Graph neural networks (GNNs) are a type of deep learning models that are trained on graphs and have been successfully applied in various domains. Despite the effectiveness of GNNs, it is still challenging for GNNs to efficiently scale to large graphs. As a remedy, distributed computing becomes a promising solution of training large-scale GNNs, since it is able to provide abundant computing resources. However, the dependency of graph structure increases the difficulty of achieving high-efficiency distributed GNN training, which suffers from the massive communication and workload imbalance. In recent years, many efforts have been made on distributed GNN training, and an array of training algorithms and systems have been proposed. Yet, there is a lack of systematic review of the optimization techniques for the distributed execution of GNN training. In this survey, we analyze three major challenges in distributed GNN training: massive feature communication, the loss of model accuracy, and workload imbalance. Then, we introduce a new taxonomy for the optimization techniques in distributed GNN training that address the above challenges. The new taxonomy classifies existing techniques into four categories: GNN data partition, GNN batch generation, GNN execution model, and GNN communication protocol. We carefully discuss the techniques in each category. In the conclusion, we summarize existing distributed GNN systems for multi–graphics processing units (GPUs), GPU-clusters.
and central processing unit (CPU)-clusters, respectively, and present a discussion about the future direction of distributed GNN training.

CCS Concepts: • General and reference → Surveys and overviews; • Computing methodologies → Distributed computing methodologies; Machine learning; Neural networks; • Mathematics of computing → Graph algorithms

Additional Key Words and Phrases: Surveys and overviews, distributed GNN training, graph data management, communication optimization, distributed GNN systems

ACM Reference Format:
Yingxia Shao, Hongzheng Li, Xizhi Gu, Hongbo Yin,Yawen Li, Xupeng Miao, Wentao Zhang, Bin Cui, and Lei Chen. 2024. Distributed Graph Neural Network Training: A Survey. ACM Comput. Surv. 56, 8, Article 191 (April 2024), 39 pages. https://doi.org/10.1145/3648358

1 INTRODUCTION

Graph neural networks (GNNs) are powerful tools to handle problems modeled by the graph and have been widely adopted in various applications, including social networks (e.g., social spammer detection [102, 132], social network analysis [107]), bio-informatics (e.g., protein interface prediction [40], disease–gene association [97]), drug discovery [12, 79], traffic forecasting [67], health care [3, 25], recommendation [37, 55, 61, 131, 135], natural language processing [156, 168], and others [31, 99, 155, 163, 166]. By integrating the information of graph structure into the deep learning models, GNNs can achieve significantly better results than traditional machine learning and data mining methods.

A GNN model generally contains multi-graph convolutional layers, where each vertex aggregates the latest states of its neighbors, updates the state of the vertex, and applies a neural network to the updated state of the vertex. Taking the traditional graph convolutional network (GCN) as an example, in each layer, a vertex uses a sum function to aggregate the neighbor states and its own state, then applies a single-layer Multi-layer Perceptron (MLP) to transform the new state. Such procedures are repeated $L$ times if the number of layers is $L$. The vertex states generated in the $L$th layer are used by the downstream tasks, such as node classification, link prediction, and so on. In the past, many research works have made remarkable progress in the design of GNN models. Prominent models include GCN [128], GraphSAGE [52], GAT [112], GIN [138], and many other application-specific GNN models [150, 152, 167]. To date, there are tens of surveys reviewing the GNN models [134, 136, 157, 170]. On the other hand, to efficiently develop different GNN models, many GNN-oriented frameworks have been proposed based on various deep learning libraries [9, 17, 39, 50, 81, 123]. Many new optimizations have been proposed to speed up GNN training, including GNN computation kernels [43, 58, 59, 95, 109, 151], efficient programming models [57, 133, 137], and full utilization of new hardware [23, 48, 146, 171]. However, these frameworks and optimizations mainly focus on training a GNN on a single machine, while not paying much attention to the scalability of input graphs.

Nowadays, large-scale GNNs [45, 68, 83] are a hot topic because of the prevalence of massively large graph data. It is common to have graphs with billions of vertices and trillions of edges, such as the social networks in Sina Weibo, WeChat, Twitter, and Meta. However, most of the existing GNN models have been tested only on small graph datasets, and it is impossible or inefficient to process large graph datasets [56]. This is because GNN models are complex and require massive computation resources when handling large graphs. A line of works achieve large-scale GNNs by designing scalable GNN models. They use simplification [41, 53, 130], quantization [5, 38, 60, 84, 106, 119, 120, 126, 161], sampling [24, 145, 147] and distillation [30, 141, 153] to design efficient models. Another line of works adopt distributed computing for GNN training, i.e.,
Distributed GNN training, because when handling large graphs, the limited memory and computing resource of a single device, e.g., a graphics processing unit (GPU), become the bottleneck of large-scale GNN training, and distributed computing provides more computing resources, e.g., multi-GPUs and central processing unit (CPU) clusters, to improve training efficiency. However, previous systems targeted distributed graph processing [85, 87] and distributed deep learning [2, 78] separately. Graph processing systems do not consider the acceleration of neural network operations, whereas deep learning systems lack the ability to process graph data. Therefore, many efforts have been made in designing efficient distributed GNN training frameworks and systems [65, 116, 117, 164, 173].

In this survey, we focus on the works and specific techniques proposed for distributed GNN training. Distributed GNN training divides the whole workload of model training among a set of workers, and all the workers process the workload in parallel. However, due to the data dependency in GNNs, it is non-trivial to apply existing distributed machine learning methods [113, 122] to GNNs. Thus, many new techniques for optimizing the distributed GNN training pipeline have been proposed. Although there are a lot of surveys [134, 157, 170] about GNNs, to the best of our knowledge, little effort has been made to systematically review the techniques for distributed GNN training. Recently, Besta and Hoefler [10] only reviewed the parallel computing paradigm of GNN, Abadal et al. [1] surveyed GNN computing from algorithms to hardware accelerators, and Vatter et al. [111] provided a comprehensive overview of the evolution of the distributed systems for scalable GNN training.

To clearly organize the techniques for distributed GNN training, we introduce a general distributed GNN training pipeline that consists of three stages: data partition, batch generation, and GNN model training. These stages involve GNN-specific execution logic that includes graph processing and graph aggregation. In the context of this general distributed GNN training pipeline, we discuss three main challenges of distributed GNN training that are caused by the data dependency in graph data and require new techniques specifically designed for distributed GNN training. To help readers understand various optimization techniques that address the above challenges better, we introduce a new taxonomy that classifies the techniques into four orthogonal categories: GNN data partition, GNN batch generation, GNN execution model, and GNN communication protocol. This taxonomy not only covers the optimization techniques used in both mini-batch distributed GNN training and full-graph distributed GNN training but also discusses the techniques from graph processing to model execution. We carefully review the existing techniques in each category, followed by describing 28 representative distributed GNN systems and frameworks either from industry or academia. Finally, we briefly discuss the future directions of distributed GNN training.

The contributions of this survey are as follows:

- This is the first survey focusing on the optimization techniques for efficient distributed GNN training, which helps researchers quickly understand the landscape of distributed GNN training.
- We introduce a new taxonomy of distributed GNN training techniques by considering the life-cycle of end-to-end distributed GNN training. At the highest level, the new taxonomy consists of four orthogonal categories: GNN data partition, GNN batch generation, GNN execution model, and GNN communication protocol.
- We provide a detailed and comprehensive technical summary of each category in our new taxonomy.
- We review 28 representative distributed GNN training systems and frameworks from industry to academia.
- We discuss the future directions of the distributed GNN training.

ACM Comput. Surv., Vol. 56, No. 8, Article 191. Publication date: April 2024.
Survey organization. In Section 2, we introduce the background of GNNs and discuss the differences between GNN training and traditional neural network training. In Section 3, we present the distributed GNN training pipeline, highlight the specific challenges faced by distributed GNN training, and introduce our taxonomy of the techniques used in distributed GNN training. On the basis of the taxonomy, we present a detailed discussion on the techniques of distributed GNN training, including data partition (Section 4), batch generation (Section 5), execution model (Section 6), and communication protocol (Section 7). In Section 8, we discuss various existing distributed GNN training systems. We unveil some promising future directions for distributed GNN training in Section 9 and present our conclusions in Section 10.

2 PRELIMINARIES OF GRAPH NEURAL NETWORKS

Graph and Graph Partition. A graph \( G = (V, E) \) consists of a vertex set \( V \) and an edge set \( E \). The set of neighbors of a vertex \( v \in V \) is denoted by \( N(v) \) and the degree of \( v \) is denoted by \( d = |N(v)| \). In the directed graph, the set of incoming and outgoing neighbors of a vertex \( v \in V \) is denoted by \( N_{in}(v) \) and \( N_{out}(v) \), respectively, and the corresponding degree is denoted by \( d_{in} = |N_{in}(v)| \) and \( d_{out} = |N_{out}(v)| \). An edge \( e \in E \) is represented by \( (u, v) \), where \( u, v \in V \). The adjacent matrix of \( G \) is represented by \( A \), where an entry \( a_{uv} \) of \( A \) equals 1 when \( (u, v) \in E \); otherwise, \( a_{uv} = 0 \). Each vertex \( v \) in a graph has an initial feature vector \( x_v \in \mathbb{R}^D \), where \( D \) is the dimension size of feature vectors, and the feature matrix of the graph is denoted by \( X \).

A graph can be partitioned in distributed settings. For an edge-cut graph partition, a graph \( G = (V, E) \) is divided into \( P \) partitions \( \{G_i = (V_i, E_i)\} \), \( 1 \leq i, j \leq P \), which satisfy \( V = \bigcup V_i, V_i \cap V_j = \emptyset, 1 \leq i, j \leq P \). The end vertices of cross edges are called boundary vertices, and other vertices in \( G \) are inner vertices. For a vertex-cut graph partition, the above \( P \) partitions should satisfy \( E = \bigcup E_i, E_i \cap E_j = \emptyset, 1 \leq i, j \leq P \). A vertex can be replicated among partitions and the number of replications of vertex \( v \) is called the replication factor. When the replication factor of a vertex is larger than 1, the vertex is called a boundary vertex, and other vertices in \( G \) are inner vertices.

Graph Neural Networks (GNNs). Given a graph \( G \) with adjacent matrix \( A \) and the feature matrix \( X \), where each row is the initial feature vector \( x_v \) of a vertex \( v \) in the graph \( G \), an \( l \)-th layer in a GNN updates the vertex feature by aggregating the features from the corresponding neighborhoods. The process can be formalized in matrix view as follows:

\[
H^l = \sigma(\tilde{A}H^{l-1}W^{l-1}),
\]

(1)

where \( H^l \) is the hidden embedding matrix and \( H^0 = X \), \( W^{l-1} \) is the model weights, \( \tilde{A} \) is a normalized \( A \), and \( \sigma \) is a non-linear function, e.g., ReLU, Sigmoid, etc. Equation (1) is the global view of GNN computation. The local view of GNN computation is the computation of a single vertex. Given a vertex \( v \), the local computation of the \( l \)-th layer in the message passing schema [47] can be formalized as follows:

\[
h^l_v = \sigma(\bigoplus_{u \in N(v)} (\phi(h^{l-1}_{uv}, h^{l-1}_u, h^{l-1}_{e_{uv}}))),
\]

(2)

where \( \oplus \) is an aggregation function, \( \phi \) is an update function, and \( h^{l-1}_{e_{uv}} \) is the hidden embedding of edge \( (u, v) \) at the \((l - 1)\)-th layer.

In contrast to traditional neural networks, where the data samples are typically independent and identically distributed (i.i.d.), GNN introduces data dependencies due to the underlying graph structure. These dependencies arise from the fact that the hidden embedding of each vertex in a GNN is updated based on the embeddings of its neighbors at the previous layer, which is formalized by Equation (1) in global view and Equation (2) in local view, respectively. This dependency enables GNNs to capture the structural information in the graph. By leveraging the graph
structure and capturing the dependencies between vertices and edges, GNNs are able to model complex relational data and perform tasks such as node classification and link prediction.

**GNN Training Methods.** To train a GNN model, a basic method is to update the model parameters once in a single epoch, which is called *full-graph GNN training*. However, full-graph GNN training is memory intensive [128] since it requires accessing the whole training graph and cannot scale to large graphs. Alternatively, we may choose to update the GNN model parameters multiple times over the course of a single epoch. This is known as *mini-batch GNN training*. An epoch is divided into multiple iterations and each iteration updates the GNN model with a batch (i.e., a subset of the whole training dataset).

Compared with traditional **deep neural network (DNN)** training, GNN training has two essential differences, which are illustrated by an example in Figure 1. First, for the batch generation in mini-batch training, the GNN constructs a batch with $L$-hop neighbors involved, whereas a traditional DNN involves the selected training samples only. Because of the data dependency in GNN training, to compute the output embedding in the final layer (layer $L$) of a training vertex, we need all the embeddings of its neighbors from the previous layer (layer $L - 1$). Recursively, in order to compute the embeddings of neighboring vertices at layer $L - 1$, embeddings of its 2-hop neighbors are required. Therefore, an exact mini-batch containing only one training vertex involves all its $L$-hop neighbors, along with their features, as the input of mini-batch training. A significant challenge arises when constructing such mini-batches: they can quickly become large and expand to include a substantial portion of the complete graph, even if only a few vertices are selected as the central training vertices. Some methods are proposed to mitigate this problem by sampling in the neighborhood to reduce the total size of the mini-batch, such as node-wise sampling [22, 52], layer-wise sampling [20, 62, 174], and subgraph sampling [27, 147].

Second, for the model computation in both mini-batch training and full-graph training, an additional aggregation process is introduced, to collect vertex information from the neighborhood. Certain GNN architectures [28, 112] may also require an extra message-passing phase prior to the aggregation stage, to generate edge-corresponded embeddings to be aggregated. After the output embeddings of training vertices are computed, a backward propagation is performed to compute
the gradient for each parameter. Similar to the forward computation, the gradients on each vertex are sent alongside the edges to their neighbors during backward propagation, incurring an additional scatter process compared with DNN training. The aggregation and scatter operation among data samples leads to a much more complex computation process compared with DNN training.

**Distributed DNN Training.** Distributed DNN training [71, 73] is a solution to large-scale DNNs.Parallelism and synchronization are two key components. (1) **Parallelism.** Data parallelism is a prevalent training paradigm in distributed DNN training. In data parallelism, the model is replicated across multiple devices or machines, with each replica processing a different subset of the training data. Computed gradients are exchanged and averaged to synchronize the model parameters, often utilizing communication operations such as all-reduce [21]. To handle large models that exceed the capacity of a single GPU, model parallelism is adopted, in which each device processes a distinct part of the model during forward and backward propagation. (2) **Synchronization.** Distributed training can be categorized into synchronous and asynchronous training. In synchronous training, all workers complete a forward and backward pass before model updates occur. This ensures that all workers are using the same model parameters for computation. In asynchronous training, workers update the model parameters independently to remove the global synchronization point between each mini-batch. Based on the idea of asynchronous training, pipeline parallelism is proposed [63, 92] to perform flexible mini-batch training and improve resource utilization to accelerate the overall training time.

Because of the large models in the DNNs, most efforts are made to manage the storage and synchronization of model parameters for the distributed DNN training. Since GNN models typically have shallow network structures, the management of model parameters is trivial. However, the input graphs of GNNs can be extremely large. The specific data dependency introduced by graph data structure significantly impacts the computation process of distributed GNN training, resulting in a substantial communication overhead that differs from DNN training, which becomes the main consideration and introduces new challenges (see Section 3.2).

