Scalable and Efficient Full-Graph GNN Training for Large Graphs

XINCHEN WAN*, Hong Kong University of Science and Technology, Hong Kong SAR, China
KAIQIANG XU*, Hong Kong University of Science and Technology, Hong Kong SAR, China
XUDONG LIAO, Hong Kong University of Science and Technology, Hong Kong SAR, China
YILUN JIN, Hong Kong University of Science and Technology, Hong Kong SAR, China
KAI CHEN, Hong Kong University of Science and Technology, Hong Kong SAR, China
XIN JIN, Peking University, China

Graph Neural Networks (GNNs) have emerged as powerful tools to capture structural information from graph-structured data, achieving state-of-the-art performance on applications such as recommendation, knowledge graph, and search. Graphs in these domains typically contain hundreds of millions of nodes and billions of edges. However, previous GNN systems demonstrate poor scalability because large and interleaved computation dependencies in GNN training cause significant overhead in current parallelization methods.

We present G3, a distributed system that can efficiently train GNNs over billion-edge graphs at scale. G3 introduces *GNN hybrid parallelism* which synthesizes three dimensions of parallelism to scale out GNN training by sharing intermediate results peer-to-peer in fine granularity, eliminating layer-wise barriers for global collective communication or neighbor replications as seen in prior works. G3 leverages *locality-aware iterative partitioning* and *multi-level pipeline scheduling* to exploit acceleration opportunities by distributing balanced workload among workers and overlapping computation with communication in both inter-layer and intra-layer training processes. We show via a prototype implementation and comprehensive experiments that G3 can achieve as much as 2.24× speedup in a 16-node cluster, and better final accuracy over prior works.

CCS Concepts: • **Information systems → Data management systems**; • **Computing methodologies → Distributed computing methodologies**.

Additional Key Words and Phrases: graph neural network, distributed training, GPU, hybrid parallelism

ACM Reference Format:
Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. 2023. Scalable and Efficient Full-Graph GNN Training for Large Graphs. *Proc. ACM Manag. Data* 1, 2, Article 143 (June 2023), 23 pages. https://doi.org/10.1145/3589288

1 INTRODUCTION

Graph-structured data are natural representations of many real-world applications such as social networks and knowledge graphs. Recent works extend deep neural networks (DNNs) to capture

*Equal contribution.

Authors' addresses: Xinchen Wan, Hong Kong University of Science and Technology, Hong Kong SAR, China, xinchen.wan@connect.ust.hk; Kaiqiang Xu, Hong Kong University of Science and Technology, Hong Kong SAR, China, kxuar@cse.ust.hk; Xudong Liao, Hong Kong University of Science and Technology, Hong Kong SAR, China, xliaoaf@connect.ust.hk; Yilun Jin, Hong Kong University of Science and Technology, Hong Kong SAR, China, yilun.jin@connect.ust.hk; Kai Chen, Hong Kong University of Science and Technology, Hong Kong SAR, China, kaichen@cse.ust.hk; Xin Jin, Peking University, Beijing, China, xinjinpku@pku.edu.cn.

2836-6573/2023/6-ART143 $15.00
https://doi.org/10.1145/3589288

structural information in graphs [14, 23, 33, 44]. This new family of DNNs, known as graph neural networks (GNNs), achieves state-of-the-art performance in machine learning tasks such as node classification [43, 46] and recommendation systems [47].

The reason behind GNN's high expressiveness is that GNNs learn from the relationships between data samples while traditional DNNs are trained over individual samples with no structural information. Figure 1 shows GNN computation process that includes *neighborhood aggregation* operations and standard NN operations: in each GNN layer, one node's new embedding is calculated by first aggregating its neighbors' embeddings from the previous layer, and then applying NN operations. This computation process repeats when a node travels through each GNN layer, capturing information from the multi-hop neighborhood of the node.

Training GNNs over billion-edge graphs is time-consuming because of the neighborhood aggregation operation. The common practice to scale out DNN training over large-scale input data is data parallelism [24]. Data parallelism splits data into multiple independent partitions, which can be trained on different workers in parallel. However, data parallelism can no longer be applied to GNN training directly because the neighborhood aggregation operation takes neighboring data samples that may reside on a remote worker. Therefore, when data parallelism splits an input graph into partitions, cross-partition neighboring data samples create large and interleaved computation dependencies among these partitions, making the partitions dependent on each other in data-parallel training. The dependency pattern, which is determined by input graph structure rather than the GNN model itself, complicates the worker synchronization scheme and poses system challenges to efficiently train GNN models in parallel.

Due to the large and interleaved dependencies among partitions in data parallel training, existing GNN systems are difficult to scale to a large number of workers. A widely adopted approach in current GNN systems involves replicating out-of-partition neighboring node data to a worker in order to train GNN over the graph partition independently [11, 50], causing duplicate computation and communication systematically. Some systems such as DGL [39] and P^3 [11] use sampling-based training methods to mitigate the overhead by sampling only a small part of the graph during each training iteration. However, sampling operations can generate biased results [6, 7] and the duplicate work still exists and grows exponentially to the number of GNN layers [18].

Recently, full-graph training (*i.e.*, no sampling used) [29, 31, 34, 35] has been a popular topic in the machine learning research community for its better performance. While GNN systems including ROC [18] are able to support full-graph training, similar to the methods above, their evaluations still show limited scalability due to imbalanced and duplicate workload, especially when training in large clusters or with deep GNN models [1, 25]. Moreover, they are not built for large input graphs because the workers needs to load the entire graph into GPU memory for processing, which is not possible when processing billion-edge graphs.

In this paper, we present G3, a distributed system that can efficiently support full-graph training of GNNs at scale on billion-edge graphs and without accuracy compromise. G3 proposes *GNN hybrid parallelism* to eliminate duplicate work and effectively manage large and complex computation dependencies between workers (§3). Furthermore, G3 uses hybrid parallelism to scale out training by dividing the process at the per-node level and sharing intermediate results in a pipelined fashion among peers.

However, GNN hybrid parallelism is not free, as it comes with higher system overhead due to potential stragglers during layer-wise synchronization and more frequent data sharing. Therefore, to maintain a maximum degree of parallelism, G3 distributes balanced computation and communication workload across workers by dividing the input graph with a locality-aware iterative graph partitioning algorithm (§4). G3 also overlaps communication with computation using a multi-level pipeline scheduling algorithm (§5) that implements both inter- and intra- layer pipelines with an

adaptive bin packing strategy. In contrast to pipeline parallelism used in prior works [11, 30, 32] which comes at the cost of accuracy loss, G3's method does not compromise model accuracy.

