Wenguang Chen received the B.S. and Ph.D. degrees in computer science from Tsinghua University, in 1995 and 2000, respectively. He was the CTO with Opportunity International Inc. from 2000-2002. Since January 2003, he joined Tsinghua University. He is currently a professor and associate head with the Department of Computer Science and Technology, Tsinghua University. He is chair of ACM China Council, chief editor of ACM Communications Chinese Edition. He joined Ant Group in 2020 as the head of graph processing technology. Since 2022, he has served as the President of Ant Technology Research Institute.
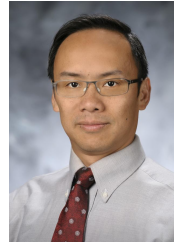
Dr. Chen is an expert in parallel and distributed systems for extreme-scale graph processing. To date, Dr. Chen has published over 100 research papers in top-tier conferences and journals, including the OSDI, PPoPP, SC, ASPLOS, ICS, EuroSys, IPDPS, MICRO, ATC, CGO, VLDB, PACT, Nature, IEEE Transactions on Parallel and Distributed Systems (TPDS), and IEEE Transactions on Computers (TC). Their work in extreme-scale graph processing, i.e., "ShenTu: Processing Multi-trillion Edge Graphs on Millions of Cores in Seconds", was a finalist for the ACM Gordon Bell Prize in 2018. The Gordon Bell Prize is commonly referred to as the Nobel Prize of Supercomputing. He has been a member of the program committee of many important academic conferences, including OSDI, SOSP, PLDI, PPoPP, SC, ASPLOS, TPDS, TC, ATC, CGO, IPDPS, CCGrid, ICPP, PACT, EuroSys, ICS, etc.

Yuan Xie (Fellow, IEEE) received the B.S. degree in electronic engineering from Tsinghua University, Beijing, China, in 1997, and the M.S. and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ, USA, in 1999 and 2002, respectively. He was an Advisory Engineer with the IBM Microelectronic Division, Burlington, NJ, USA, from 2002 to 2003. He was a Full Professor with Pennsylvania State University, State College, PA, USA, from 2003 to 2014. He was a Visiting Researcher with the Interuniversity Microelectronics Center (IMEC), Louvain, Belgium, from 2005 to 2007 and in 2010. He was a Senior Manager and a Principal Researcher with the AMD Research China Laboratory, Beijing, from 2012 to 2013. He is currently a Professor with the Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA, USA. His research interests include VLSI design, electronics design automation (EDA), computer architecture, and embedded systems.

Dr. Xie serves as a Committee Member for the IEEE Design Automation Technical Committee (DATC). He is a Fellow of AAAS/ACM and an expert in computer architecture who has been inducted into ISCA/MICRO/HPCA Hall of Fame. He was a recipient of the Best Paper Awards at HPCA 2015, ICCAD 2014, GLSVLSI 2014, ISVLSI 2012, ISLPED 2011, ASPDAC 2008, and ASICON 2001; the Best Paper Nominations at ASPDAC 2014, MICRO 2013, DATE 2013, ASPDAC 2010–2009, and ICCAD 2006; the 2016 IEEE Micro Top Picks Award at the 2008 IBM Faculty Award, and the 2006 NSF CAREER Award. He has served as the TPC Chair for ICCAD 2019, HPCA 2018, ASPDAC 2013, ISLPED 2013, and MPSOC 2011. He serves as the Editor-in-Chief for ACM Journal on Emerging Technologies in Computing Systems and as an Associate Editor for the ACM Transactions on Design Automation for Electronics Systems, the IEEE Transactions on Computers, the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, the IEEE Transactions on VLSI, IEEE Design and Test, and IET Computers and Design Techniques. Through extensive collaboration with industry partners, AMD, HP, Honda, IBM, Intel, Google, Samsung, IMEC, Qualcomm, Alibaba, Seagate, Toyota, and so on, he has helped the transition of research ideas to industry.

Shirui Pan received a Ph.D. in computer science from the University of Technology Sydney (UTS), Ultimo, NSW, Australia. He is a Professor at the School of Information and Communication Technology, Griffith University, Australia. Prior to this, he was a Senior Lecturer with the Faculty of IT, Monash University, Australia. His research interests include data mining and machine learning. To date, Dr. Pan has published over 150 research papers in top-tier journals and conferences, including the IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI), IEEE Transactions on Knowledge and Data Engineering (TKDE), IEEE Transactions on Neural Networks and Learning Systems (TNNLS), ICML, NeurIPS, KDD, AAAI, IJCAI, WWW, CVPR, ICDE, and ICDM. His research has attracted over 10,000 citations.

Dr. Pan is an expert in graph neural networks (GNNs). He is recognized as one of the AI 2000 AAAI/IJCAI Most Influential Scholars in Australia (2021, 2022), and one of the World's Top 2% Scientists (2021). His research on GNNs received the IEEE ICDM Best Student Paper Award (2020), and the JCDL Best Paper Honorable Mention Award (2020). He has seven papers on GNNs recognized as the Most Influential Papers in KDD (x1), IJCAI (x5), and CIKM (x1) (Feb 2022). In particular, his survey paper on *"A Comprehensive Survey on Graph Neural Networks"* in TNNLS-21 has been cited over 4,100 times; his research on "Graph WaveNet" (582 citations) is the most cited research at IJCAI-2019 as of September 2022. He serves as an Area Chair/Senior Program Committee member (SPC) for ICDM, AAAI, and IJCAI, and a PC member for NeurIPS, KDD, WWW, etc. He is an Australian Research Council (ARC) Future Fellow.