### 3 DISTRIBUTED GNN TRAINING AND CHALLENGES

#### 3.1 General Distributed GNN Training Pipeline

To better understand the general workflow of end-to-end distributed GNN training, we divide the training pipeline into three stages: data partition, batch generation, and GNN model training. The GNN model training stage is further divided into model execution (including the design of the execution model and communication protocol) and parameter update. Figure 2 visualizes the high-level abstraction of end-to-end distributed GNN training workflow.

**Data partition.** This is a preprocessing stage enabling distributed training. It distributes the input data (i.e., graph and feature) among a set of workers. Considering that the training data in GNNs are dependent, the data partition stage becomes more complex than the one in traditional distributed machine learning. As shown in Figure 2, the cross-worker edges among the partitioned data (i.e., subgraphs) imply the data dependency. If we acknowledge the data dependency among partitions, the distributed training efficiency is reduced by communication; if we simply ignore the data dependency, the model accuracy is destroyed. Therefore, data partition is a critical stage for the efficiency of end-to-end distributed GNN training.

**GNN batch generation.** This stage is only involved in distributed mini-batch training. In this stage, each worker generates a computation graph from the partitioned input graph and feature. However, due to the data dependency, the computation graph generation is quite different from the traditional deep learning model. The computation graph for the mini-batch training strategy may not be generated correctly without accessing remote input data. The remote data accessing makes the distributed batch generation more complicated than the traditional deep learning model.
Distributed Graph Neural Network Training

Fig. 2. The abstraction of distributed GNN training pipeline. Key stages are noted with the number of the section in which they will be further discussed. Stages marked with star ★ are optional for different training methods: batch generation is only involved in mini-batch training, and the communication protocol that is used to communicate hidden embeddings and gradients is involved in full-graph training.

GNN model training. This is the core stage for the embedding computing and model updating. It is further divided into model execution stage and parameter update stage. The model execution stage is specifically designed for distributed GNN training, whereas the parameter update stage is consistent with that of traditional DNN model training. We further divide the model execution stage into execution model and communication protocol. For distributed mini-batch training, only the execution model is involved, which manages the scheduling of different training operators and how to overlap them efficiently with the batch generation stage. For distributed full-graph training, both execution model and communication protocol are included. The execution model involves $k$-layer graph aggregation of GNN models and the aggregation exhibits an irregular data access pattern. Furthermore, the graph aggregation in each layer needs to access the hidden embeddings and computed gradients of the remote neighbors via the communication protocol, and the synchronization schema between layers should also be considered. Finally, for the parameter update stage, the existing techniques in classical distributed machine learning can be directly applied to the distributed GNN training. In conclusion, the distributed GNN model training stage is more complicated than traditional DNN training and needs careful design for both the execution model and the communication protocol.

3.2 Challenges in Distributed GNN Training

Due to the data dependency, it is non-trivial to train distributed GNNs efficiently. We summarize three major challenges here.

Challenge #1: Massive feature communication. For the distributed GNN training, massive feature communication is incurred in both the batch generation and GNN model training stage. When the GNN model is trained in a mini-batch manner, the batch generation needs to access the remote graph and features as in operation ① in Figure 2 to create a batch for local training. Even if in multi-GPU training with a centralized graph store (e.g., CPU memory), the computation graph generation incurs massive data movement from the graph store to workers (e.g., GPU memory). Existing empirical studies demonstrate that the cost of mini-batch construction becomes the bottleneck during the end-to-end training [64, 162]. When the GNN model is trained in a full-graph manner, the computation graph can be the same as the partitioned graph without communication. However, the graph aggregation in each layer needs to access the hidden features in the remote neighbors of vertices (operation ② in Figure 2), which causes massive hidden...
feature (or embedding) communication. In summary, both distributed mini-batch training and distributed full-graph training of GNN models suffer from massive feature communication.

**Challenge #2: The loss of model accuracy.** Mini-batch GNN training is more scalable than full-graph training, thus, it is adopted by most existing distributed GNN training systems. However, with the increase of model depth, the exact mini-batch training suffers from the neighbor-explosion problem. A de-facto solution is to construct an approximated mini-batch by sampling or ignoring cross-worker edges. Although the approximated mini-batches improve the training efficiency, it cannot guarantee the model convergence in theory. Therefore, for the distributed mini-batch GNN training, we need to make a trade-off between model accuracy and training efficiency. In addition, there is a trend to develop full-graph distributed GNN training, which has a convergence guarantee and is able to achieve the same model accuracy compared with the local training on a single worker.

**Challenge #3: Workload imbalance.** Workload balance is an intrinsic problem in distributed computing. However, the various workload characteristics of GNN models increase the difficulty of partitioning the training workload balance among workers. Because it is hard to model the GNN workload in a simple and unified way. Without formal cost models, the classical graph partitioning algorithms cannot be used to balance the GNN workload among workers. In addition, distributed mini-batch GNN training requires that each worker handle the same number of mini-batches with the same batch size (i.e., subgraph size), not simply balancing the number of vertices in subgraphs. In summary, it is easy to encounter workload imbalance when training GNNs in a distributed environment, thus, incurring workers waiting for each other and destroying the training efficiency.

### 3.3 Taxonomy of Distributed GNN Training Techniques

To realize the distributed GNN training and optimize the efficiency via solving the above challenges, many new techniques have been proposed in the past years. Figure 3 illustrates the relationship between challenges and the proposed techniques. Previous works typically present their own contributions as a component of their proposed systems or frameworks and may lack a comprehensive discussion of the techniques used in the same stage of distributed GNN training or the techniques addressing the same challenge. In this survey, we introduce a new taxonomy by organizing the techniques specifically designed for distributed GNN training on the basis of the stages in the end-to-end distributed training pipeline. With such a design, we organize similar techniques that optimize the same stage in the distributed GNN training pipeline together and help readers fully understand the existing solutions for the different stages in distributed GNN training.

![Fig. 3. Challenges and techniques of distributed GNN training.](image-url)
According to the previous empirical studies, due to the data dependency, the bottleneck of distributed GNN training generally comes up in the stages of data partition, batch generation and GNN model training, as shown in the pipeline. Furthermore, various training strategies (e.g., mini-batch training, full-graph training) bring in different workload patterns and result in different optimization techniques used in batch generation stage and model training stage. For example, the computation graph generation in batch generation stage is important to mini-batch training while communication protocol is important to full-graph training. In consequence, our new taxonomy classifies the techniques specifically designed for distributed GNN training into four categories (i.e., GNN data partition, GNN batch generation, GNN execution model and GNN communication protocol) as shown in Figure 4. In the following, we introduce the overview of each category.

**GNN data partition.** In this category, we review the data partition techniques for distributed GNN training. The goal of data partition is to balance the workload and minimize the communication cost for a GNN workload. GNN training is an instance of distributed graph computing.
Many traditional graph partition methods can be directly used. However, they are not optimal for distributed GNN training because of the new characteristics of GNN workloads. Researchers have put much effort into designing GNN-friendly cost models that guide traditional graph partition methods. In addition, the graph and features are two typical types of data in GNN, both of which are partitioned. Some works decouple the features from the graph structure and partition them independently. In Section 4, we elaborate on the existing GNN data partition techniques.

**GNN batch generation.** In this category, we review the techniques of GNN batch generation for mini-batch distributed GNN training. The methods of mini-batch generation affect both the training efficiency and model accuracy. Graph sampling is a popular approach to generating a mini-batch for large-scale GNN training. However, the standard graph sampling techniques do not consider the factors of distributed environments and each sampler on a worker will frequently access data from other workers, incurring massive communication. Recently, several new GNN batch generation methods optimized for distributed settings have been introduced. We further classify them into distributed sampling mini-batch generation and partition-based mini-batch generation. In addition, the cache has been extensively studied to reduce communication during GNN batch generation. In Section 5, we elaborate on the existing GNN batch generation techniques.

**GNN execution model.** In this category, we review the execution model for both mini-batch and full-graph GNN training. In mini-batch GNN training, sampling and feature extraction are two main operations that dominate the total training time. To improve efficiency, different execution models are proposed to schedule the training stage of a mini-batch with the sampling and feature extraction stage in order to fully utilize computing resources. In full-graph GNN training, the neighbors’ state for each vertex is aggregated in the forward computation and the gradients are scattered back to the neighbors in the backward computation, leading to massive communication of remote vertices. Due to the data dependency and irregular computation pattern, traditional machine learning parallel models (e.g., data parallel, model parallel) are not optimal for graph aggregation, especially when feature vectors are high dimensional. We introduce the full-graph execution model from both matrix view and graph view. From the matrix view, we classify the execution of distributed SpMM into computation-only, communication-computation, and communication-computation-reduction models. From the graph view, we classify the execution models from two perspectives. From the perspective of the execution orders of graph operators, we classify the full-graph execution models into one-shot execution models and chunk-based execution models. From the perspective of the synchronization timeliness in graph aggregation, we classify the full-graph execution models into synchronous execution models and asynchronous execution models. Based on the removal of different synchronization points, the asynchronous execution model can be further categorized into two types. A detailed description of these techniques is presented in Section 6.

**GNN communication protocol.** In this category, we review the communication protocols of distributed full-graph parallel training. The graph aggregation of distributed full-graph training needs to access remote hidden embedding because the neighbors of a vertex cannot always be local with graph partitioning. Based on the synchronous and asynchronous execution models in graph aggregation, these communication protocols can also be categorized as synchronous communication protocols and asynchronous communication protocols, respectively. The former assumes that in each layer, the model accesses the latest embedding while the latter assumes that in each layer the model is allowed to access stale embedding (or historical embeddings). A detailed description of various communication protocols is presented in Section 7.

## 4 GNN DATA PARTITION

In this section, we review existing techniques of GNN data partition in distributed GNN training. Figure 5 describes the overview of the techniques. Considering the fact that graphs and features
are two typical types of data in GNN, we classify partition methods into graph partition and feature partition. The optimization objectives are workload balance and communication and computation minimization, which aim at addressing challenges #1 and #3. In addition, the cost model is another critical component that captures the characteristics of GNN workloads. With a well-designed cost model, workloads on each partition can be accurately estimated. A graph partition strategy can better address the challenges in GNN by adopting a more accurate cost model. In the following, we first present various cost models, which are the basis of graph partition. Then we discuss the graph partition and feature partition.

4.1 Cost Model of GNN
The cost model estimates the computation and communication cost of a GNN workload. Therefore, building a cost model is an important technique that underpins the data partition. In general, for a graph analysis task, we use the number of vertices to estimate the computation cost and the number of cross-edges to estimate the communication cost [70]. However, this simple method is not proper for the GNN tasks because they are not only influenced by the number of vertices and cross-edges but also influenced by the dimension of features, the number of layers, and the distribution of training vertices. Researchers have proposed several GNN-specific cost models, including the heuristics model, learning-based model, and operator-based model.

The **heuristics model** selects several graph metrics to estimate the cost via simple user-defined functions. Several heuristics models for GNN workloads have been introduced in the context of streaming graph partitions [104, 139], which assign a vertex or block to a partition one by one. Such models define an affinity score for each vertex or block and the score helps the vertex or block select a proper partition.

Assume that a GNN task is denoted by $GNN(L, G, V_{train}, V_{valid}, V_{test})$, where $L$ is the number of layers, $G$ is the graph, and $V_{train}, V_{valid},$ and $V_{test}$ are the vertex set for train, validation, and test, respectively. The graph $G$ is partitioned into $K$ subgraphs. In the context of the streaming graph partition, for each assignment of vertex or block, let $P_i (1 \leq i \leq K)$ be the set of vertices that have been assigned to it already, and $V_{train}^i, V_{valid}^i,$ and $V_{test}^i$ are the corresponding vertex set belong to partition $P_i$.

Lin et al. [80] define an affinity score vector with $K$ dimension for each train vertex $v_t \in V_{train}$, in which each score represents the affinity of the vertex to a partition. Let $IN(v_t)$ be the $L$-hop in-neighbor set of train vertex $v_t$. The score of $v_t$ with respect to partition $P_j$ is defined as
Table 1. The Vertex Features Used in ROC’s Cost Model

| Name  | Type                | Definition | Description |
|-------|---------------------|------------|-------------|
| $x_1$ | graph-structural feature | 1          | the vertex itself |
| $x_2$ | graph-structural feature | $|N(v)|$ | the number of neighbors |
| $x_3$ | graph-structural feature | $|C(v)|$ | continuity of neighbors |
| $x_4$ | runtime feature | $\sum_i \left\lceil \frac{\eta_i}{WS} \right\rceil$ | mem accesses to load neighbors |
| $x_5$ | runtime feature | $\sum_i \left\lceil \frac{\xi_i}{WS} \right\rceil$ | mem accesses to load the activations of all neighbors |

follows:

$$score_{li}^i = |V_{train} \cap IN(v_l)| \cdot \frac{V_{avg}^i - |V_{train}^i|}{|P_l|},$$  

(3)

where $V_{avg}^i = \frac{V_{train}^i}{K}$. This score function implicitly balances the number of training vertices among partitions. Liu et al. [82] define a similar affinity score with respect to a block (or subgraph) $B$. The formal definition is

$$|P_l \cap IN(B)| \cdot \left(1 - \frac{|P_l|}{P_{avg}}\right) \cdot \left(1 - \frac{|V_{train}^i|}{V_{train}^i}\right),$$  

(4)

where $P_{avg} = \frac{|V|}{K}$. Zheng et al. [162] define the affinity score of a block $B$ by considering all the training, validation, and test vertices. The formula is

$$\frac{\text{CrossEdge}(P_l, B)}{|P_l|} \cdot \left(1 - \alpha \frac{V_{train}^i}{V_{train}^i} - \beta \frac{V_{valid}^i}{V_{valid}^i} - \gamma \frac{V_{test}^i}{V_{test}^i}\right),$$  

(5)

where $\text{CrossEdge}(P_l, B)$ is the number of cross-edges between $B$ and $P_l$, and $\alpha, \beta, \gamma$ are hyper-parameters manually set by users.

The learning-based model takes advantage of machine learning techniques to model the complex cost of GNN workloads. The basic idea is to manually extract features via feature engineering and apply classical machine learning to train the cost model. The learning-based model is able to estimate the cost using not only the static graph structural information but also runtime statistics of GNN workloads, thus achieving a more accurate estimation than the heuristic models. Jia et al. [65] introduce a linear regression model for GNN computation cost estimation. The model estimates the computation cost of a single GNN layer $l$ with regard to any input graph $G = (V, E)$. For each vertex in the graph, they select five features (listed in Table 1), including three graph-structural features and two runtime features. The estimation model is formalized as follows:

$$t(l, v) = \sum_i w_i(l)x_i(v)$$  

(6)

$$t(l, G) = \sum_{v \in V} t(l, v) = \sum_{v \in V} \sum_i w_i x_i(v) = \sum_i w_i \sum_{v \in V} x_i(v) = \sum_i w_i x_i(G),$$  

(7)

where $w_i(l)$ is a trainable parameter for layer $l$, $x_i(v)$ is the $i$-th feature of $v$, and $x_i(G)$ sums up the $i$-th feature of all vertices in $G$.