We evaluate G3 on large datasets with representative GNN models and compare it with state-of-the-art GNN systems. Our results show that G3 can achieve up to 2.24× speedup in a 16-node cluster, and better final accuracy (~6% higher) over prior works. Besides, we demonstrate G3's ability in exploring complex GNN models and high-dimensional graph datasets: it can achieve up to 25.9× speedup with a 4-layer deep GNN model and 1.94× when training over graphs with 256-dimensional input features.

Overall, this paper makes the following contributions:

- G3 introduces GNN hybrid parallelism to handle large and complex computation dependencies, enabling scalable GNN training on large graphs without accuracy loss.
- G3 accelerates the training process by balancing workloads across workers with locality-aware iterative graph partitioning, and overlapping communication with computation using multi-level pipeline scheduling.
- We implement a G3 prototype and conduct comprehensive experiments over large graphs, showing that G3 achieves as much as 2.24× speedup in a 16-node cluster, and better final accuracy over prior works.

2 BACKGROUND AND MOTIVATION

We first review the computation process of GNN training and then demonstrate the scalability issues in distributed GNN training of prior work through analysis and experiments.

2.1 Graph Neural Networks (GNNs)

GNNs are a family of neural networks that learn node representations from graph-structured data (*i.e.*, nodes and links) [14, 23, 33, 44]. The key idea is to aggregate information from neighbors following the graph structure, and perform embedding transformations layer-by-layer.

GNN Computation in One Layer. Figure 1 shows the process of GNN computation on the blue node in GNN layer-k, which consists of two stages: neighbor aggregation stage and NN operations stage. In the aggregation stage, layer-k aggregates the node's neighboring nodes' embeddings calculated from the preceding layer (if layer-1, the embeddings are the corresponding nodes' input features). Afterwards, layer-k applies standard DNN operations such as matrix multiplication in Graph Convolution Network (GCN) [23], to the aggregation result, and finally outputs the layer-k embedding for the blue node. All other nodes in the graph process the same above computation in layer-k with their own neighbors.

Formally, the above computation process of GNN layer-k is expressed as follows:

$$\begin{aligned} a_v^{(k)} &= Aggregate^{(k)}(\{h_u^{(k-1)} \mid u \in \mathcal{N}(v)\}) \\ h_v^{(k)} &= Update^{(k)}(h_v^{(k-1)}, a_v^{(k)}) \end{aligned}$$

where $h_v^{(k)}$ denotes the node embedding of node v in the GNN layer-k, and $\mathcal{N}(v)$ denotes neighbors of v. For each node v in layer-k, *Aggregate* first outputs $a_v^{(k)}$ by gathering the embeddings of its neighbors $h_{u\in\mathcal{N}(v)}^{(k-1)}$ with an accumulation function such as mean or sum, then *Update* computes the node's new embedding from its previous embedding $h_v^{(k-1)}$ and the aggregation result $a_v^{(k)}$.

Forward/Backward Propagation. For an L-layer GNN model, the above computation iterates from layer-1 to L among all nodes in the forward propagation. Note that after the layer-L computation, the embeddings $h_v^{(L)}$ capture information for all neighbors within L hops of v, and can be

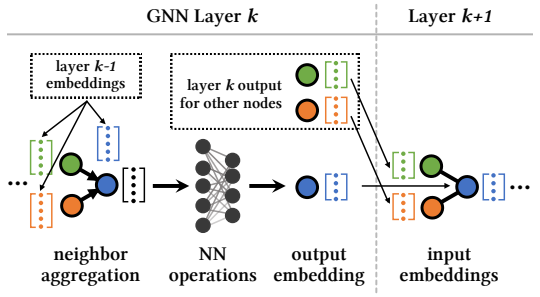


Fig. 1. Computation of one node in layer k by first aggregating its neighbors' embeddings from previous layer $k-1$, then applying NN operations.

used for downstream tasks such as node classification and link prediction. Next, all nodes' $h^{(L)}$ with their labels are fed to a loss function to generate the loss. With the loss, gradients are then calculated from layer-L to 1 following the chain rule in the backward propagation. Finally, the optimizer updates the GNN model with the gradients attached to it, ending the GNN training for one iteration.

As shown in Figure 1, GNN's propagation between consecutive layers is not only over the same data sample but also across different samples due to the neighborhood aggregation operations. These cross-data-sample propagation paths create computation dependencies between data samples.

2.2 DNN Parallelization Methods

Data, model, and pipeline parallelism are the most common parallelization methods used in distributed DNN training [12, 17, 26, 37, 38, 42]. However, we cannot directly apply these methods to GNN training due to its unique aggregation operations (§2.1). Here we briefly introduce how they can be adopted and implemented in GNN training, as well as their disadvantages that are specific to the GNN training process.

Data Parallelism. Data parallel training is widely implemented in current GNN systems [18, 28, 50, 51] as large graphs may not be able to fit in a single GPU memory and hence must be partitioned and processed in a distributed manner. Implementing data parallelism in GNN needs to partition the input graph. However, these graph partitions cannot be independently processed, due to the computation dependencies between these partitions created by GNN's neighbor aggregation operations. Existing systems adopt methods such as cross-partition neighbor replication to satisfy these computation dependencies. However, as we show in §2.3, these methods introduce system overhead and lead to scalability limitations.

Model Parallelism. In DNN training, model parallelism partitions a model across workers. The worker that holds the input layer of the model is fed with the training data, followed by intermediate results being transmitted sequentially across workers. As the training workflow is still serialized between workers, the advantage is not speeding up the training process, but reducing the memory footprint of a huge model. However, since GNNs typically use small models which commonly have only 2 to 5 layers, model parallelism is not used to accelerate training in previous GNN systems.

Pipeline Parallelism. In pipeline parallelism, the DNN model is partitioned and distributed across workers as in model parallelism. The training data is also split into batches, which are then streamed into the first worker. In the pipeline, every worker computes output activations or gradients for a batch and immediately propagates them to the downstream worker, which hence keeps workers busy and improves workers' utilizations. While pipeline parallelism assumes that

batches are entirely independent to process [30], this assumption no longer holds in GNN training. When a worker processes a batch, it needs to re-build a subgraph from the batch, which needs to include multi-hop out-of-batch neighbors for neighborhood aggregation operation in GNN, causing computation overhead due to duplicate work across workers. A study [8] shows that adopting pipeline parallelism in GNN training leads to significant efficiency degradation: training time increases by 5× in a 4-node cluster compared to the single GPU result.

2.3 Training GNNs on Large Graphs

Training GNNs over real-world graphs with tens of millions of nodes and billions of edges poses challenges in hardware capacity and overall efficiency. We put existing methods into two categories and summarize their techniques as follows.