Wang et al. [121] use a polynomial function $f$ to estimate the computation cost of a vertex over a set of manually selected features. The formal definition is

$$f = \sum_{i=0}^{T} n_i m_i,$$  

(8)
where $T$ is the number of neighbor types (e.g., a metapath [121]), which is defined by the GNN models, $n_i$ is the number of neighbors for the $i$-th type, $m_i$ is the total feature dimensions of the $i$-th type of neighbor instance (i.e., a neighbor instance of the $i$-th type has $n$ vertices and each vertex has feature dimension $f_i$, then $m_i = n \times f_i$). Refer to the original work [121] for the detailed examples of the function $f$. The total computation cost of a subgraph is the sum of the estimated costs of the vertices in the subgraph.

The operator-based model enumerates the operators in a GNN workload and estimates the total computation cost by summing the cost of each operator. Zhao et al. [158] divide the computation of a GNN workload into forward computation and backward computation. The forward computation of a GNN layer is divided into aggregation, linear transformation, and activation function. The backward computation of a GNN layer is divided into gradient computation towards loss function, embedding gradient computation, and gradient multiplications. As a result, the costs of computing embedding $h_l^l(v)$ of vertex $v$ in layer $l$ in forward and backward propagation are estimated by $c_f(v, l)$ and $c_b(v, l)$, respectively:

$$c_f(v, l) = \alpha |N_v| d_{l-1} + \beta d_l d_{l-1} + \gamma d_l,$$  \hspace{1cm} (9)

$$c_b(v, l) = \begin{cases} (\lambda + \eta) d_l + (2\beta + \eta) d_l d_{l-1}, & l = L \\ \alpha |N_v| d_l + (\beta + \eta) d_l d_{l-1} + \eta d_l, & 0 < l < L \end{cases},$$ \hspace{1cm} (10)

where $d_l$ is the dimension of hidden embeddings in the $l$-th GNN layer, $|N_v|$ is the number of neighbors of vertex $v$, $\alpha$, $\beta$, $\gamma$, and $\eta$ are constant factors that can be learned through testing the runtime in practice. Finally, the computation cost of a mini-batch $B$ is computed by summing up the computation cost from all vertices in the $B$ from layer $1$ to layer $L$ as follows:

$$C(B) = \sum_{l=0}^{L-1} \sum_{v \in \bigcup_{u \in B} N_u^l} (c_f(v, l) + c_b(v, l)),$$ \hspace{1cm} (11)

where $N_u^l$ represents the vertices in graph $G$ that are $l$-hops away from vertex $u$.

### 4.2 Graph Partition in GNNs

The GNN is a type of graph computation task. Classical graph partition methods can be directly applied to GNN workloads. Many distributed GNN training algorithms and systems such as AliGraph [173] and DistGNN [89] adopt METIS [70], Vertex-cut [49], Edge-cut [70], streaming graph partition [104], and other graph partition methods [11] to train the GNN models in different applications. Recently, Tripathy et al. [110] empirically studied the performance of 1-D, 2-D, 1.5-D, and 3-D graph partitioning methods with respect to GCN models. However, due to the unique characteristics of GNN workloads, classical graph partition methods are not optimal to balance the GNN workload while minimizing the communication cost.

In distributed mini-batch GNN training, to balance the workload, we need to balance the number of train vertices (i.e., the number of batches per epoch) among workers. As a result, new optimization objectives are introduced. Zheng et al. [164] formulate the graph partition problem in the DistDGL as a multi-constraint partition problem that aims at balancing training/validation/test vertices/edges in each partition. They adopt the multi-constraint mechanism in METIS to realize the customized objective of graph partition for both homogeneous graphs and heterogeneous graphs [165], and further expand this mechanism to a two-level partitioning strategy that handles the situation with multi-GPUs and multi-machines and balances the computation. Lin et al. [80] apply streaming graph partition – **Linear Deterministic Greedy (LGD)** [104] and use a new...
affinity score (Equation (3)) to balance the GNN workload. Furthermore, they ensure that each vertex in a partition has its complete $L$-hop neighbors to avoid massive communication during sampling. Recently, block-based streaming graph partition methods [82, 162] have been introduced for mini-batch distributed GNN training. They first use multi-source BFS to divide the graph into many small blocks, then apply greedy assignment heuristics with customized affinity scores (Equations (4) and (5)) to partition the coarsened graph and uncoarsen the block-based graph partition by mapping back the blocks to the vertices in the original graph. Besides workload balance, some works minimize the total computation cost. Zhao et al. [158] prove that the computation cost function (Equation (11)) is a submodular function and use METIS to partition the graph into cohesive mini-batches achieving $2 - \frac{2}{M}$ approximation of the optimal computation cost.

In distributed full-graph GNN training, we need to balance the workload of each GNN layer among a set of workers while minimizing the embedding communication among workers. One approach is to leverage a heuristic method to model the computation workload and the communication overhead, and adopt the multi-constraint graph partitioning mechanism. Wan et al. [118] models the computation workload as both the number of vertices and the number of edges, and treat them as two constraints. The communication overhead is modeled as the number of remote vertices in each worker. Therefore, with multi-constraint graph partitioning, the computation workload can be balanced and the communication overhead can be reduced. To further balance the communication overhead, an iterative re-partitioning stage is adopted to swap vertices between workers with the highest and lowest communication volumes. Demirci et al. [29] model the computation workload as the total number of vertices’ neighbors in each worker. To model the communication overhead under point-to-point (P2P)–based communication protocol (introduced in Section 7.1), they introduce the hypergraph to model the communication cost of each vertex as the connectivity of its corresponding net in the hypergraph. Based on the hypergraph model, existing methods [16] for hypergraph partitioning are adopted to distribute vertices on different workers. Another approach is to apply learning-based cost models to model the complex cost for diverse GNN models. On top of the model-aware cost models, existing graph partition methods are adopted. For example, Jia et al. [65] utilize the linear regression–based cost model (Equation (7)) and apply range graph partitioning to balance the workload estimated by the cost model. In addition, range graph partition guarantees that each partition holds consecutively numbered vertices, thus reducing the data movement cost between CPU and GPU. Wang et al. [121] use Equation (8) to estimate the computation cost of a partition and apply the application-driven graph partition method [36] to generate a workload balancing plan, which adaptively mitigates the workload and minimizes the communication cost. Based on the vertex-cut partition, Hoang et al. [54] leverage the two-dimensional (2D) Cartesian vertex cut to improve scalability.

4.3 Feature Partition in GNN

Being an important type of data in GNN, features need to be partitioned as well. Most distributed GNN training solutions treat features as the vertex properties of a graph and they are partitioned along with the graph partition. For example, if the graph is partitioned by the edge-cut method, then each vertex feature is stored in the partition where the corresponding vertex lies. In other words, the feature matrix is partitioned in a row-wise fashion.

Considering the different processing patterns of features compared with the graph structure, a line of works partition a feature matrix independent of the graph. When a graph is partitioned by the 2D partition method, Tripathy et al. [110] partition the feature matrix with the 2D partition method as well, while Md et al. [89] use a row-wise method to partition the feature matrix with replication and ensure that each vertex has the complete feature locally. Gandhi and Iyer [46] deeply analyze the communication pattern in distributed mini-batch GNN training and find
that classical graph partition methods only reduce the communication cost at the first hop process. Furthermore, sophisticated graph partitioning, e.g., METIS, is an expensive preprocesstep. They introduce partitioning the graph and feature separately. The graph is partitioned randomly to avoid the expensive preprocess. The input features are partitioned along the feature dimension (i.e., column-wise partition). In other words, each partition contains a sub-column of all vertices’ features. With this data partition strategy, they further design a new execution model, introduced in Section 6.2.5, and train the GNN model with minimal communication, as described in Section 7.2.2, especially when the input feature has high dimension and the hidden features have low dimension. Wolf et al. [129] also use a column-wise feature partition method of distributed training for ultra-wide GCN models. In contrast, in this method, the graph is partitioned with METIS and both input and intermediate features are partitioned along the feature dimension.

5 GNN BATCH GENERATION

Mini-batch GNN training is a common approach to scaling GNNs to large graphs. Graph sampling is a de-facto tool to generate mini-batches in standalone mode. So far, many sampling-based GNNs [20, 27, 52, 62, 147, 174] have been proposed, which can be classified into vertex-wise sampling, layer-wise sampling and subgraph-wise sampling. Different batch generation methods influence both the training efficiency and training accuracy. To avoid graph sampling becoming a bottleneck, there are some explorations to improve the performance and scalability of it [6, 32, 76, 144].

In mini-batch distributed GNN training, the data dependence brings in massive communication for the batch generation process. To improve the training efficiency in distributed settings, several new GNN batch generation techniques that are specific to distributed training have been proposed and address challenge #1 and challenge #2. As shown in Figure 6, one solution is generating a mini-batch via distributed sampling and the other is directly using the local partition (or subgraph) as the mini-batch. In the following, we describe these techniques.

5.1 Mini-batch Generation with Distributed Sampling

Based on existing sampling-based GNNs, it is straightforward to obtain the distributed version by implementing distributed sampling. In other words, we use distributed graph sampling to create mini-batches over large graphs. Most of existing distributed GNN systems, such as AliGraph [173], DistDGL [164], and BGL [82], follow this idea. However, in a distributed environment, a basic sampler on a worker will frequently access data from other workers, incurring massive communication. In multi-GPU settings, although graphs and features might be centralized and stored in CPU memory, each GPU needs to access the mini-batch from CPU memory via massive CPU–GPU data movement.

To reduce communication in distributed computing, caching is a common optimization mechanism for improving data locality. At the system level, many GNN-oriented cache strategies have
been proposed. The basic idea of caching is to store frequently accessed remote vertices locally. Zheng et al. [164] propose to replicate the remote neighbors of boundary vertices locally for each partitioned graph in DistDGL and ensure that each vertex locally has all neighbors so that the mini-batch generator (e.g., sampler) can compute locally without communicating to other partitions. However, such a method only reduces the communication caused by the direct (i.e., one-hop) neighbor access and is unaware of the access frequency. Zhu et al. [173] introduce a metric called $l$-th importance of $v$, denoted by $\text{Imp}^{(l)}(v)$. The metric is formally defined as $\text{Imp}^{(l)}(v) = \frac{D^{(l)}_{in}(v)}{D^{(l)}_{out}(v)}$, where $D^{(l)}_{in}(v)$ and $D^{(l)}_{out}(v)$ are the number of $l$-hop in- and out-neighbors of the vertex $v$. To balance the communication reduction and storage overhead, they only cache the vertices when its $\text{Imp}^{(l)}(v)$ is larger than a threshold. Lin et al. [80] design a static GPU cache that stores the high out-degree vertices for the multi-GPU training. The static cache can reduce the feature movement between CPU and GPU. Min et al. [90] introduce weighted reverse PageRank (Weighted R-PageRank) [7], which incorporates the labeling status of the vertices into the reverse PageRank method to identify the frequent access vertices. The hot vertex features with a high score are cached in GPU memory and the rest of the cold vertex features are placed in CPU memory. Due to the different bandwidths between NVLink and PCIe, they divided the hot vertex features into the hottest one and the next hottest one. The hottest vertex features are replicated over multiple GPUs. The next hottest data is scattered over multiple GPUs that can be accessed from the peer GPU memory. While the above strategies only focus on the feature cache, the latest studies [105, 154] find that when GPU sampling is used, caching both the vertex features and graph topology achieves better performance. To find the optimal cache allocation for graph topology and vertex features, Sun et al. [105] leverage a linear search strategy to obtain the ratio of the two cache sizes and allocate each cache with the top hottest data. In contrast, Zhang et al. [154] model this allocation problem as a knapsack problem and design a greedy algorithm to find the optimal cache allocation strategy.

The above static cache strategies cannot be adaptive to various graph sampling workloads. Liu et al. [82] propose a dynamic cache mechanism that applies the first-in-first-out (FIFO) policy by leveraging the distribution of vertex features. To improve the cache hit ratio, they introduce proximity-aware ordering for a mini-batch generation. The order constrains the training vertex sequence in breadth-first search (BFS) order. To ensure model convergence, they introduce randomness to the BFS-based sequence by selecting a BFS sequence in a round-robin fashion and randomly shifting the BFS sequence. Yang et al. [142] propose a more robust cache approach, called a pre-sampling-based caching policy. Given a graph $G$, a sampling algorithm $A$, and a training set $T$, it first executes $K$ sampling stages to collect statistics, then uses the statistics to identify the frequency (or hotness) of the data, and finally caches the data with larger hotness. Kayer et al. [69] adopt the same idea and propose an analysis-based cache policy that performs more accurate predictions on the data hotness. A propagation model is designed in the analysis method, which iteratively propagates and calculates the sampled probability of each vertex given $G$, $A$, and $T$. Both the pre-sampling-based and analysis-based caching policies consider all three factors that influence the data access frequency, which are more robust than previous cache strategies for the various GNN workloads.

Besides the cache strategy, another approach is to design new distributed sampling techniques that are communication-efficient and maintain the model accuracy. A basic idea of communication-efficient sampling is to prioritize a sampling local vertex. However, this introduces bias for the generated mini-batch. Inspired by the linear weighted sampling methods [20, 174], Jiang et al. [66] propose a skewed linear weighted sampling for the neighbor selection. Concretely, the skewed sampling scales the sampling weights of local vertices by a factor $s > 1$ and Jiang et al. theoretically
prove that the training can achieve the same convergence rate as the linear weighted sampling by properly selecting the value of $s$. Cai et al. propose a technique called collective sampling primitive (CSP) \cite{15}. CSP reduces communication costs by pushing the sampling tasks to remote workers instead of pulling the entire neighbor lists to the local worker. This is beneficial because the full neighbor list of a center vertex is often much larger than the sampled results. With CSP, sampling tasks for remote vertices are conducted on remote workers and only the results are sent back, reducing communication overhead.

In addition to the above GNN-specific distributed sampling techniques, other general distributed graph sampling methods on CPU clusters or GPU clusters can also provide help, such as C-SAW \cite{93}, Knightking \cite{143} and Skywalker \cite{124}.

### 5.2 Partition-Based Mini-batch Generation

According to the abstraction of distributed GNN training pipeline in Figure 2, we know that the graph is partitioned at first. If we restrict each worker to only training the GNN model with the local partition, massive communication can be avoided. In such a training strategy, a partition is a mini-batch, which we call a partition-based mini-batch. PSGD-PA \cite{96} is a straightforward implementation of the above idea with a Parameter Server. In GraphTheta \cite{75}, the partitions are obtained via a community detection algorithm.

However, the ignorance of cross edges among partitions causes a loss of model accuracy. To improve the model accuracy, subgraph (i.e., partition) expansion is introduced. It preserves the local structural information of boundary vertices by replicating remote vertices locally. Xue et al. \cite{140} use METIS to obtain a set of unoverlapped subgraphs and then expand each subgraph by adding the one-hop neighbors of vertices that do not belong to the subgraph. Similarly, Angerd et al. \cite{4} use the breadth-first method to expand the $l$-hop subgraph. Zhao et al. \cite{160} define a Monte Carlo-based vertex importance measurement and use a depth-first sampling to expand the subgraph. They further introduce variance-based subgraph importance and weighted global consensus to reduce the impact of subgraph variance and improve the model accuracy. In addition, Ramezani et al. \cite{96} introduced LLCG (Learn Locally, Correct Globally) to improve the model accuracy for the partition-based mini-batch training. LLCG follows the Parameter Server architecture and periodically executes a global correction operation on a parameter server to update the global model. The empirical results demonstrate that the global correction operation can improve the model accuracy significantly.