Sampling-based GNN Training. Some GNN training methods use neighbor sampling strategies that down-sample nodes' neighbors in the neighborhood aggregation operation to reduce computation cost [6, 14, 36, 47, 48], and hence speed up the training process. However, as observed in previous works [16, 18], these techniques often ignore a large fraction of neighbors and thus may suffer from accuracy loss. For example, a classical GNN model, GraphSAGE [14], samples and aggregates only a small number (10 to 25) of neighbors for each node. Moreover, the duplicate work still exists and grows exponentially to the number of GNN layers, making neighbor sampling less efficient when training deep GNN models [1, 25]. A recent work [40] also reports that the sampling step is bounded by the I/O throughput of the storage and becomes the bottleneck limiting GPU utilization.

Besides, some sampling-based GNN training methods are used together with graph partitioning when the input graph is too large to fit into a single device. In this case, the result may suffer from both accuracy loss and scalability limitation as well [39].

Full-graph GNN Training. Efficient full-graph GNN training is especially challenging on large graphs as it is both computation-intensive and memory-intensive. Therefore, to improve overall efficiency, many existing GNN systems [18, 28, 50, 51] leverage data parallelism by partitioning and distributing the input graph to multiple workers that collaboratively train the same GNN model. However, the cross-partition computation dependencies in data parallel training make workers dependent on each other. These computation dependencies, determined by the input graph structure, are usually large and complex in real-world graphs. To satisfy the dependencies, two types of schemes are proposed in previous works:

- **Layer-wise communication barriers**. NeuGraph [28] implements layer-wise communication barriers for global synchronization of intermediate results. This communication introduces extra overhead and can be straggled by unbalanced workloads.
- **Cross-partition neighbor replication**. The method of loading all out-of-partition neighboring node features for each worker's assigned partition, as used in DGL [50], ROC [18], and other systems, leads to duplicate computation and communication. This is because the same data may be loaded and computed by multiple workers. Additionally, as GNN training aggregates node embeddings from an L-hop neighborhood, the amount of duplicate work increases exponentially with the number of GNN layers L.

Our experiment confirms the analysis. Figure 2 shows the speedup of training throughput for ROC (results from [18]) and DGL on the Reddit dataset [14]. When the number of workers scales from 2 to 16, the performance of DGL and ROC only increases as much as 1.4× and 3.2×, far lower than the ideal 8× speedup ratio, as represented by Linear.

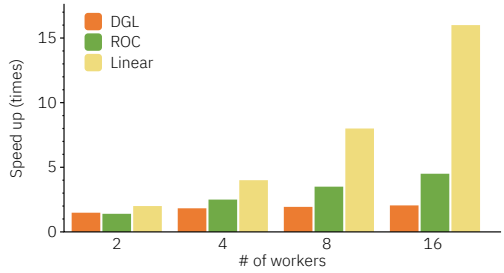


Fig. 2. Scalability of existing GNN systems without neighbor sampling on Reddit. Numbers are reported based on each system's throughput on 1-node. Linear represents the ideal speedup ratio.

Full-graph GNN training is actively researched in today's machine learning research community [31, 34, 40]. However, for the reasons described above, current GNN systems fall short of supporting full-graph training when high accuracy and efficiency are desired at the same time.

3 GNN HYBRID PARALLELISM

Existing systems are less efficient in training GNNs at scale due to the complex computation dependencies created by neighbor aggregation, and the naive adoption of existing DNN parallelization methods results in significant overhead.

We propose *GNN hybrid parallelism*, a novel parallelization strategy that avoids duplicate work by sharing intermediate results peer-to-peer between GNN layers.

3.1 GNN Hybrid Parallelism Workflow

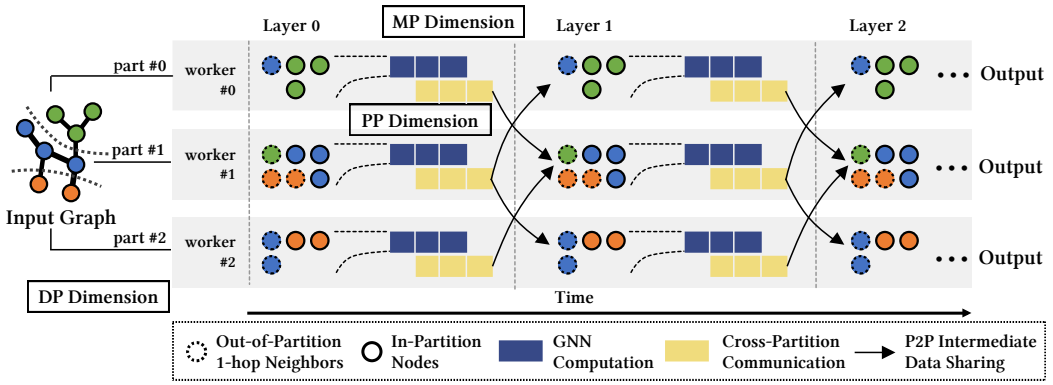


Fig. 3. A running example of hybrid-parallel training workflow on an 8-node graph with 3 workers. The workflow incorporates three parallel dimensions (data, model, and pipeline). The detailed explanation on the example is in §3.1.

G3 uses GNN hybrid parallelism to enable both data parallelism and model parallelism, while also pipelines the inter- and intra-layer training process. Figure 3 depicts a running example on an 8-node graph with 3 workers.

To begin with, the input graph is divided into smaller partitions for training, known as the **data-parallel dimension**. In the given example, an 8-node graph is split into three parts, each

marked with a different color. After partitioning, each partition is assigned to a separate worker, and edges connecting the partitions become computation dependencies between the workers.

The training process continues by having each worker train its own GNN model with the assigned partition. Once a worker completes computation on one layer, the output is not only used as input for the next layer computation on that worker, but also sent peer-to-peer to other workers that have dependencies on it (as demonstrated by the cross-worker data-sharing arrows in the given example). This creates the **model-parallel dimension**.

In each layer, nodes are divided into smaller groups called "bins" (depicted as colored rectangles in Figure 3) and processed one at a time by a worker, creating a pipeline opportunity for overlapping computation and communication: once a bin's computation is finished, the worker can send its output to other workers and start processing the next bin. These pipelines exist for both inter- and intra-worker tasks, forming the **pipeline-parallel dimension**.

The backward pass is similar to the forward pass, with the difference being that the inputs are gradients rather than node embeddings. Once an epoch is completed, the GNN model parameters are aggregated and averaged using the AllReduce method across all workers.

Benefits in GNN Hybrid Parallelism. GNN hybrid parallelism enables the system to scale with the growth of the graph size and model size for the reasons listed below.