## 6 GNN EXECUTION MODEL

The execution model is the core stage for GNN model training, as shown in Figure 2. It is responsible for scheduling different operations to achieve high training efficiency. For distributed mini-batch training, as the sampling and feature extraction operations in batch generation dominate the training efficiency and make the computation graph generation expensive, the execution model focuses on how to efficiently schedule these operations with the batch training operation. For distributed full-graph training, the computation graph execution is complicated due to the data dependency among workers, and the execution model focuses on the scheduling of graph aggregation and communication during computation graph execution.

In the following, due to different execution methods of mini-batch and full-graph GNN training, we discuss the execution model of them separately.

### 6.1 Mini-Batch Execution Model

In distributed mini-batch GNN execution, sampling and feature extraction are two unique operations, which become the bottleneck. As shown in \cite{46, 82}, the sampling and feature extraction
takes up almost 83% to 99% of the overall training time. The low efficiency is caused by massive communication and resource contention. Therefore, several execution models for mini-batch execution are proposed.

A conventional execution model performs the sampling and feature extraction of each computation graph (i.e., batch) in sequence before batch training is performed, as shown in Figure 7(a). When CPU-based graph sampling is applied, the sampling is executed on a CPU to construct the graph structure; feature extraction with respect to the graph is executed on a GPU with cache support, and training is executed on the GPU as well. When GPU-based sampling [64, 93] is applied, all the operations in a pipeline are executed on a single device. This conventional execution model is popular among existing GNN training systems [80, 164, 173]. The optimizations in the context of the conventional execution model lie in the optimization of the sampling operation and the cachestrategy for feature extraction, as introduced in Sections 4 and 5. However, the conventional execution model causes resource contention. For example, processing graph sampling requests and subgraph construction compete for CPUs when CPU-based graph sampling is applied; graph sampling and feature extraction compete for GPUs’ memory when GPU-based graph sampling is used.

A factored execution model uses a dedicated device or isolated resources to execute a single operation, as shown in Figure 7(b). Empirical studies demonstrate that during the mini-batch GNN training, an operation shares a large amount of data in different epochs and preserves inter-epoch data locality. This observation motivates the factored execution model to avoid resource contention and improves the data locality. Yang et al. [142] introduce a factored execution model that eliminates resource contention on GPU memory. The new model assigns a sampling operator and training operator to individual devices. More specifically, it uses some dedicated GPUs for graph sampling, and the remaining GPUs are used for feature extraction and model training. With the help of the factored design, the new execution model is able to sample larger graphs while GPUs for feature extraction can cache more features. To improve the CPU utilization when CPU-based graph sampling is used, Liu et al. [82] assign isolated resources (i.e., CPUs) to different operations and balance the execution time of each operation by adjusting resource allocation. They introduced a profiling-based resource allocation method to improve CPU utilization and processing efficiency. The new resource allocation method first profiles the cost of each operator and solves an optimization problem, minimizing the maximal completion time of all operations with resource constraints.
An operator-parallel execution model enables inter-batch parallelism and generates multi-computation graphs in parallel by mixing the execution of all operators in the pipeline, as shown in Figure 7(c). Zheng et al. [165] decompose the batch generation stage into multiple stages (e.g., graph sampling, neighborhood subgraph construction, and feature extraction) and each stage is treated as an operator. Dependencies exist among different operators, while there are no dependencies among mini-batches. Therefore, the operators in different mini-batches can be scheduled into a pipeline, and operators without dependencies can be executed in parallel. In this way, the batch generation of a later batch can be performed immediately after or concurrently with that of the former batch instead of waiting for the former batch to finish the model training stage. The idea introduced by Cai et al. [15] is very similar, as they set a sampler and a loader to conduct the sampling and feature extraction operation, respectively. A queue is used to manage the execution of different mini-batches. With a followed trainer to perform the execution of computation graph, the training process of mini-batches forms a training pipeline. Several mini-batches can be executed simultaneously, thus improving the GPU utilization and accelerating both the computation graph generation and execution stage. Zheng et al. [162] also decouple the batch generation while further modeling the dependencies among different operators into a directed acyclic graph (DAG). Treating each operator as a node in the DAG, a two-level scheduling strategy is proposed by considering both the scheduling among DAGs (coarse-grained) and among operators (fine-grained). With such two-level scheduling, the cost of CPU-based sampling and GPU-based feature extraction can be balanced to improve the CPU utilization and reduce the end-to-end GNN training time.

Execution model with pull-push parallelism. All of the above execution models construct the subgraph with sampling first followed by extracting features according to the subgraph. When the input features are high-dimensional, the communication caused by the feature extraction becomes the bottleneck in the end-to-end GNN training pipeline. The expensive high-quality graph partition introduced in Section 4.2.1 is an option to reduce communication. Gandhi and Iyer [46] introduce a new execution model with pull-push parallelism. The new model partitions the graph using fast hash partition and the feature matrix using column-wise hash partition. It constructs the subgraph by pulling remote vertices and edges locally and then pushes the constructed subgraph to all the workers. Then each worker extracts partial features and performs aggregation of these partial features for all the received subgraphs in parallel, forming the model parallel stage shown in Figure 7(d). A data parallel stage that trains subgraphs (i.e., mini-batches) individually on each worker is performed next. Light communication overhead is incurred between the model parallel stage and data parallel stage to move the low-dimensional hidden embeddings. With the help of pull-push parallelism, the execution model replaces the expensive input feature movement with light graph structure movement, thus improving the efficiency of batch generation and model training.

6.2 Full-Graph Execution Model

For distributed mini-batch GNN training, it is trivial to parallelize the execution of computation graphs, i.e., mini-batches, among the distributed workers, since each mini-batch is completely stored in a single worker. For distributed full-graph or large-batch GNN training, the execution of a computation graph is partitioned among a set of workers. Due to the data dependency in the computation graph, it is non-trivial to achieve the high efficiency of the GNN training. So far, many computation graph execution models specific to GNN training have been proposed.

In the following section, we first discuss the full-graph execution model from the perspective of vertex computation. We delve into the execution model from both the view of graph operation (Section 6.2.1) and matrix multiplication (Section 6.2.2). From the view of graph operation, the GNN execution can be modeled as SAGA-NN [86], which is inspired by the traditional GAS model.
in graph processing. From the view of matrix multiplication, the core of GNN execution can be modeled as \textbf{SpMM (Sparse Matrix Multiplication)}. We also make a comparison in Section 6.2.3. Second, we focus on the update mode, which determines whether the vertex embeddings and model parameters used for computation are updated in time or with delay. We categorize the full-graph execution model into synchronous execution models (Section 6.2.4) and asynchronous execution models (Section 6.2.5). Note that the above two perspectives (i.e., vertex computation and update mode) intersect, and a system simultaneously adopts one execution model from each of the two perspectives.

6.2.1 Graph View. From the view of graph processing, we use the most well-known programming model \textbf{SAGA-NN} \cite{86} for the following discussion. SAGA-NN divides the forward computation of a single GNN layer into four operators \textit{Scatter (SC)}, \textit{ApplyEdge (AE)}, \textit{Gather (GA)}, and \textit{ApplyVertex (AV)}. SC and GA are two graph operations, in which vertex features are scattered along the edges and gathered to the target vertex, respectively. AE and AV may contain neural network (NN) operations, which process directly on the edge features or the aggregated features of the target vertices, respectively.

According to different computation paradigms of graph operators (i.e., SC and GA), we divide computation graph execution models into \textit{one-shot execution} and \textit{chunk-based execution}.

\textbf{One-Shot Execution.} One-shot execution is the most common model adopted by GNN training. On each worker, all the neighbor features (or embeddings) needed for one graph aggregation of local vertices is collected in advance to the local storage, and the graph aggregation is processed in one shot, which is illustrated in the upper part of Figure 8. The remote information may require communication with remote workers; the communication protocol will be discussed in Section 7. From the local view of graph aggregation, each target vertex is assigned to a specific worker. The worker gathers the neighbors’ features of the target vertex and aggregates all of these features at the same time. As an example, DistDGL \cite{164} adopts this model straightforwardly. BNS-GCN \cite{116} leverages a sampling-based method to aggregate neighbor vertices’ features at random. One-shot execution can also be easily incorporated with the asynchronous execution model \cite{117} (see Section 6.2.5), in which the stale embeddings are collected for graph aggregation.

\textbf{Chunk-Based Execution.} One-shot execution is straightforward, yet it causes expensive memory consumption and heavy network communication. For example, storing all the data required in the local storage of a worker may cause the \textbf{Out-of-Memory (OOM)} problem. On the other hand, transferring all the vertex features may lead to heavy network overhead. To address these problems, an alternative approach is to split the complete aggregation into several partial aggregations and
accumulate the partial aggregations to get the final graph aggregation. To achieve this, the neighborhoods of a vertex are grouped into several sub-neighborhoods. We call each sub-neighborhood along with the vertex features or embeddings a \textit{chunk}. The chunks can be processed sequentially (\textit{sequential chunk-based execution}) or in parallel (\textit{parallel chunk-based execution}), which is illustrated in the lower part of Figure 8.

Under \textit{sequential chunk-based execution}, partial aggregation is conducted sequentially and accumulated to the final aggregation result one after another. NeuGraph \cite{86} uses the 2D partitioning method to generate several edge chunks, thus the neighborhood of a vertex is partitioned accordingly into several sub-neighborhoods. It assigns each worker with the aggregation job of certain vertices and feeds their edge chunks sequentially to compute the final result. SAR \cite{91} uses the edge-cut partitioning method to create chunks, retrieves the chunks of vertex from remote workers sequentially, and computes the partial aggregations locally. Sequential chunk-based execution effectively addresses the OOM problem, since each worker only needs to handle the storage and computation of one chunk at a time.

Under \textit{parallel chunk-based execution}, the partial aggregations of different chunks are computed in parallel. After all chunks finish partial aggregation, communication is invoked to transfer the results, and the final aggregation result is computed at the same time. Since the communication volume incurred by transferring the partial aggregation result is much less than transferring the complete chunk, the network communication overhead can be reduced. DeepGalois \cite{54} straightforwardly adopts this execution model. DistGNN \cite{89} orchestrates the parallel chunk-based execution model with the asynchronous execution model to transfer the partial aggregation with staleness. FlexGraph \cite{121} further overlaps the communication of remote partial aggregations with the computation of the local partial aggregation to improve efficiency.

6.2.2 \textbf{Matrix View}. As described in Section 2, the matrix formulation of a GNN model is given by $H_l = \sigma(\tilde{A}H_{l-1}W_{l-1})$, which involves SpMM since matrix $A$ is sparse. In order to perform computations for the GNN model, three matrices (i.e., $A$, $H$, and $W$) need to be stored either locally or in a distributed manner. For a GNN on large graphs, it is impractical for a single worker (e.g., a GPU processor) to store all three matrices simultaneously. At least one matrix must be partitioned and distributed across different workers. The execution of distributed SpMM in a GNN can be divided into three stages: communication, computation, and reduction. The computation stage is the core of matrix multiplication, which is performed locally in each worker. Prior to the computation stage, workers may require specific blocks of a matrix from other workers, necessitating a communication stage. Typically, the communication stage is accomplished through a broadcasting mechanism. After the computation stage, a worker may only possess partial results of the final matrix block. In such cases, a reduction stage is necessary to gather the remaining partial results from other workers. Workers retrieve and sum up these partial results to obtain the final matrix.

According to the existence of the above three stages, we categorize the distributed SpMM into three execution models: \textit{computation-only}, \textit{communication-computation}, and \textit{communication-computation-reduction}. The existence of the communication and reduction stage is determined by two factors: ①\textbf{partition strategy}, how the matrix is partitioned and stored; and ②\textbf{stationary strategy}, which matrix is kept as the stationary matrix (i.e., no communication is incurred to move data in this matrix). The partition strategy determines whether a matrix is replicated stored on multiple workers or partitioned into blocks and distributed to different workers. The partition strategy also dictates the matrix partitioning mechanisms, such as 1D, 2D, etc. The stationary strategy determines the choice of the stationary matrix. During the execution of distributed SpMM, each partitioned block of the stationary matrix is pinned on the corresponding worker, and no communication is required for the stationary matrix. We summarize how these factors impact the execution model in Table 2, which will be introduced in detail in the following.
As discussed in Section 2, the weight parameter matrix $W$ is relatively small in GNN models. As a result, $W$ is fully replicated across all workers, adhering to the basic data parallelism principle. The subsequent discussion primarily focuses on matrices $A, H,$ and their product $P = AH$.

**Computation-only Execution Model.** In this model, both the communication and reduction stages are eliminated by adopting a specific partition strategy. One of $A$ or $H$ is fully replicated across each worker, while the other matrix is partitioned properly among different workers to achieve a communication-free paradigm [72]. For instance, matrix $A$ is replicated across all workers and matrix $H$ is partitioned into column blocks. Each worker holds a column block, which contains multiple columns of $H$. In this case, no communication is required prior to local computation. After local computation, each worker holds a corresponding column block of the final matrix $P = AH$, thus, no reduction is required either. In this case, only the computation stage is performed in the SpMM operation. However, this execution model lacks strong scalability if both matrices $A$ and $H$ exceed the memory capacity of an individual worker since at least one of the matrices needs to be fully replicated across all workers.

**Communication-Computation Execution Model.** For the circumstances in which both matrices $A$ and $H$ have to be partitioned and stored on the workers in a distributed manner, the communication-computation execution model is introduced. In this execution model, the workers need to share the matrix partitions they hold with each other, necessitating the communication stage prior to local computation. This communication can be performed either in a broadcast fashion or a P2P fashion, as detailed in Section 7. Furthermore, since the reduction stage is not executed, a $P$-Stationary strategy is adopted in the communication-computation execution model and no communication is required to obtain the resulting matrix $P$.

The third column in Table 2 shows that when a $P$-Stationary strategy is adopted, many partition strategies (1D, 1.5D, and 2D) can be applied in the communication-computation execution model. For $P$-Stationary 1D partitioning, each worker stores a row block of matrices $A, H,$ and $P$. During the communication stage, each worker broadcasts its row block of $H$ to all other workers. Subsequently, local computation is performed to compute the block rows of $P$. In this paradigm, matrix $A$ is also stationary, making this 1D $P$-Stationary SpMM also $A$-Stationary. It is worth noting that the 1D partitioning can also be performed in a column-wise manner [46]. In such cases, the 1D $P$-Stationary is also $H$-Stationary. In short, under 1D partitioning, both $A$-Stationary and $H$-Stationary are equivalent to $P$-Stationary. Therefore, the 1D partitioning follows a communication-computation execution model. However, 1D $P$-Stationary faces scalability challenges as a worker needs to broadcast the partition to all remote workers, resulting in linear communication costs proportional to the number of workers [110]. To address this, optimizations such as P2P communication and non-blocking techniques [29, 94] can be employed to accelerate the communication stage, which will be discussed in Section 7.

For $P$-Stationary 1.5D partitioning, either matrix $A$ or matrix $H$ is partitioned in a 2D manner, whereas the other matrix is partitioned in a 1D manner. To partition a matrix in a 2D manner, each processor holds a row-column block of the complete matrix, comprising elements in the matrix that satisfy both the assigned column IDs and row IDs for the processor. Under 1.5D partitioning, although matrix $P$ is set to be stationary, either matrix $A$ or matrix $H$ needs to be broadcast to all processors, leading to scalability challenges similar to 1D partitioning.

Another approach is to leverage $P$-Stationary 2D partitioning, where both matrix $A$ and matrix $H$ are partitioned in a 2D manner. For a row-column block in matrix $P$, the processor holding this block also holds the corresponding row-column blocks for matrix $A$ and matrix $H$. It only needs to receive the blocks with the same row ID of matrix $A$ and the blocks with the same column ID of matrix $H$. In this way, the total communication overhead is reduced.
Table 2. Two Factors in Distributed SpMM Execution Model

| Replicated | A-Stationary | H-Stationary | P-Stationary | Non-stationary |
|------------|--------------|--------------|--------------|----------------|
| 1D         | CC           | CC           | CC           | -              |
| 1.5D       | CCR          | CCR          | CC           | -              |
| 2D         | CCR          | CCR          | CC           | -              |
| 3D         | -            | -            | -            | CCR            |

The rows indicate the partition strategy and columns indicate the stationary strategy. C: Computation-only. CC: Communication-computation. CCR: Communication-computation-reduction. Non-stationary corresponds to 3D partition, where none of the three matrices is kept stationary during the execution.

Communication-computation-reduction Execution Model. As described above, adopting a P-Stationary strategy is the key to eliminating the reduction stage. Therefore, in this model, a P-Stationary strategy is not adopted. In other words, different from the communication-computation execution model in which a P-Stationary strategy is adopted, other stationary strategies, including A-Stationary, H-Stationary, and Non-Stationary, are considered. As discussed previously, under both replicated and 1D partitioning, all stationary strategies are equivalent to P-Stationary, and no reduction stage is required. For A-Stationary and H-Stationary, under 1.5D and 2D partitioning, the results obtained after the local computation stage are still partial. Each worker performs a reduction operation to sum up these remote partial results with its local partial result. The 3D partitioning can be viewed as the aggregation of multiple 2D partitioning SpMM operations. As a result, a final reduction operation is required to aggregate the results of multiple 2D partitioning SpMM operations. Under 3D partitioning, none of the three matrices A, H, P can be stationary. Therefore, 3D partitioning only corresponds to Non-stationary strategy.

Discussion. Both partition strategy and stationary strategy play crucial roles in distributed SpMM computation. We summarize the impact of these two factors on the distributed SpMM execution model as shown in Table 2. For the circumstances in which one of A and H is replicated and stored, a communication-free SpMM can be performed, leading to a computation-only execution model. However, since the GNN model consists of both SpMM and GeMM (General dense Matrix Multiplication) operations, a redistribution step is required between them to achieve both communication-free SpMM and GeMM [72]. Additionally, the order in which SpMM and GeMM are executed (i.e., SpMM first or GeMM first) can also impact the communication and computation overhead [72, 127]. When both matrices A and H are distributed across different processors and the 1D partitioning method or the P-Stationary is adopted, SpMM follows the communication-computation execution model. However, this execution model faces scalability and communication overhead. Finally, for A-Stationary or H-Stationary with 1.5D or 2D partitioning, and for the Non-stationary strategy, they follow the communication-computation-reduction execution model. The communication cost in the overall SpMM operation varies depending on the selected stationary mechanism, and different matrix partitioning methods contribute differently to the communication cost, as discussed in more detail in previous studies [100, 110].

6.2.3 Comparison of Graph View and Matrix View. We provide a comparison between the graph view and matrix view of GNN models and highlight the connections between these two views. The graph view and matrix view are two different programming interfaces that handle GNN models at different levels. The matrix view is a lower-level representation of GNN models, as it directly manages the data presented in matrices. On the other hand, the graph view provides a higher-level understanding of GNN models. In most GNN frameworks such as DGL and PyG, the programming
interfaces are organized based on the graph view for ease of use and convenience. However, during model training, graph operations such as feature aggregation are ultimately transformed into matrix multiplications. Both the one-shot execution and chunk-based execution in the graph view can be mapped to the matrix view. In a simple one-shot execution, each vertex is assigned to a specific worker along with its input features. The worker performs an aggregation when it receives the features of all its neighbors. This corresponds to the 1D partitioning $P$-Stationary in the matrix view, which follows a communication-computation paradigm. In other stationary mechanisms that follow a communication-computation-reduction paradigm, the partial results obtained from the reduction operation correspond to the partial aggregation results in the chunk-based execution from the graph view.

6.2.4 Synchronous Execution Model. In synchronous execution model (Figure 9(a)), the computation of each operator begins after the communication is synchronized. Almost all distributed SpMM executions follow a synchronous model. Therefore, to achieve a better understanding of the difference between synchronous and asynchronous execution models while also adhering to the page limit, we introduce the following from the graph view only with the SAGA-NN model as an example. In a SAGA-NN model, four operators and their backward counterparts form an execution pipeline for the graph data, following the order of SC-AE-GA-AV-$\nabla$AV-$\nabla$GA-$\nabla$AE-$\nabla$SC.

Among these eight stages, two involve the communication of states of boundary vertices, that is, GA and $\nabla$GA. In GA, neighbor vertices features should be aggregated to the target vertices, thus, the features of boundary vertices should be transferred. In $\nabla$GA, the gradients of the boundary vertices should be sent back to the workers they belong to. Therefore, GA and $\nabla$GA are two synchronization points in a synchronous execution model. On these points, the execution flow has to be blocked until all communication finishes. Systems such as NeuGraph [86], CAGNET [110], FlexGraph [121], and DistGNN [89] apply this execution model. To reduce communication cost and improve training efficiency, several communication protocols are proposed, which we review in Section 7.1. We point out that these two synchronization points exist irrespective of the view of execution model. From a matrix view, GA corresponds to the $AH$ SpMM operation, and $\nabla$GA corresponds to the $AG$ SpMM operation, where $G$ represents the gradient matrix in backward propagation.

6.2.5 Asynchronous Execution Model. The asynchronous execution model allows the computation to start with historical states and avoids the expensive synchronization cost. According to the different types of states, we classify the asynchronous execution model into type I asynchronization and type II asynchronization.

Type I asynchronization. As mentioned above, two synchronization points exist in the execution pipeline of the SAGA-NN model, and similar synchronization points exist in other programming models as well. Removing such synchronization points of computation graph computation
in the execution pipeline introduces type I asynchronization [89, 94, 117]. In type I asynchronization (Figure 9(b)), workers do not wait for the hidden embeddings (hidden features after the first GNN layer) of boundary vertices to arrive. Instead, it uses the historical hidden embeddings of the boundary vertices from previous epochs which are cached or received earlier to perform the aggregation of target vertices. Similar historical gradients are used in the ∇GA stage.

Type II asynchronization. During GNN training, another synchronization point occurs when the weight parameters need to be updated. This is identical to traditional deep learning models, where many efforts have been made to remove this synchronization point [92, 98]. Gandhi and Iyer [46] further adopt this idea into GNN models, which forms type II asynchronization. Under such protocol, mini-batches are executed in parallel and form a mini-batch pipeline.

Note that type I and type II asynchronization can be adopted individually or jointly. Detailed communication protocol for a GNN-specific Type I asynchronous execution model is described in Section 7.2.

7 GNN COMMUNICATION PROTOCOLS

In this section, we review GNN communication protocols that support different computation graph execution models, which are responsible for transferring hidden embeddings and gradients. According to the types of computation graph execution models, we classify GNN communication protocols into synchronous GNN communication protocols and asynchronous GNN communication protocols.

7.1 Synchronous GNN Communication Protocols

Synchronous GNN communication protocols handle the communication in a synchronous execution model, where a synchronous barrier is set in GA and ∇GA of each layer, to ensure that the hidden embeddings used for aggregation are computed from the exact last layer in the current iteration. There are several approaches to transferring the hidden embeddings; we introduce them one by one as follows.

7.1.1 Broadcast-Based Protocol. Broadcasting is a general technique to share information among a set of workers. In the GNN context, each worker broadcasts the latest hidden embeddings or graph structure it holds according to the underlying data partitioning methods, and other workers start the corresponding graph convolutional computation once they receive the required data from that worker. When all workers finish broadcasting, the hidden embeddings of the full graph are shared among all workers, and all workers receive the latest hidden embeddings they need in the current iteration. The authors of CAGNET [110] model the GNN computation as matrix computation. They partition the matrices (i.e., graph adjacency matrix and feature matrix) into sub-matrices with 1-D, 1.5-D, 2-D, and 3-D graph partitioning strategies. During the GNN training, each worker directly broadcasts the partial matrix of graph structure or vertex features to other workers for synchronization, guaranteeing that the vertices on a worker have a complete neighborhood. The efficiency of the broadcast can be destroyed by the workload imbalance problem, which causes a stall in computation.

7.1.2 P2P-Based Protocol. The aforementioned broadcast sends data to all the workers in the distributed environment and might cause redundant communication. In contrast, P2P transmission is a fine-grained communication method to share information among workers, which only transfers data alongside cross edges in an edge-cut graph partition or replicated vertices in a vertex-cut graph partition. For example, ParallelGCN [29] leverages edge-cut graph partitioning, and maintains a special data structure to manage the different vertex sets for different worker pairs. Each vertex set indicates the vertices that need to be sent between the two workers. The communication
of these vertices is then performed in a P2P fashion. The communication volume is thus reduced
since only necessary vertices are sent and received. DistGNN [89] employs the vertex-cut partitioning strategy. Each edge is assigned to one specific worker, while each vertex along with its features may be stored repetitively. Among the replications, one vertex is called the master vertex.

When aggregating the neighbors of a vertex, P2P transmission is applied. Specifically, partial aggregations of the replications of the master vertex are sent to the worker where the master vertex is located instead of broadcasting to all workers. As a result, the master vertex can aggregate the embeddings from all its neighbors. However, according to the studies of CAGNET [110], the naïve P2P transmission brings in the overhead of request-and-send operations and increases the latency. To mitigate this extra overhead, pipeline-based protocol is introduced, which further overlaps the communication with local computation, as described in Section 7.1.3.

7.1.3 Pipeline-Based Protocol. Since the naïve P2P-based protocol leads to an extra overhead of request-and-send operations introduced in Section 7.1.2, a pipeline-based protocol is proposed as an optimization technique. It divides the complete task of vertex aggregation on workers to several sub-tasks and overlaps the communication and computation of these sub-tasks. The execution of these sub-tasks thus forms a pipeline.

G3 [118] introduces a bin packing mechanism to pack local vertices into different bins (i.e., sub-tasks). During the aggregation stage, some vertices may be ready to execute the aggregation of the next layer prior to other vertices once all of their remote neighbors’ embeddings are received. These vertices are prioritized to be packed into bins and to compute the aggregation results. Once the computation of a bin finishes, the aggregation result of the vertices in that bin is sent to remote workers. In this way, a fine-grained overlap of communication and computation can be achieved, and different bins form a pipeline.

In systems that adopt the chunk-based execution model introduced in Section 6.2.1, such as SAR [91], DistGNN [89], and FlexGraph [121], the aggregation of each neighborhood chunk can be treated as a sub-task, providing the opportunity to overlap the communication and computation of different chunks. Here we call the aggregation of a vertex’s partial neighborhood the partial aggregation. SAR [91] aggregates the chunks in a predefined order. Each worker in SAR first computes the partial aggregation of the local neighborhood of the target vertex then fetches the remote neighborhood in the predefined order, computes the partial aggregation of the remote neighborhood, and accumulates the results locally one-by-one. Consequently, the aggregation of the complete neighborhood in SAR is divided into different stages and, in each stage, a partial aggregation is performed. The final aggregation is computed when all partial aggregation completes. SAR applies the pipeline-based communication protocol to reduce memory consumption since it does not need to store all the features of the neighborhood and compute the aggregation at once. ParallelGCN adopts a similar idea without pre-defining the aggregation order. The remote chunks of neighborhood are received in a random order, and it starts aggregating the received vertices immediately. Furthermore, to reduce the overhead of network communication caused by transferring of the partial neighborhood, DeepGalois [54] performs a remote partial aggregation before communication. Therefore, only the partial aggregation results need to be transferred. The cd-0 communication strategy in DistGNN is similar, which simply performs remote partial aggregation, fetches it to the local worker, and computes the final aggregation result. To further improve the training efficiency, FlexGraph [121] overlaps the computation and communication. Each worker first issues a request to partially aggregate the neighborhood at the remote worker and then computes the local partial aggregation while waiting for the remote partial aggregations to complete the transfer. After receiving the remote partial aggregations, the worker directly aggregates them with the local partial aggregation. Therefore, the local partial aggregation is overlapped with the communication to get remote partial aggregation.
Distributed Graph Neural Network Training

Table 3. Summary of Asynchronous Communication Protocols

| Type I | Type II | Use Stale Embedding | Use Stale Gradient | Staleness Model | Staleness bound | Parallelism | Mechanism       |
|--------|---------|---------------------|-------------------|----------------|----------------|-------------|----------------|
| ✓      | ✗       | ✓                   | ✓                 | Epoch-adaptive | s              | Data        | Embedding server |
| ✓      | ✗       | ✓                   | ✓                 | Epoch-adaptive | s              | Data        | Embedding server |
| DistGNN| ✓       | ✓                   | ✓                 | Epoch-fixed    | 3              | Model       | Pipeline       |
| PipeGCN| ✓       | ✓                   | ✓                 | Epoch-fixed    | 3              | Model       | Pipeline       |
| P3     | ✗       | ✓                   | ✓                 | Epoch-fixed    | 0              | Data        | Mini-batch pipeline |
| CM-GCN | -       | -                   | -                 | -              | -              | -           | -              |

The staleness bound $s$ is set by users.

7.1.4 Communication Via Shared Memory. To train large GNNs at scale, it is also feasible to leverage the CPU memory to retrieve the required information instead of GPU–GPU communication. The complete graph and feature embeddings are stored in shared memory (i.e., CPU memory) and the device memory of each GPU is treated as a cache. In ROC [65], the authors assume that all the GNN data is fit in the CPU memory and they repetitively store the whole graph structure and features in the CPU DRAM on each worker. GPU-based workers retrieve the vertex features and hidden embeddings from the local CPU-shared memory. For larger graphs or scenarios in which all the GNN data cannot fit into the CPU DRAM of a single worker, distributed shared memory is preferred. DistDGL [164] partitions the graph and stores the partitions with a specially designed KVStore. During the training, if the data is co-located with the worker, DistDGL accesses them via local CPU memory; otherwise, it incurs Remote Procedure Call (RPC) requests to retrieve information from other remote CPU memory. In NeuGraph [86], the graph structure data and hidden embeddings are also stored in shared memory. To support large-scale graphs, the input graph is partitioned into $P \times P$ edge chunks by the 2D partitioning method, and the feature matrix (as well as hidden embedding matrix) is partitioned into $P$ vertex chunks by the row-wise partition method. As described in Section 6.2.1, it follows a chunk-based execution model. To compute the partial aggregation of a chunk, each GPU fetches the corresponding edge chunk as well as the vertex chunk from the CPU memory. To speed up the vertex chunk transfer among GPUs, a chain-based streaming scheduling method is applied considering the communication topology to avoid bandwidth contention.