First, hybrid parallelism handles interleaved computation dependencies in GNN training without the need for cross-partition replication, as the *model-parallel dimension* shares intermediate results peer-to-peer, reducing system overhead.

Second, hybrid parallelism creates opportunities to overlap computation and communication in inter- and intra-layer training by packing graph nodes into bins and processing them in pipelines.

3.2 System Challenges

The benefits of GNN hybrid parallelism come with challenges that must be overcome to fully exploit acceleration opportunities.

Challenge 1: Balanced Workload Distribution. Data parallelism is a technique that divides a workload into small partitions to process them simultaneously. In order to prevent stragglers in parallel processing, it is important to have a balanced workload in each partition. In GNN hybrid parallelism, communication across partitions becomes significant and it is important to balance the communication workload as well (§4.1). Previous works, such as DGL [50] and NeutronStar [40], use off-the-shelf graph partitioning algorithms that only balance computation workload but do not consider the added complexity of communication in GNN training. In GNN training, simply minimizing the global edge-cut on the input graph, like in graph processing tasks, cannot balance the communication workload over each partition (discuss in §4.1). To address this issue, we formulate the cost factors and propose a locality-aware iterative graph partitioning algorithm (§4) that efficiently balances workloads across workers.

Challenge 2: Efficient Pipeline Scheduling. In GNN hybrid parallelism, workers communicate with each other and share intermediate results to satisfy the GNN computation dependencies across workers. To construct efficient pipelines globally that overlap computation and communication, the order of computation is important as it affects the blocking time of each worker due to cross-worker dependencies. Existing methods, *e.g.*, HGL [13] and JasmineGraph [20], use simple FIFO strategies when processing graph nodes, which does not fully utilize the opportunity to overlap communication and computation. G3 tackles this problem with a multi-level pipeline scheduling algorithm (§5) which proactively schedules for both inter- and intra-layer training pipelines to maximize the chance of overlapping, with a bin packing mechanism to assign computation priority and adapt to memory and model size.

Notation	Description
$G = (V, E)$	Input graph with its node and edge set
$P(G, k) = \{V_1, ..., V_k\}$	k-partition result of graph G V_i represents the nodes in partition i
$E(V_i)$	Edges in G that destine to nodes in V_i
V_i^{remote}	Out-of-partition neighbors of partition i
d_v	Degree of node v from G
$N(v)$	Neighbors of node v from G

Table 1. Notations and factors affecting GNN workload over a partition in hybrid parallelism.

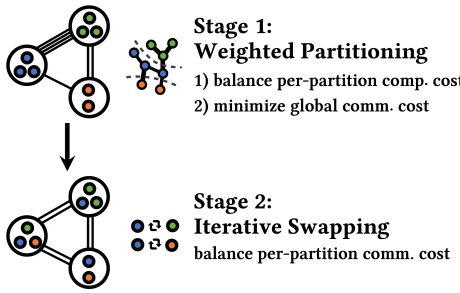


Fig. 4. Two stages of partitioning algorithm

3.3 Comparison with Recent Works

We note that similar ideas of GNN hybrid parallelism have been explored in some recent works [11, 34]. However, existing solutions fail to fully exploit the opportunities to reach the maximum degree of parallelism. They did not identify the challenges of high system synchronization overhead raised by the idea of GNN hybrid parallelism. More specifically, existing solutions (1) did not balance cross-partition communication in graph partitioning algorithms, and (2) did not explore more efficient pipelines with minimized waiting time by assigning different computation priorities.

G3 fully explores these challenges that are unique in distributed GNN training, presents comprehensive optimization and implementation to address the system challenges, and evaluates the solutions with both end-to-end and microbenchmark experiments.

Moreover, in the pipeline-parallel dimension, nodes are grouped into smaller groups called *bins* and processed in pipelines. This allows G3 to adapt bin sizes to the GPU memory capacity and enables G3 to handle larger graphs, unlike full-graph training systems like NeutronStar [40] and BNS-GCN [34] which may encounter GPU memory limitations with large graphs.

4 BALANCED WORK PARTITIONING

In distributed GNN training, achieving a balanced workload among partitions requires more than just equal graph nodes and global min edge-cut. In this section, we formulate the cost factors in distributed GNN training workload and propose a locality-aware iterative graph partitioning algorithm that efficiently balances these factors across workers.

4.1 Cost Factors in Distributed GNN Training

When assigning a graph partition to a worker, several graph-related factors affect the worker's training workload. We summarize several factors which determine the workload of partition V_i as follows:

(1) Number of nodes $|V_i|$, which directly contributes to the GNN computation cost.

(2) Number of edges $|E(V_i)|$, which affects the GNN computation cost due to neighborhood aggregation.

(3) Number of out-of-partition neighboring nodes $|V_i^{remote}|$, which determines the amount of data to be transferred out to other workers after each layer's computation. This is essentially the peer-to-peer communication cost in the model parallel dimension of hybrid parallelism.

Chunk-based graph partitioning method, used in NeuGraph [28], ROC [18] and NeutronStar [40], partitions the graph into chunks that contain nodes with consecutive IDs. While it balances $|V_i|$, the *actual* amount of workload may be highly unbalanced because the method does not consider

the total number of edge cuts, resulting in unbalanced $|E(V_i)|$ as well as large and unbalanced $|V_i^{remote}|$.

METIS algorithm [21], used by DGL [50] and BNS-GCN [34], can find a partition decision with minimum edge-cut. However, minimizing edge-cut does not balance $|V_i^{remote}|$ for each partition i. Hence, the communication workload across workers still varies.

4.2 Locality-aware Iterative Partitioning

The partitioning algorithm has two stages as depicted in Figure 4: (1) the weighted partitioning stage that balances per-partition computation cost and minimizes global communication costs. (2) the iterative re-partitioning stage that balances per-partition communication costs.

First, in the weighted graph partitioning stage, the algorithm generates a graph partitioning decision with a balanced computation cost among partitions while minimizing the global communication volume. The global communication volume is defined as the sum of each partition's communication volume, which is determined by the number of each partition's remote nodes. We formalize our optimization objectives as follows:

$$\text{minimize } T_{max\_nn} = \max_{V_p \in P} |V_p|, \tag{1}$$

$$\text{minimize } T_{max\_aggr} = \max_{V_p \in P} \sum_{v \in V_p} d_v, \tag{2}$$

$$\text{minimize } T_{total\_comm} = \sum_{V_p \in P} |V_p^{remote}|, \tag{3}$$

where $P = \{V_1, V_2, ...\}$ denotes the partitioning decision and V_i contains all nodes v in partition i. Equation 1 and 2 essentially balance computation cost among partitions while Equation 3 minimizes global communication cost.