7.2 Asynchronous GNN Communication Protocol

Asynchronous GNN communication protocols are used to support asynchronous execution models. As described in Section 6.2.5, there are two asynchronizations in GNN training: type I asynchronization, which removes the synchronization in aggregation stages; and type II asynchronization, which removes synchronization in model weight updating stages. Type I asynchronization is specific to GNN model computation, whereas type II asynchronization has been widely used in traditional deep learning and the corresponding techniques can be directly introduced into GNN training. In the following, we focus on reviewing the asynchronous GNN communication protocols that are specific to distributed GNN training. Table 3 summarizes the popular asynchronous GNN communication protocols.

7.2.1 Staleness Models for Asynchronous GNN Communication Protocol. Generally, bringing in asynchronization means information with staleness should be used in training. Specifically, in type I asynchronization, the GA or ∇GA stage is performed without completely gathering the latest states of the neighborhood, and the historical vertex embeddings or gradients are used in aggregation. Different staleness models are introduced to maintain the staleness of historical information; each should ensure that the staleness of aggregated information is bounded so that the GNN training is converged. In the following, we review three popular staleness models.
**Epoch-Fixed Staleness.** One straightforward method is to aggregate historical information with fixed staleness [46, 89, 117]. Let $e$ be the current epoch in training and let $\tilde{e}$ be the epoch of the historical information used in aggregation. In the epoch-fixed staleness model, $|\tilde{e} - e| = \epsilon_e$, where $\epsilon_e$ is a hyper-parameter set by users. In this way, the staleness is bounded explicitly by $\epsilon_e$.

**Epoch-Adaptive Staleness.** Another variation of the above basic model is to use an epoch-adaptive staleness [18, 108]. During the aggregation of a vertex $i$, let $\tilde{e}_{N_j}$ be the epoch of historical information of any vertex $j \in N(i)$ (i.e., any neighbor vertex of $i$) used in aggregation. In the epoch-adaptive staleness model, for all $j \in N(i)$, $|\tilde{e}_{N_j} - e| \leq \epsilon_A$ holds. This means that in different epochs, the staleness of historical information for aggregation may be different, and in one epoch, the staleness of aggregated neighbors of a vertex may also be different. Generally, once $\tilde{e} - e$ reaches $\epsilon_A$, then the latest embeddings or gradients should be broadcast under decentralized training or pushed to the historical embedding server under centralized training. If the above condition does not hold when aggregation is performed, then the aggregation is blocked until the historical information within the staleness bound (i.e., $\epsilon_A$) can be retrieved.

**Variation-Based Staleness.** Third, the staleness can also be measured by the variation of embeddings or gradients. In other words, we only aggregate embeddings or gradients when they are significantly changed. Specifically, let $H_w^{(l)}$ be the embeddings of layer $l$ held by worker $W$ and let $\tilde{H}_w^{(l)}$ be the historical embedding last shared with other workers. In the variation-based staleness model, $H_w^{(l)} - \tilde{H}_w^{(l)} \leq \epsilon_V$, where $\epsilon_V$ is the maximum difference bound set by users so that the latest embeddings or gradients are broadcast if the historical version available for other workers are too stale. In this way, the staleness is bounded with the help of $\epsilon_V$.

7.2.2 Realization of Asynchronous GNN Communication Protocols. As described above, historical information can be used in aggregation during GA or VGA stages. Some protocols only take the GA stage into consideration, supporting asynchronous embedding aggregation. Some protocols are designed to use historical information during both GA and VGA stages, supporting both asynchronous embedding and gradient aggregation.

Asynchronous Embedding Aggregation. Peng et al. [94] take the basic training paradigm of broadcast in CAGNET into consideration and design a skip-broadcast mechanism under a 1D partition. This mechanism auto-checks the staleness of the partitioned vertex features on each compute vertex and skips the broadcast if it is not too stale. If the broadcast of a partition is skipped, then other workers should use the historical vertex embeddings cached previously to process the forward computation. Three staleness check algorithms are further proposed, each one of which ensures that the staleness of cached vertex embeddings is bounded.

Chai et al. [18] adopt a similar idea while using parameter servers to maintain the historical embeddings of all workers. The vertex hidden embeddings are pushed to these servers every several epochs, and the workers pull the historical embeddings to their local cache in the next epoch after the push. The staleness of historical vertex embeddings is thus bounded by the push–pull period.

The cd-r algorithms introduced by Md et al. [89] overlap the communication of partial aggregation results from each worker with the forward computation in a GNN. Specifically, the partial aggregation in iteration $i$ is transmitted asynchronously to the target vertex, and the final aggregation is performed in iteration $i + r$. Under this algorithm, the staleness is bounded to $r$.

Asynchronous Embedding and Gradient Aggregation. As we introduced in Section 6.2.4, GA and VGA are two synchronization points in the GNN execution model. While the above protocols only use historical vertex embeddings during stage GA, Wan et al. [117] also take the synchronization point VGA into consideration and use stale vertex gradients in the back propagation. During the training, both embeddings and gradients from the neighborhood are asynchronously sent in a P2P fashion in the last epoch and received by the target vertex in the current epoch.
Therefore, the communication is overlapped in both forward and backward computation and the training pipeline with a fixed-epoch gap is thus constructed, in which workers are only allowed to use features or gradients exactly one epoch ago, and the staleness is bounded. Thorpe et al. [108] design a finer and more flexible pipeline. Similar to the method proposed by Wan et al. [117], both stale vertex embeddings and vertex gradients are used in the pipeline. Moreover, it also removes the type II synchronization; thus, a trainer may use stale weight parameters and start the next epoch immediately. The staleness in the pipeline is explicitly bounded with $S$ set by users. The bounded staleness $S$ ensures that the trainer that moves the fastest in the pipeline is allowed to use historical embeddings or gradients at most $S$ epochs earlier. With this staleness bound, the staleness of weight parameters is also bounded accordingly.

**Vertex-Level Asynchronization.** Without using stale embeddings or gradients, asynchronization can also be designed in the vertex level [158], where each vertex starts the computation of the next layer as soon as all its neighbors’ embeddings are received. In contrast to the traditional synchronous method in which all vertices on a worker start the computation of a layer together, during vertex-level asynchronous processing, different vertices in one worker may be computing different layers at the same time. Note that this asynchronization does not make any difference in the aggregation result, since all the information required should be the latest, and no embeddings or gradients from previous epochs are used.

### 8 DISTRIBUTED GNN TRAINING SYSTEMS

In recent years, many distributed GNN systems have been proposed as a combination of distributed graph processing systems and distributed deep learning systems. The distributed GNN systems take the general idea of efficiently processing graph data from the former systems, while combining the capability of processing structured data from the latter systems. A recent survey systematically introduced the evolution from the graph processing system to the GNN system [111]. Based on the taxonomy proposed in this survey, we review 28 distributed GNN training systems and frameworks as the representatives summarized in Table 4, and mainly classify them into three categories.
according to the computation resources: (a) systems on a single machine with multi-GPUs, (b) systems on GPU clusters, and (c) systems on CPU clusters. In addition, there are a few distributed GNN training systems using more diversified computation resources, such as serverless threads, mobile devices, and edge devices. We classify them as miscellaneous systems. In the table, we make a summarization of the techniques the systems incorporate according to our new taxonomy. Note that for the column of execution model, we ignore the conventional, one-shot, and synchronous execution models as they are the basic execution models adopted by most systems. Therefore, we only note the execution models of systems with special designs. For a detailed introduction of each system and their relations, please refer to our online supplemental materials.

9 FUTURE DIRECTION

As a general solution to training on large-scale graphs, distributed GNN training has gained widespread attention in recent years. In addition to the techniques and systems discussed above, there are other interesting but emerging research topics in distributed GNN training. We discuss some interesting directions in the following.

Benchmark for distributed GNN training. Many efforts have been made in benchmarking traditional deep learning models. For instance, DAWNBench [26] provides a standard evaluation criterion to compare different training algorithms. It focuses on the end-to-end training time consumed to converge to a state-of-the-art accuracy as well as the inference time. Both single machine and distributed computing scenarios are considered. Furthermore, many benchmark suites are developed for traditional DNN training to profile and characterize the computation [33, 88, 172]. As for GNNs, Dwivedi et al. [35] attempt to benchmark the GNN pipeline and compare the performance of different GNN models on medium-size datasets. Meanwhile, larger datasets [34, 42, 56] for graph machine learning tasks have been published. However, to our knowledge, few efforts have been made to compare the efficiency of different GNN training algorithms, especially in distributed computing scenarios. GNNMark [8] is the first benchmark suite developed specifically for GNN training. It leverages the NVIDIA nvprof profiler [13] to characterize kernel-level computations. The NVBit framework [114] profiles memory divergence and further modifies PyTorch source code to collect data sparsity. However, it lacks flexibility for quick profiling of different GNN models and does not pay much attention to profiling the distributed and multi-machine training scenarios. Therefore, it would be practical for a new benchmark to be designed for large-scale distributed GNN training.

Large-scale dynamic graph neural networks. In many applications, graphs are not static. The vertex attributes or graph structures often evolve with changes, which requires the representations to be updated in time. Li and Chen [74] proposed a general cache-based GNN system to accelerate representation updating. It sets a cache for hidden representations and selects valuable representations to save time for updating. DynaGraph [51] efficiently trains GNNs via cached message passing and timestep fusion. Furthermore, it optimizes the graph partitioning in order to train GNNs in a data-parallel method. Although dynamic GNNs has long been an interesting area of research, as far as we know, there are no more works that specifically focus on distributed dynamic GNN training. The dynamism, including features and structure, poses new challenges to the ordinary solution in a distributed environment. Graph partitioning has to quickly adjust to the change of vertices and edges while meeting the requirements of load balance and communication reduction. The update of the graph structure drops the cache hit ratio, which significantly influences the end-to-end performance of GNN training.

Large-scale heterogeneous graph neural networks. Many heterogeneous GNN structures have been proposed in recent years [19, 44, 77, 125, 148, 159]. However, few distributed systems take the unique characteristic of heterogeneous graphs into consideration to support
heterogeneous GNNs. Since the size of the feature and the number of neighbors may vary greatly for vertices with different types, processing heterogeneous graphs in a distributed manner may cause severe problems, such as load imbalance and high network overhead. The Paddle Graph Learning framework provides easy and fast programming of message passing–based graph learning on heterogeneous graphs with distributed computing. DistDGLv2 [165] takes the imbalanced workload partition into consideration and leverages the multi-constraint technique in METIS to mitigate this problem. More attention should be paid to this research topic to address the above problems.

**GNN model compression technique.** Although model compression, including pruning, quantization, weight sharing, and more, is widely used in deep learning, it has not been extensively applied in distributed GNN training. The compression on network structures such as pruning [169] can be contacted with a graph sampling strategy to solve the OOM problem. Model quantization is another promising approach to improving the scalability of GNN models. SGQuant [38] is a GNN-tailored quantization algorithm that develops multi-granularity quantization and auto-bit selection. Degree-Quant [106] stochastically protects (high-degree) vertices from quantization to improve weight update accuracy. BinaryGNN [5] applies a binarization strategy inspired by the latest developments in binary neural networks for images and knowledge distillation for graph networks. For the distributed settings, Song et al. [103] recently proposed EC-Graph for distributed GNN training with CPU clusters, which aims to reduce the communication costs by message compression. It adopts lossy compression and designs compensation methods to mitigate the induced errors. Meanwhile, a Bit-Tuner is used to keep the balance between model accuracy and message size. GNN model compression is orthogonal to the aforementioned distributed GNN optimization techniques, which deserves more attention to help improve the efficiency of distributed GNN training.

10 CONCLUSION

Distributed GNN training is one of the successful approaches of scaling GNN models to large graphs. In this survey, we systematically reviewed the existing distributed GNN training techniques from graph data processing to distributed model execution and covered the life-cycle of end-to-end distributed GNN training. We divide the distributed GNN training pipeline into three stages, data partition, batch generation, and GNN model training, which heavily influence GNN training efficiency. To clearly organize the new technical contributions that optimize these stages, we proposed a new taxonomy that consists of four orthogonal categories: GNN data partition, GNN batch generation, GNN execution model, and GNN communication protocol. In the GNN data partition category, we described the data partition techniques for distributed GNN training; in the GNN batch generation category, we presented techniques of fast GNN batch generation for mini-batch distributed GNN training; in the GNN execution model category, we discussed the execution model used in mini-batch and full-graph training; and in the GNN communication protocol category, we discussed both synchronous and asynchronous protocols for distributed GNN training. After carefully reviewing the techniques in the four categories, we summarized existing representative distributed GNN systems for multi-GPUs, GPU-clusters, and CPU-clusters, and gave a discussion about the future direction in optimizing distributed GNN training.

REFERENCES

[1] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. 2021. Computing graph neural networks: A survey from algorithms to accelerators. ACM Comput. Surv. 54, 9 (2021), 1–38. https://doi.org/10.1145/3477141

1https://github.com/PaddlePaddle/PGL
[2] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In 12th USENIX Symposium on Operating Systems Design and Implementation. 265–283.

[3] David Ahmmed-Aristizabal, Mohammad Ali Armin, Simon Denman, Clinton Fookes, and Lars Petersson. 2021. Graph-based deep learning for medical diagnosis and analysis: Past, present and future. Sensors 21, 14 (2021), 4758.

[4] Alexandra Angerd, Keshav Balasubramanian, and Murali Annavaram. 2020. Distributed training of graph convolutional networks using subgraph approximation. arXiv preprint arXiv:2012.04930 (2020), 1–14.

[5] Mehdi Bahri, Gaetan Bahl, and Stefanos Zafeiriou. 2021. Binary graph neural networks. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 9492–9501.

[6] Youhui Bai, Cheng Li, Zhiqi Lin, Yuefu Wu, Youshun Miao, Yunxin Liu, and Yinlong Xu. 2021. Efficient data loader for fast sampling-based GNN training on large graphs. IEEE Transactions on Parallel and Distributed Systems 32, 10 (2021), 2541–2556.

[7] Ziv Bar-Yossef and Li-Tal Mashiach. 2008. Local approximation of pagerank and reverse pagerank. In Proceedings of the 17th ACM Conference on Information and Knowledge Management. 279–288.

[8] Trinayan Baruah, Kaustubh Shvidikar, Shi Dong, Yifan Sun, Saiful A. Mojumder, Kihoon Jung, José L. Abellán, Yash Ukidave, Ajay Joshi, John Kim, et al. 2021. GNName: A benchmark suite to characterize graph neural network training on GPUs. In 2021 IEEE International Symposium on Performance Analysis of Systems and Software. 13–23.

[9] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Matteusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. 2018. Relational inductive biases, deep learning, and graph networks. arXiv preprint arXiv:1806.01261 (2018), 1–40.

[10] Maciej Besta and Torsten Hoefler. 2022. Parallel and distributed graph neural networks: An in-depth concurrency analysis. arXiv preprint arXiv:2205.09702 (2022), 1–27.

[11] Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. 2013. Scalable matrix computations on large scale-free graphs using 2D graph partitioning. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. 1–12.

[12] Pietro Bongini, Monica Bianchini, and Franco Scarselli. 2021. Molecular generative graph neural networks for drug discovery. Neurocomputing 450 (2021), 242–252.

[13] Thomas Bradley. 2012. GPU performance analysis and optimisation. NVIDIA Corporation (2012), 1–117.