The objectives above formulate a multi-constraint graph partitioning problem [22]: a vector of weights is assigned to each node, and the goal is to produce a partitioning such that the partitioning satisfies a balanced constraint associated with each weight, while attempting to minimize the global communication volume (or edge-cut).

In our formulation, the balanced sum of node feature size (Equation 1) and the balanced sum of node degree (Equation 2) in each partition are two constraints while we aim at minimizing the total communication volume (Equation 3).

Note that even though the global communication volume is minimized by the algorithm above, the communication cost of each partition cannot be balanced under the multi-constraint graph partitioning problem setting, as it is a result of the actual partitioning and hence cannot be predefined as a constraint.

Second, in the iterative re-partitioning stage, the algorithm swaps nodes between partitions to balance the communication volume in each partition. We formalize our optimization objective as follows:

$$\text{minimize } T_{max\_comm.} = \max_{V_p \in P} |V_p^{remote}| \tag{4}$$

The algorithm is shown in Algorithm 1. In each iteration, we first find two partitions that have the highest and lowest number of remote nodes, denoted as V_{max} and V_{min}. Then we consider the locality in each node by picking a node v_{max} in V_{max} which has the highest number of remote neighbors (*i.e.*, out-of-partition nodes in its neighborhood) and a node v_{min} in V_{min} which has the lowest number of remote neighbors if moved out of V_{min}, and then we swap these two nodes.

Algorithm 1: Locality-aware Iterative Partition

Data:
(1) Input Graph: G
(2) Number of Workers: n
(3) Multi-constraint Partition: $P^{Multi}(G, n) = \{V_1, .., V_n\}$
Result: Final Partition $P = \{V_1, .., V_n\}$

```
1  begin
2      P ⟵ P^Multi(G, n)
3      while true do
4          V_max ⟵ argmax_{V_p ∈ P} |V_p^remote|
5          V_min ⟵ argmin_{V_p ∈ P} |V_p^remote|
6          ratio ⟵ |V_min^remote| / |V_max^remote|
7          if ratio is converged then
8              return P
9          end
10         v_max ⟵ argmax_{v ∈ V_max} |v^remote|
11         v_min ⟵ argmin_{v ∈ V_min} |N(v) − V_max|
12         V_max ⟵ V_max − {v_max} + {v_min}
13         V_min ⟵ V_min − {v_min} + {v_max}
14     end
15 end
```

The above process repeats until the difference between $|V_{min}^{remote}|$ and $|V_{max}^{remote}|$ converges below a threshold γ (we use 0.5% in our experiment). We also stop the process if cyclic swapping.

Fast Neighbor Tracking Algorithm. Finding every partition's out-of-partition neighbors V_i^{remote} can be time-consuming (line 4–5) because the search of all nodes in V_i takes $O(|V|)$ time. We present an algorithm that enables incremental updates to track each node's out-of-partition neighbors. The algorithm reduces the time complexity to $O(d_v)$, where v is the node moving in or out.

Making incremental updates in V_i^{remote} is not straightforward. This is because moving a node out of V_i does not necessarily remove all its remote neighbors from V_i^{remote}, as those neighbors could be remote neighbors to other nodes in V_i. To solve this problem, we maintain the in-partition degrees d_v^{in} for every node $v \in V_i^{remote}$. For example, when a node v moves out from V_i, the in-partition degrees of v's remote neighbors, v^{remote}, will be decreased by 1, and the node will be removed from V_i^{remote} if its d_v^{in} reaches 0. The technique is used when implementing node swapping (line 12–13) between the two affected node sets, rather than triggering a global recount. Moreover, as the search procedures (line 4–5 and line 10–11) are independent without dependencies, they are parallelized on multiple CPU cores with shared memory.

5 MULTI-LEVEL PIPELINE SCHEDULING

In GNN hybrid parallelism, workers share intermediate results to meet computation dependencies. To achieve efficient cross-worker pipelines to overlap computation and communication, the computation sequence is crucial as it impacts worker waiting time. G3 proposes a multi-level pipeline scheduling algorithm that exploits opportunities for host-GPU communication, GPU computation, and network communication via inter- and intra-layer pipelines.

5.1 Bin Packing Mechanism

The multi-level pipeline scheduling algorithm utilizes a bin packing mechanism: In each layer, nodes are divided into smaller groups called *bins* and processed one at a time by a worker. Once a bin's computation is finished in one layer, the worker can send its output to other workers and start processing the next layer.

The lifespan of a bin has 5 stages in the forward pass, as shown in Figure 5. First, the scheduling algorithm decides on the allocation of nodes and generates a subgraph containing these nodes and their neighboring nodes (even if initially they are not in the bin). Next, the subgraph and its node embeddings will be transferred from host memory to GPU memory and used for GNN computation. Finally, the output will be transferred out from GPU and shared with remote workers when required.

The backward pass is similar to the forward pass, except that the inputs are gradients instead of node embeddings. We reuse the bin packing decision in the forward pass as the backward pass of a bin is performed in exactly the reverse direction of the forward. The scheduling decision made in the forward is a good indication of how the backward should be scheduled.

Analysis and Comparison. Utilizing the bin packing mechanism in GNN training allows for fine-grained division of the training process at the per-node level, providing opportunities for overlapping computation and communication. Furthermore, it helps avoid loading the entire graph onto the GPU, which can prevent GPU memory issues when the input graph is large (§7.2), as seen in other GNN systems [34, 40].

We note that HGL [13] and JasmineGraph [20] also utilize bin packing, but they pack bins at the subgraph level and process them in a FIFO order without any priority setting. In contrast, G3 packs bins at the node level and adapts to GPU memory capacity to improve utilization while avoiding running out of GPU memory. G3 also proactively schedules inter- and intra-layer pipelines based on computation priority (§5.2).

5.2 Pipeline Strategies

In GNN hybrid parallelism, cross-partition dependencies are handled by sharing intermediate results via peer-to-peer communication. Specifically, a node's embedding is sent to the workers where its remote neighbors are located. To optimize efficiency, the communication of a bin is overlapped with the computation in the next layer, either within the same worker (intra-layer pipeline) or on a remote worker (inter-layer pipeline), as shown in Figure 5.

(1) Inter-Layer Pipeline

We propose two designs that maximize overlapping opportunities in inter-layer pipelines.

Node Computation Priority. Since the total communication cost for a layer is fixed and distributed evenly among workers with our partitioning algorithm (in §4), we aim to begin data transmission as early as possible. Therefore, our scheduling algorithm (described in §5.3) prioritizes computation for nodes that have more remote neighbors outside of their partition, as these nodes generate more communication workload.