[14] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaishao Ma, James Cheng, and Fan Yu. 2021. DGCL: An efficient communication library for distributed GNN training. In Proceedings of the European Conference on Computer Systems. 130–144.

[15] Zhenkun Cai, Qihui Zhou, Xiao Yan, Da Zheng, Xiang Song, Chenguang Zheng, James Cheng, and George Karypis. 2023. DSP: Efficient GNN training with multiple GPUs. In Proceedings of the ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming. 392–404.

[16] Umit V. Catalyurek and Cevdet Aykanat. 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. IEEE Transactions on Parallel and Distributed Systems 10, 7 (1999), 673–693.

[17] Yukuo Cen, Zhenyu Hou, Yan Wang, Qibin Chen, Yi Shen Luo, Xingcheng Yao, Aohan Zeng, Shiguang Guo, Peng Zhang, Guohao Dai, et al. 2021. CogDL: An extensive toolkit for deep learning on graphs. arXiv preprint arXiv:2103.00959 (2021), 1–11.

[18] Zheng Chai, Guangji Bai, Liang Zhao, and Yue Cheng. 2022. Distributed graph neural network training with periodic historical embedding synchronization. arXiv preprint arXiv:2206.00057 (2022), 1–20.

[19] Yaomin Chang, Chuan Chen, Weibo Hu, Zibin Zheng, Xiaocong Zhou, and Shouzhi Chen. 2022. MEGNN: Meta-path extracted graph neural network for heterogeneous graph representation learning. Knowledge-Based Systems 235 (2022), 107611.

[20] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast learning with graph convolutional networks via importance sampling. In International Conference on Learning Representations. 1–15.

[21] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting distributed synchronous SGD. arXiv preprint arXiv:1604.00981 (2016), 1–10.

[22] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic training of graph convolutional networks with variance reduction. In Proceedings of the 35th International Conference on Machine Learning (Vol. 80). 942–950.

[23] Xiaobing Chen, Yuke Wang, Xinfeng Xie, Xing Hu, Abanti Basak, Ling Liang, Mingyu Yan, Lei Deng, Yuefu Wu, Zidong Du, et al. 2022. Rubik: A hierarchical architecture for efficient graph neural network training. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 41, 4 (2022), 936–949.

[24] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An efficient algorithm for training deep and large graph convolutional networks. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 257–266.
Distributed Graph Neural Network Training

[25] Edward Choi, Zhen Xu, Yujia Li, Michael Dusenberry, Gerardo Flores, Emily Xue, and Andrew Dai. 2020. Learning the graphical structure of electronic health records with graph convolutional transformer. In Proceedings of the AAAI Conference on Artificial Intelligence. 606–613.

[26] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. 2017. Dawnbench: An end-to-end deep learning benchmark and competition. Training 100, 101 (2017), 102.

[27] Weilin Cong, Rana Forsati, Mahmut Kandemir, and Mehrdad Mahdavi. 2020. Minimal variance sampling with provable guarantees for fast training of graph neural networks. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 1393–1403.

[28] Gabriele Corso, Luca Cavalleri, Dominique Beaini, Pietro Liò, and Petar Velickovic. 2020. Principal neighbourhood aggregation for graph nets. In Advances in Neural Information Processing Systems. 13260–13271.

[29] Gunduz Vehbi Demirci, Aparajita Haldar, and Hakan Ferhatosmanoglu. 2023. Scalable graph convolutional network training on distributed-memory systems. In Proceedings of the VLDB Endowment, Vol. 16. 711–724.

[30] Xiang Deng and Zhongfei Zhang. 2021. Graph-free knowledge distillation for graph neural networks. In Proceedings of the 30th International Joint Conference on Artificial Intelligence. 2321–2327.

[31] Kien Do, Truyen Tran, and Svetha Venkatesh. 2019. Graph transformation policy network for chemical reaction prediction. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 750–760.

[32] Jialin Dong, Da Zheng, Lin F. Yang, and George Karypis. 2021. Global neighbor sampling for mixed CPU-GPU training on giant graphs. In Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining. 289–299.

[33] Shi Dong and David Kaeli. 2017. DNNMark: A deep neural network benchmark suite for GPUs. In Proceedings of the General Purpose GPUs. 63–72.

[34] Yuanqi Du, Shiyu Wang, Xiaojue Guo, Hengning Cao, Shujie Hu, Junji Jiang, Aishwarya Varala, Abhinav Angirekula, and Liang Zhao. 2021. GraphGT: Machine learning datasets for graph generation and transformation. In 35th Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2), 1–17.

[35] Vijay Prakash Dwivedi, Chaitanya K. Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. 2020. Benchmarking graph neural networks. arXiv preprint arXiv:2003.00982 (2020), 1–47.

[36] Wenqi Fan, Yao Ma, Ming Zhao, Shujie Hu, Junji Jiang, Aishwarya Varala, Abhinav Angirekula, and Liang Zhao. 2021. GraphGT: Machine learning datasets for graph generation and transformation. In 35th Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2), 1–17.

[37] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch geometric. arXiv preprint arXiv:1903.02428 (2019), 1–9.

[38] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. 2017. Protein interface prediction using graph convolutional networks. In Proceedings of the 31st International Conference on Neural Information Processing Systems. 6533–6542.

[39] Fabrizio Frasca, Emanuele Rossi, Davide Eynard, Ben Chamberlain, Michael Bronstein, and Federico Monti. 2020. SIGN: Scalable inception graph neural networks. arXiv preprint arXiv:2004.11198 (2020), 1–17.

[40] Scott Freitas, Yuxiao Dong, Joshua Neil, and Duen Horng Chau. 2021. A large-scale database for graph representation learning. In Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1), 1–13.

[41] Qi Jin, Yuke Wang, Xu Li, Shu Yang, Xueqiao Peng, and Yufei Ding. 2020. SGQuant: Squeezing the last bit of graph neural network computation on GPU. In Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing. 122–134.

[42] Xinyu Fu, Jiani Zhang, Ziqiao Meng, and Irwin King. 2020. MAGNN: Metapathaggregated graph neural network for heterogeneous graph embedding. In Proceedings of The Web Conference 2020. 2331–2341.

[43] Yasuhiro Fujiwara, Yasutoshi Ida, Atsutoshi Kumagai, Masahiro Nakano, Akisato Kimura, and Naonori Ueda. 2023. Efficient network representation learning via cluster similarity. Data Sci. Eng, 8, 3 (2023), 279–291.

[44] Xinliang Fu, Jiali Zhang, Ziqiao Meng, and Irwin King. 2020. SIGN: Scalable inception graph neural networks. arXiv preprint arXiv:2004.11198 (2020), 1–17.

[45] Scott Freitas, Yuxiao Dong, Joshua Neil, and Duen Horng Chau. 2021. A large-scale database for graph representation learning. In Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1), 1–13.

[46] Qiang Fu, Yuede Ji, and H. Howie Huang. 2022. TLPGNN: A lightweight two-level parallelism paradigm for graph neural network computation on GPU. In Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing, 122–134.

[47] Fabrizio Frasca, Emanuele Rossi, Davide Eynard, Ben Chamberlain, Michael Bronstein, and Federico Monti. 2020. SIGN: Scalable inception graph neural networks. arXiv preprint arXiv:2004.11198 (2020), 1–17.
[49] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). 17–30.

[50] Daniele Grattarola and Cesare Alippi. 2021. Graph neural networks in TensorFlow and Keras with Spectral [application notes]. IEEE Computational Intelligence Magazine 16, 1 (2021), 99–106.

[51] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. 2022. DynaGraph: Dynamic graph neural networks at scale. In Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management & Systems (GRAD & Systems Data Analytics (NDA)). 1–10.

[52] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In Proceedings of the 31st International Conference on Neural Information Processing Systems. 1025–1035.

[53] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. 2020. LightGCN: Simplifying and powering graph convolution network for recommendation. In Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval. 639–648.

[54] Loc Hoang, Xuhao Chen, Hochan Lee, Roshan Datathtri, Gurbinder Gill, and Keshav Pingali. 2021. Efficient distribution for deep learning on large graphs. In Proceedings of the 1st MLSys Workshop on Graph Neural Networks and Systems. 1–9.

[55] Linmei Hu, Siyong Xu, Chen Li, Cheng Yang, Chuan Shi, Nan Duan, Xing Xie, and Ming Zhou. 2020. Graph neural news recommendation with unsupervised preference disentanglement. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. 4255–4264.

[56] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. Advances in Neural Information Processing Systems 33 (2020), 22118–22133.

[57] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. Featgraph: A flexible and efficient backend for graph neural network systems. In SC’20: International Conference for High Performance Computing, Networking, Storage and Analysis. 1–13.

[58] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: General-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. 1–12.

[59] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and bridging the gaps in current GNN performance optimizations. In Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 119–132.

[60] Linyong Huang, Zhe Zhang, Zhaoyang Du, Shuangchen Li, Hongzhong Zheng, Yuan Xie, and Nianxiong Tan. 2022. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Advances in Neural Information Processing Systems, Vol. 35. 1–10.

[61] Tinglin Huang, Yuxiao Dong, Ming Ding, Zhen Yang, Wenzheng Feng, Xinyu Wang, and Jie Tang. 2021. MixGCF: An improved training method for graph neural network-based recommender systems. In Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining. 665–674.

[62] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive sampling towards fast graph representation learning. In Proceedings of the 32nd International Conference on Neural Information Processing Systems. 4563–4572.

[63] Yanping Huang, Yulong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, Hyoukjoong Lee, Jiqian Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient training of giant neural networks using pipeline parallelism. In Advances in Neural Information Processing Systems, Vol. 32. 1–10.

[64] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating graph sampling for graph machine learning using GPUs. In Proceedings of the 16th European Conference on Computer Systems. 311–326.

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

[66] Feng Jiang and Masuma Akter Rumi. 2021. Communication-efficient sampling for distributed training of convolutional networks. arXiv preprint arXiv:2101.07706 (2021), 1–11.

[67] Weiwei Jiang and Jiayun Luo. 2022. Graph neural network for traffic forecasting: A survey. Expert Systems with Applications 207 (2022), 117921.

[68] Chaityana K. Joshi. 2022. Recent advances in efficient and scalable graph neural networks. Retrieved from https://www.chaitjo.com/post/efficient-gnn/ (2022).

[69] Tim Kaler, Alexandros Iliopoulos, Philip Murzynowski, Tao Schardl, Charles E. Leiserson, and Jie Chen. 2023. Communication-efficient graph neural networks with probabilistic neighborhood expansion analysis and caching. In Proceedings of Machine Learning and Systems. 1–14.

[70] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing 20, 1 (1998), 359–392.
[71] Yunyong Ko, Kibong Choi, Jiwon Seo, and Sang Wook Kim. 2021. An in-depth analysis of distributed training of deep neural networks. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium. 994–1003.

[72] Süreyya Emre Kurt, Jinghua Yan, Aravind Sukumaran-Rajam, Prashant Pandey, and P. Sadasivan. 2023. Communication optimization for distributed execution of graph neural networks. In IEEE International Parallel and Distributed Processing Symposium. 512–523.

[73] Matthias Langer, Zhen He, Wenny Rahayu, and Yanbo Xue. 2020. Distributed training of deep learning models: A taxonomic perspective. IEEE Transactions on Parallel and Distributed Systems 31, 12 (2020), 2802–2818.

[74] Haoyang Li and Lei Chen. 2021. Cache-based GNN system for dynamic graphs. In Proceedings of the 30th ACM International Conference on Information & Knowledge Management. 937–946.

[75] Houyi Li, Yongchao Liu, Yongyong Li, Bin Huang, Peng Zhang, Guowei Zhang, Xintan Zeng, Kefeng Deng, Wenguang Chen, and Changhua He. 2021. GraphTheta: A distributed graph neural network learning system with flexible training strategy. arXiv preprint arXiv:2104.10569 (2021), 1–18.

[76] Hongzheng Li, Yingxia Shao, Junping Du, Bin Cui, and Lei Chen. 2022. An I/O-efficient disk-based graph system for scalable second-order random walk of large graphs. Proceedings of the VLDB Endowment 15, 8 (2022), 1619–1631.

[77] Longhai Li, Lei Duan, Junuchen Wang, Chengxin He, Zhihao Chen, Guicai Xie, Song Deng, and Zhaohang Luo. 2023. Memory-enhanced transformer for representation learning on temporal heterogeneous graphs. Data Sci. Eng. 8, 2 (2023), 98–111.

[78] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch distributed: Experiences on accelerating data parallel training. Proceedings of the VLDB Endowment 13, 12 (2020), 3005–3018.

[79] Shuangli Li, Jingbo Zhou, Tong Xu, Liang Huang, Fan Wang, Haoyi Xiong, Weili Huang, Dejing Dou, and Hui Xiong. 2021. Structure-aware interactive graph neural networks for the prediction of protein-ligand binding affinity. In Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining. 975–985.

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

[81] Meng Liu, Youzhi Luo, Limei Wang, Yaochen Xie, Hao Yuan, Shurui Gui, Haiyang Yu, Zhao Xu, Jingtun Zhang, Yi Liu, et al. 2021. DIG: A turnkey library for diving into graph deep learning research. Journal of Machine Learning Research 22 (2021), 1–9.

[82] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. 2023. BGL: GPU-efficient GNN training by optimizing graph data I/O and preprocessing. In Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation. 103–118.

[83] Xin Liu, Mingyu Yan, Lei Deng, Guoqi Li, Xiaochun Ye, Dongrui Fan, Shurui Pan, and Yuan Xie. 2022. Survey on graph neural network acceleration: An algorithmic perspective. In Proceedings of the 31st International Joint Conference on Artificial Intelligence. 5521–5529.

[84] Zirui Liu, Kaixiong Zhou, Fan Yang, Li Li, Rui Chen, and Xia Hu. 2021. EXACT: Scalable graph neural networks training via extreme activation compression. In International Conference on Learning Representations. 1–32.

[85] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A framework for machine learning and data mining in the cloud. Proceedings of the VLDB Endowment 5, 8 (2012), 716–727.

[86] Lingxia Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel deep neural network computation on large graphs. In 2019 USENIX Annual Technical Conference. 443–458.

[87] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In Proceedings of the ACM SIGMOD International Conference on Management of Data. 135–146.

[88] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debo Dutta, Udit Gupta, Kim Hazelwood, Andy Hock, Xinyuan Huang, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. 2020. MLPerf training benchmark. In Proceedings of Machine Learning and Systems. 336–349.

[89] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K. Ahmed, and Sasikanth Avancha. 2021. DistGNN: Scalable distributed training for large-scale graph neural networks. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 1–14.

[90] Seung Won Min, Kun Wu, Mert Hidayetoglu, Jinjun Xiong, Xiang Song, and Wen-mei Hwu. 2022. Graph neural network training and data tiering. In Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. 3555–3565.
[91] Hesham Mostafa. 2022. Sequential aggregation and rematerialization: Distributed full-batch training of graph neural networks on large graphs. In Proceedings of Machine Learning and Systems. 265–275.

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

[93] Santosh Pandey, Lingda Li, Adolfo Hoisie, Xiaoye S. Li, and Hang Liu. 2020. C-SAW: A framework for graph sampling and random walk on GPUs. In International Conference for High Performance Computing, Networking, Storage and Analysis. 1–15.

[94] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. SANCUS: Staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. Proceedings of the VLDB Endowment 15, 9 (2022), 1937–1950.