Eager Computation. From the perspective of the receiving worker, as soon as the worker completes computation for the current layer, it can begin GNN computation on nodes that are ready for the next layer. A node is considered ready when its remote neighbors' embeddings have all been received from other workers.

When the memory footprint of ready nodes exceeds a threshold, eager computation packs them together and begins processing. The threshold is set to half the size of the GPU memory. This threshold is consistent with the bin size in the intra-layer pipeline (§5.2) to create smooth pipelines.

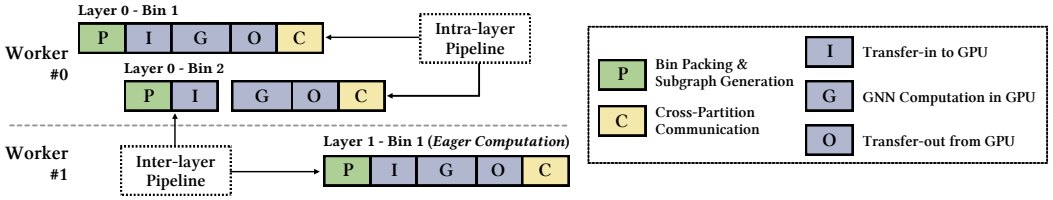


Fig. 5. An example of the intra-layer pipeline and inter-layer pipeline with two bins originated from worker #0.

The above optimization opportunity exists because our node priority setting ensures that other workers will begin transmitting output embeddings as soon as possible. This way, the receiving worker can begin processing some of the nodes in the next layer using the partial results received from other workers.

(2) Intra-layer Pipeline

Computing with GPUs requires loading data into GPU memory. Because GPU memory is relatively small compared to host memory, even when the input graph is partitioned across multiple workers, individual graph partitions may still be larger than the GPU memory of a worker. Therefore, as described in the bin packing mechanism, a layer's computation might be split into multiple bins and processed in the GPU sequentially (Figure 5).

In order to construct an effective pipeline that overlaps GPU computation and host-GPU communication, we utilize at most half the size of GPU memory for each bin so that one bin can be processed while another is loading in or out of the GPU. Additionally, the algorithm adapts bin sizes based on the memory requirements of each model layer to optimize efficiency. Since different models have different memory footprints, our algorithm adaptively adjusts bin sizes based on the model structure.

5.3 Scheduling Algorithm

We now introduce the multi-level pipeline scheduling algorithm that realizes the bin packing mechanism and pipeline strategies outlined above.

Step 1. Sorting Nodes. The worker uses BFS to traverse its assigned graph partition from any node in every connected component, sorting the nodes in the order of their visit. Then, the worker separates the nodes with remote neighbors from the rest and prioritizes them by placing them at the front of the list, creating the priority list $\mathcal{L}$. This step corresponds to the node computation priority setting in inter-layer pipelines.

Step 2. Calculating Bin Size Limit. We estimate the GPU memory required for one node in a GNN layer by using the layer's input and output embedding size. The bin size limit is then adaptively calculated for each layer by dividing half of the GPU memory size by the per-node memory usage. This ensures that each bin can be processed while another is loading in or out of the GPU without exceeding the GPU's memory limit. This step decides the bin size limit for intra-layer pipelines.

Step 3. Index-based Bin Packing. We iterate through the priority list $\mathcal{L}$ sequentially (index-based), adding ready nodes and their neighbors into a bin. Once a bin reaches its limit, we move on to a new bin and continue adding until all nodes in $\mathcal{L}$ are processed. This step schedules bins for intra-layer pipelines on each worker and realizes eager computation.

Index-based partitioning preserves locality when adjacent nodes are stored close to each other as [18, 52] shows, and our BFS-based sorting in the first step creates such locality, which increases the chance of packing nodes and their neighbors into the same bin and helps accelerate neighborhood aggregation in GNN training.

Analysis. The scheduling algorithm's **time complexity** is primarily determined by the BFS used for the computation priority of nodes, which takes $O(|E| + |V|)$, the same as initializing the graph data. In practice, GNN training begins on the GPU after the first bin is generated on the CPU. While the remaining bins are generated on the CPU, their costs do not affect overall performance as they are hidden under the GPU computation.

Proving the **correctness** of the algorithm is straightforward, as it only alters the computation order for the nodes without affecting the computation itself in GNN training. Our experiments on time-to-accuracy (§7.3) also confirm the correctness of the algorithm and our implementation.

6 IMPLEMENTATION

In this section, we give an architecture overview of G3 (§6.1), and present the implementation details of the G3 prototype (§6.2).

6.1 Overview

The workflow of G3 is shown in Figure 6. G3 takes a large graph as input and goes through two stages (*i.e.*, locality-aware iterative graph partitioning and multi-level pipeline scheduling). In particular, the first stage partitions the input graph iteratively in order to balance the workload according to the cost model. Then, the second stage trains the GNN using inter- and intra-layer pipeline scheduling to overlap communication with computation.

The graph partitioning stage is processed offline in the **scheduler**. For the online scheduling stage, the architecture of G3 takes advantage of the scheduler (*i.e.*, global and local task scheduler) to synchronize the training process. Specifically, each **worker** in the scheduling stage runs a local task scheduler to schedule its local task and an executor process for training tasks, while the worker with rank 0 runs a global scheduler that coordinates the model synchronizations across the cluster.

6.2 System Implementation

We implement G3 purely in Python. The functionalities in G3 include graph partition, task execution, and communication handlers between nodes in the graph. We build G3 on top of DGL for GNN operations and PyTorch for parameter synchronization, respectively.

Scheduler. The scheduler process takes charge of graph partitioning and online model coordination. Before training, the scheduler generates each partition's nodeID lists as described in Algorithm 1. The lists are then passed to DGL partition APIs to partition the graph into corresponding subgraphs, which contain all nodes listed and the edges destined for them. As described in §4.2, to accelerate the partition process, G3 parallelizes it through multiple CPU cores. During training, the global scheduler runs a coordination thread for layer-wise parameter synchronization. Note that G3 wraps and synchronizes each layer's parameters separately. Therefore, the coordination thread is responsible for monitoring the layer-wise BP programs in all workers, and notifying all workers to simultaneously update the corresponding layer parameters once all workers have finished the computation.