[95] Md Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. 2021. FusedMM: A unified SDDMM-SpMM kernel for graph embedding and graph neural networks. In 2021 IEEE International Parallel and Distributed Processing Symposium. 256–266.

[96] Morteza Ramezani, Weilin Cong, Mehrdad Mahdavi, Mahmut T. Kandemir, and Anand Sivasubramaniam. 2021. Learn locally, correct globally: A distributed algorithm for training graph neural networks. arXiv preprint arXiv:2111.08202 (2021), 1–32.

[97] Jiahua Rao, Xiang Zhou, Yutong Lu, Huiying Zhao, and Yuedong Yang. 2021. Imputing single-cell RNA-seq data by combining graph convolution and autoencoder neural networks. Iiscience 24, 5 (2021), 102393.

[98] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In Advances in Neural Information Processing Systems. 1–9.

[99] Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. 2018. Graph networks as learnable physics engines for inference and control. In Proceedings of the 35th International Conference on Machine Learning. 4470–4479.

[100] Oguz Selvitopi, Benjamin Brock, Israt Nisa, Alok Tripathy, Katherine Yelick, and Aydin Buluç. 2021. Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication. In Proceedings of the ACM International Conference on Supercomputing. 431–442.

[101] Shihui Song and Peng Jiang. 2022. Rethinking graph data placement for graph neural network training on multiple GPUs. In Proceedings of the 36th ACM International Conference on Supercomputing. 1–10.

[102] Zheng Song, Fengshan Bai, Jianfeng Zhao, and Jie Zhang. 2021. Spammer detection using graph-level classification model of graph neural network. In 2021 IEEE 2nd International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering. 531–538.

[103] Zhen Song, Yu Gu, Jianzhong Qi, Zhigang Wang, and Ge Yu. 2022. EC-Graph: A distributed graph neural network system with error-compensated compression. In 2022 IEEE 38th International Conference on Data Engineering. 648–660.

[104] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 1222–1230.

[105] Jie Sun, Li Su, Zuocheng Shi, Wenting Shen, Zeke Wang, Lei Wang, Jie Zhang, Yong Li, Wenyan Sun, Jingren Zhou, and Fei Wu. 2023. Legion: Automatically pushing the envelope of multi-GPU system for billion-scale GNN training. In Proceedings of the USENIX Annual Technical Conference. 165–179.

[106] Shyam A. Tailor, Javier Fernandez-Marques, and Nicholas D. Lane. 2020. Degree-Quant: Quantization-aware training for graph neural networks. arXiv preprint arXiv:2008.05000 (2020), 1–22.

[107] Qiaoyu Tan, Ninghao Liu, and Xia Hu. 2019. Deep representation learning for social network analysis. Frontiers in Big Data 2 (2019), 2.

[108] John Thorpe, Yifan Qiao, Jonathan Yelofson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Myung Kim, et al. 2021. Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In USENIX Symposium on Operating Systems Design and Implementation. 495–514.

[109] Chao Tian, Lingxiao Ma, Zhi Yang, and Yafei Dai. 2020. PCGCN: Partition-centric processing for accelerating graph convolutional network. In 2020 IEEE International Parallel and Distributed Processing Symposium. 936–945.

[110] Alok Tripathy, Katherine Yelick, and Aydin Buluç. 2020. Reducing communication in graph neural network training. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. 1–14.

[111] Jana Vatter, Ruben Mayer, and Hans-Arno Jacobsen. 2023. The evolution of distributed systems for graph neural networks and their origin in graph processing and deep learning: A survey. Comput. Surveys (2023), 1–35.

[112] Petar Veličković, Guillem Cucurull, Aranxta Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph attention networks. In International Conference on Learning Representations. 1–12.

[113] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. 2020. A survey on distributed machine learning. ACM Computing Surveys 53, 2 (2020), 1–33.
[114] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs. In Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture. 372–383.

[115] Borui Wan, Juntao Zhao, and Chuan Wu. 2023. Adaptive message quantization and parallelization for distributed full-graph GNN training. In Proceedings of Machine Learning and Systems. 1–15.

[116] Cheng Wan, Youjie Li, Ang Li, Nam Sung Kim, and Yingyan Lin. 2022. BNS-GCN: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. In Proceedings of Machine Learning and Systems. 673–693.

[117] Cheng Wan, Youjie Li, Cameron R. Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. 2022. PipeGCN: Efficient full-graph training of graph convolutional networks with pipelined feature communication. In International Conference on Learning Representations. 1–24.

[118] Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. 2023. Scalable and efficient full-graph GNN training for large graphs. In Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data. 1–23.

[119] Hanchen Wang, Defu Lian, Ying Zhang, Lu Qin, Xiangjian He, Yiguang Lin, and Xuemin Lin. 2021. Binarized graph neural network. In World Wide Web. 825–848.

[120] Junfu Wang, Yunhong Wang, Zhen Yang, Liang Yang, and Yuanfang Guo. 2021. Bi-GCN: Binary graph convolutional network. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 1561–1570.

[121] lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. 2021. FlexGraph: A flexible and efficient distributed framework for GNN training. In Proceedings of the 16th European Conference on Computer Systems. 67–82.

[122] M. Wang, W. Fu, X. He, S. Hao, and X. Wu. 2022. A survey on large-scale machine learning. IEEE Transactions on Knowledge & Data Engineering 34, 06 (2022), 2574–2594.

[123] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zhiwei Ye, Mufei Li, Linjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In ICLR Workshop on Representation Learning on Graphs and Manifolds. 1–7.

[124] Pengyu Wang, Chao Li, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. 2023. Scalable and efficient full-graph GNN training for large graphs. In Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data. 1–23.

[125] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S. Yu. 2019. Heterogeneous graph attention network. In The World Wide Web Conference. 2022–2032.

[126] Yuke Wang, Boyuan Feng, and Yufei Ding. 2022. QGTC: Accelerating quantized graph neural networks via GPU tensor core. In Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 107–119.

[127] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs. In 15th USENIX Symposium on Operating Systems Design and Implementation. 515–531.

[128] Max Welling and Thomas N. Kipf. 2017. Semi-supervised classification with graph convolutional networks. In J. International Conference on Learning Representations. 1–14.

[129] Cameron R. Wolfe, Jingkang Yang, Arindam Chowdhury, Chen Dun, Artun Bayer, Santiago Segarra, and Anastasios Kyrillidis. 2021. GIST: Distributed training for large-scale graph convolutional networks. arXiv preprint arXiv:2102.10424 (2021), 1–28.

[130] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. 2019. Simplifying graph convolutional networks. In Proceedings of the 36th International Conference on Machine Learning. 6861–6871.

[131] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. 2022. Graph neural networks in recommender systems: A survey. Computer Surveys (2022), 1–37.

[132] Yongji Wu, Defu Lian, Yiheng Xu, Le Wu, and Enhong Chen. 2020. Graph convolutional networks with Markov random field reasoning for social spammer detection. In Proceedings of the AAAI Conference on Artificial Intelligence. 1054–1061.

[133] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyuan Li, Chengguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: Vertex-centric programming for graph neural networks. In Proceedings of the 16th European Conference on Computer Systems. 359–375.

[134] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A comprehensive survey on graph neural networks. IEEE Transactions on Neural Networks and Learning Systems 32, 1 (2021), 4–24.

[135] Shuo Xiao, Dongqing Zhu, Chaogang Tang, and Zhenzhen Huang. 2023. Combining graph contrastive embedding and multi-head cross-attention transfer for cross-domain recommendation. Data Sci. Eng. 8, 3 (2023), 247–262.
[136] Yaochen Xie, Zhao Xu, Jingtun Zhang, Zhengyang Wang, and Shuiwang Ji. 2022. Self-supervised learning of graph neural networks: A unified review. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 1 (2022), 1–1.

[137] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. 2022. Graphiler: Optimizing graph neural networks with message passing data flow graph. In *Proceedings of Machine Learning and Systems*. 515–528.

[138] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How powerful are graph neural networks?. In *International Conference on Learning Representations*. 1–17.

[139] Ning Xu, Bin Cui, Lei Chen, Zi Huang, and Yingxia Shao. 2015. Heterogeneous environment aware streaming graph partitioning. In *IEEE Conference on Knowledge and Data Engineering*. 1560–1572.

[140] Zihui Xue, Yuedong Yang, Mengtian Yang, and Radu Marculescu. 2022. SUGAR: Efficient subgraph-level training via resource-aware graph partitioning. *arXiv preprint arXiv:2202.00075* (2022), 1–16.

[141] Bencheng Yan, Chaokun Wang, Gaoyang Guo, and Yunkai Lou. 2020. TinyGNN: Learning efficient graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1848–1856.

[142] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: A factored system for sample-based GNN training over GPUs. In *Proceedings of the 17th European Conference on Computer Systems*. 417–434.

[143] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. KnightKing: A fast distributed graph random walk engine. In *Proceedings of the ACM Symposium on Operating Systems Principles*. 524–537.

[144] Hongbo Yin, Yingxia Shao, Xupeng Miao, Yawen Li, and Bin Cui. 2022. Scalable graph sampling on GPUs with compressed graph. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 2383–2392.

[145] Yuning You, Tianlong Chen, Zhangyang Wang, and Yang Shen. 2020. L2-GCN: Layer-wise and learned efficient training of graph convolutional networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1–9.

[146] Hanqing Zeng and Viktor Prasanna. 2020. GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms. In *2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 255–265.

[147] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph sampling based inductive learning method. In *International Conference on Learning Representations*. 1–19.

[148] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V. Chawla. 2019. Heterogeneous graph neural network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 793–803.

[149] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. 2020. AGL: A scalable system for industrial-purpose graph machine learning. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3125–3137.

[150] Guo Zhang, Hao He, and Dina Katabi. 2019. Circuit-GNN: Graph neural networks for distributed circuit design. In *Proceedings of the 36th International Conference on Machine Learning*. 7364–7373.

[151] Hengrui Zhang, Zhongming Yu, Guohao Dai, Guyue Huang, Yufei Ding, Yuan Xie, and Yu Wang. 2022. Understanding GNN computational graph: A coordinated computation, IO, and memory perspective. In *Proceedings of Machine Learning and Systems*. 467–484.

[152] Weijia Zhang, Hao Liu, Yanchi Liu, Jingbo Zhou, and Hui Xiong. 2020. Semi-supervised hierarchical recurrent graph neural network for city-wide parking availability prediction. In *The 34th AAAI Conference on Artificial Intelligence*. 1186–1193.

[153] Wentao Zhang, Xupeng Miao, Yingxia Shao, Jiawei Jiang, Lei Chen, Olivier Ruas, and Bin Cui. 2020. Reliable data distillation on graph convolutional network. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1399–1414.

[154] Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. 2023. DUCATI: A dual-cache training system for graph neural networks on giant graphs with the GPU. In *Proceedings of the ACM on Management of Data*. 1–24.

[155] Xiao-Meng Zhang, LiLiang, LinLiu, and Ming-Jing Tang. 2021. Graph neural networks and their current applications in bioinformatics. *Frontiers in Genetics* 12 (2021), 1–22.

[156] Yuning You, Tianlong Chen, Zhangyang Wang, and Yang Shen. 2020. L2-GCN: Layer-wise and learned efficient training of graph convolutional networks. In *Proceedings of the 36th International Conference on Machine Learning*. 7364–7373.

[157] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Yuan Qi. 2020. AGL: A scalable system for industrial-purpose graph machine learning. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3125–3137.

[158] Guo Zhang, Hao He, and Dina Katabi. 2019. Circuit-GNN: Graph neural networks for distributed circuit design. In *Proceedings of the 36th International Conference on Machine Learning*. 7364–7373.

[159] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How powerful are graph neural networks?. In *International Conference on Learning Representations*. 1–17.

[160] Ning Xu, Bin Cui, Lei Chen, Zi Huang, and Yingxia Shao. 2015. Heterogeneous environment aware streaming graph partitioning. In *IEEE Conference on Knowledge and Data Engineering*. 1560–1572.

[161] Zihui Xue, Yuedong Yang, Mengtian Yang, and Radu Marculescu. 2022. SUGAR: Efficient subgraph-level training via resource-aware graph partitioning. *arXiv preprint arXiv:2202.00075* (2022), 1–16.

[162] Bencheng Yan, Chaokun Wang, Gaoyang Guo, and Yunkai Lou. 2020. TinyGNN: Learning efficient graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1848–1856.
[160] Taige Zhao, Xiangyu Song, Jianxin Li, Wei Luo, and Imran Razzak. 2021. Distributed optimization of graph convolutional networks using subgraph variance. arXiv preprint arXiv:2110.02987 (2021), 1–12.

[161] Yiren Zhao, Duo Wang, Daniel Bates, Robert Mullins, Mateja Jamnik, and Pietro Lio. 2020. Learned low precision graph neural networks. arXiv preprint arXiv:2009.09232 (2020), 1–14.

[162] Chengguang Zheng, Hongzhai Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. 2022. ByteGNN: Efficient graph neural network training at large scale. Proceedings of the VLDB Endowment 15, 6 (2022), 1228–1242.

[163] Chuanpan Zheng, Xiaoliang Fan, Cheng Wang, and Jianzhong Qi. 2020. GMAN: A graph multi-attention network for prediction. In Proceedings of the AAAI Conference on Artificial Intelligence. 1234–1241.

[164] Da Zheng, Chao Ma, Minjie Wang, Jingjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed graph neural network training for billion-scale graphs. In 10th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms. 36–44.

[165] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. 2022. Distributed hybrid CPU and GPU training for graph neural networks on billion-scale heterogeneous graphs. In Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. 4582–4591.

[166] Xun Zheng, Chen Dan, Bryon Aragam, Pradeep Ravikumar, and Eric P. Xing. 2020. Learning sparse nonparametric DAGs. In The 23rd International Conference on Artificial Intelligence and Statistics. 3414–3425.

[167] Shanna Zhong, Jiahui Wang, Kun Yue, Liang Duan, Zhengbao Sun, and Yan Fang. 2023. Few-shot relation prediction of knowledge graph via convolutional neural network with self-attention. Data Sci. Eng. 8, 4 (2023), 385–395.

[168] Wanjun Zhong, Jingjing Xu, Duyu Tang, Zhenan Xu, Nan Duan, Ming Zhou, Jiahai Wang, and Jian Yin. 2020. Reasoning over semantic-level graph for fact checking. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. 6170–6180.

[169] Hongkuan Zhou, Ajitesh Srivastava, Hanqing Zeng, Rajgopal Kannan, and Viktor Prasanna. 2021. Accelerating large scale real-time GNN inference using channel pruning. Proceedings of the VLDB Endowment 14, 9 (2021), 1597–1605.

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

[171] Zhe Zhou, Cong Li, Xuechao Wei, and Guangyu Sun. 2021. GcCNear: A hybrid architecture for efficient GCN training with near-memory processing. arXiv preprint arXiv:2111.00680 (2021), 1–15.

[172] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. 2018. Benchmarking and analyzing deep neural network training. In 2018 IEEE International Symposium on Workload Characterization. 88–100.

[173] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A comprehensive graph neural network platform. Proceedings of the VLDB Endowment 12, 12 (2019), 2094–2105.

[174] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. 2019. Layer-dependent importance sampling for training deep and large graph convolutional networks. In Proceedings of the 33rd International Conference on Neural Information Processing Systems. 11249–11259.

Received 31 October 2022; revised 9 December 2023; accepted 31 January 2024