Workers. The worker process takes charge of P2P data sharing, online data-loading, and task execution. The worker first analyzes its subgraph and sets up send&recv hash tables for P2P data sharing. The hash tables contain dst/src workers with respect to their nodeIDs, which indicate the P2P inter-layer transmission paths. As each worker processes upon its dedicated partition throughout training, such tables remain unchanged and reside permanently in host memory for fast lookup. For the data-loading part, the worker achieves fast online bin packing by leveraging unused CPU resources (the same way as partition does) to judge ready nodes, and spawns data-loading threads for each bin to wrap the bin's nodes and their incident edges into `DGLBlock` format and then pushed into `TaskQ` for computation. The main process consumes tasks sequentially, and then

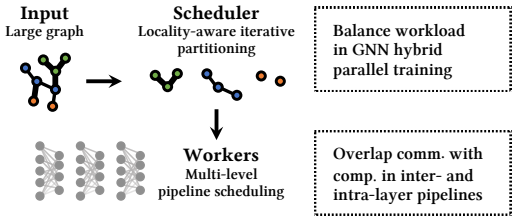


Fig. 6. Overview of G3 System

	Ogbn-products	Amazon	Ogbn-papers	Twitter-2010
Nodes	2.45M	1.60M	111.1M	41.65M
Edges	61.86M	132.2M	1.616B	2.405B
Features	100	200	128	128
Classes	47	107	172	100
Avg. Deg	50.5	82.7	29.1	57.7

Table 2. Graph datasets used in our evaluation.

splits the output result into pieces and spawns threads to send them out, referring to the generated P2P hash tables.

7 EVALUATION

In this section, we evaluate G3 with comprehensive testbed experiments. We first show the scalability of G3 by comparing it with mini-batch systems (§7.1), and its throughput performance compared to other full-graph systems (§7.2). Then we show G3's fast convergence speed and better accuracy compared to all baselines (§7.3). We also report the GPU utilization of each system (§7.4). Furthermore, we show the potential of G3 by exploring more complex GNN models and graphs in terms of layers and features (§7.5–§7.6). Finally, we demonstrate the effectiveness of G3's designs by comparing them with the state-of-the-art strategies (§7.7).

In summary, our key results reveal that:

- G3 achieves higher throughput than all baselines by up to 2.24× in a 16-node cluster, and as much as 5.5× better GPU utilization.
- G3 achieves up to 2.3× faster convergence speed and better final accuracy (~6% higher) compared to baselines.
- G3 is able to explore more complex GNN models and graph datasets. It achieves as much as 25.9× speedup with a 4-layer GNN model, and 1.94× speedup when training over graphs with 256-dimensional features compared to DGL.
- The microbenchmarks reveal that balanced partition and multi-level pipeline improve the training performance by up to 11.66×, and the benefits increase when scaling.

Experiment setup. We evaluate G3 using 8 physical servers (each with 2 RTX 3090 GPUs, 80 CPU cores (2.1GHz Intel Xeon Gold 5218R), 256GB RAM, and 2 Mellanox ConnectX5 NICs), and 4 Mellanox SN2100 switches. We divide one physical server into two docker containers, each with a 3090 GPU, 40 CPU cores, 128GB RAM, and a 10Gbps virtual Ethernet interface[1] to get a 16-node testbed. All nodes run 64-bit Ubuntu 18.04 with CUDA v11.1, DGL v0.6.1, and PyTorch v1.10.1.

Datasets. Table 2 lists four graph datasets that we used in our evaluation, including three popular GNN datasets: Ogbn-products [15], Amazon [48], and Ogbn-papers [15], and one graph dataset Twitter-2010 [3, 4]. We generate random features for Twitter-2010 ensuring that the ratio of labeled nodes remains consistent with what we observed in the OGB datasets, and only use the throughput results when training over it.

GNN models & Metrics. We use three representative GNN models, GraphSAGE [14], GCN [23], and GAT [33] for evaluation. By default, we adopt standard 2-layer GNNs with a hidden dimension of 16 in all experiments. For GAT, we use 8 attention heads. We test the performance of G3 on node

[1]We use SR-IOV to separate the resource of physical NIC. [9] shows that it achieves nearly the same performance as the non-virtualized environments.

classification tasks (*e.g.*, predicting the category of a product in the Ogbn-products dataset). We use *throughput* as our evaluation metric, which is the sum of all workers' average throughput.

Baseline. We compare G3 with two kinds of GNN training systems, *i.e.*, mini-batch system and full-graph system, for different evaluation purposes.

For mini-batch systems, we compare G3 with DGL [39, 50], a representative deep learning library for graphs, in terms of scalability and accuracy. We use DGL with PyTorch backend and conduct distributed training experiments. We use METIS, the default partition algorithm of DGL, in all DGL evaluations. We also compare with ClusterGCN [7], a method that does not synchronize features among workers during training, in terms of time-to-accuracy. Note that we do not compare with it upon throughput, as it has no feature communication throughout training and hence always achieves linear scalability, but at the cost of significant accuracy loss (see §7.3).

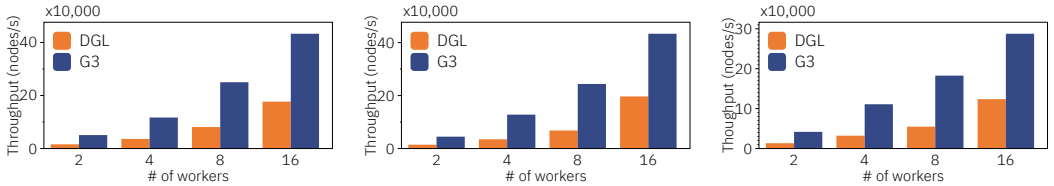
For full-graph systems, we compare G3 with BNS-GCN [34] and NeutronStar [40] in terms of throughput, accuracy, and GPU utilization. For system configurations, we maintain the default partition strategies for each system, *i.e.*, using METIS for BNS-GCN and a chunk-based approach for NeutronStar. We set the sampling ratio to 1 to ensure full-graph training in BNS-GCN. We implement GraphSAGE aggregators based on the existing GCN implementation in NeutronStar. We excluded these systems from our scalability experiments as they are not specifically designed for scalability. As previously reported in their original papers (Figure 5 in[34], Figure 12 in [40]), these systems show diminishing returns when increasing the number of workers. This is primarily due to large system overhead from cross-partition replication, as well as unbalanced workloads and underutilized overlapping opportunities, as discussed in §2.3. Additionally, these systems load the entire graph directly onto the GPU for parallel processing, making it infeasible to train GNNs on billion-edge graphs due to GPU memory constraints (Figure 11), especially when the cluster size is small.

Training pattern. In our experiments, G3, BNS-GCN, and NeutronStar perform full-graph training, while DGL performs sampling-based mini-batch training. For mini-batch training, we set the batch size of DGL to 1000 in all experiments. For a fair comparison upon throughput, we set our initial bin size (§5.2) equal to the batch size when comparing G3 with DGL. For neighborhood sampling in DGL, we adopt the sampling strategy (25, 10) [14] in all experiments. For full-graph training, we set G3's initial bin size to different values according to the input graph. For small graphs including Ogbn-products and Amazon, the initial bin size is set to 5000 to achieve high initial throughput like other full-graph systems, as they parallelize training by directly loading the whole graph into GPUs. When training over large graphs including Ogbn-papers and Twitter, we set G3's initial bin size to 1000 to avoid GPU memory explosion.

7.1 Scalability

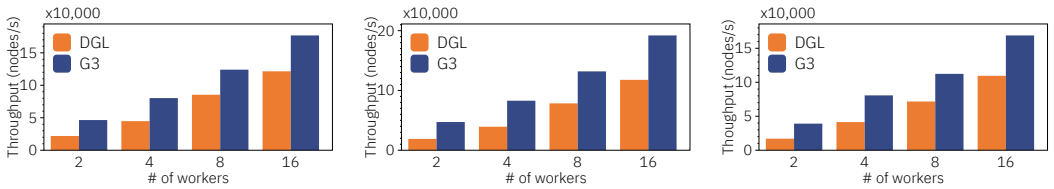
We demonstrate G3's scalability by comparing with DGL over four datasets in a cluster whose size ranges from 2 to 16. We train three models on each system to obtain the global throughput. The results are shown in Figure 7, 8, 9, and 10.

Different Systems. DGL shows poor throughput performance in all models with increasing cluster size. For example, when training over Ogbn-papers with the cluster size from 8 to 16, we only observe as much as 1.32× speed up compared to the double cluster size. This corresponds with the discussion in §2 that even with sampling strategies, the data parallelism scheme adopted by DGL suffers from duplicate computation and communication among workers. Compared to DGL, G3 achieves better scalability improvement almost in all cases. For example, G3 speeds up training by 1.64× and 1.74× on average over Ogbn-papers and Twitter-2010 when the cluster scales from 8 to 16, which is higher than those of DGL, *i.e.*, 1.32× and 1.44×. We remind the readers that training in large graphs is more challenging as their dependencies are more complex and would exaggerate



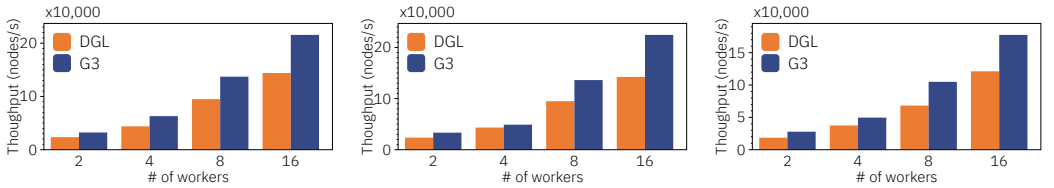
(a) GraphSAGE (b) GCN (c) GAT

Fig. 7. Training 3 GNN models over Ogbn-products. G3 is able to gain up to 3.66× higher throughput over DGL.



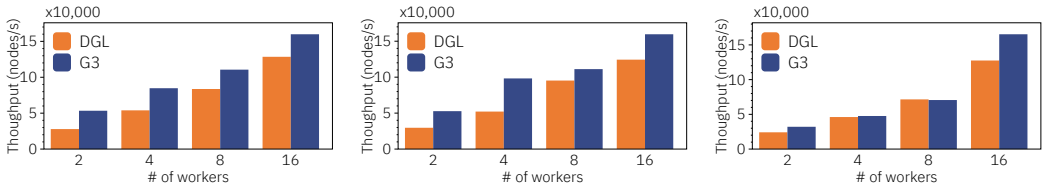
(a) GraphSAGE (b) GCN (c) GAT

Fig. 8. Training 3 GNN models over Amazon. G3 is able to gain up to 2.48× higher throughput over DGL.



(a) GraphSAGE (b) GCN (c) GAT

Fig. 9. Training 3 GNN models over Ogbn-papers. G3 is able to gain up to 1.47× higher throughput over DGL.

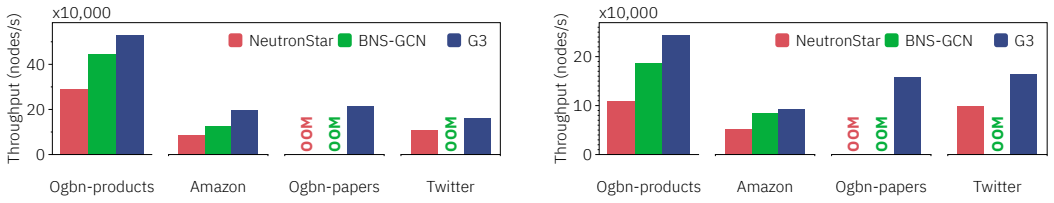


(a) GraphSAGE (b) GCN (c) GAT

Fig. 10. Training 3 GNN models over Twitter-2010. G3 is able to gain up to 1.91× higher throughput over DGL.

the scalability issue. Therefore the results above indicate better scalability of G3 than DGL. Overall, G3 outperforms DGL by 2.04×, 1.51×, 1.32×, and 1.13× in a 16-node cluster when training over four datasets, respectively.

Different GNN models. The training performance varies significantly across different GNN models. In a 16-node cluster, while G3 achieves higher performance improvement (1.24–2.45× speedup) with GraphSAGE and GCN models, it only speeds up GAT training process by 1.25–1.46×. In specific, G3 and DGL achieves almost the same throughput when the cluster size is small (from 2 to 8). This is because: 1) GAT model is more computation-intensive; 2) the improvement of communication is marginal when in small clusters. Hence, there is little room for G3 to optimize training. Moreover, GAT training involves extra intermediate attention tensors generation during forward propagation. The extra generated tensors increase the data volume transmitted between CPU and GPU, which therefore slows down the training process.



(a) GraphSAGE (b) GAT

Fig. 11. Full-graph training over 4 datasets. G3 is able to gain 1.08-2.24× higher throughput over baselines.

7.2 Comparison with Other Full-graph Systems

We compare with other full-graph systems, *i.e.*, BNS-GCN and NeutronStar, in terms of training throughput over four datasets in a 16-node cluster. We train GraphSAGE and GAT models in each system and obtain the global throughput.

The results are shown in Figure 11. G3 achieves 1.08-2.24× speedups over two baselines in all settings. NeutronStar performs worst in all settings because it adopts the chunk-based partition strategy that requires a significant communication workload. BNS-GCN performs better because METIS partition minimizes global edge-cut and thus incurs a low global communication workload. However, such strategy cannot balance communication for each partition, as described in §4.1, and hence may have straggler issues. Besides, both systems require loading every subgraph in each assigned worker for full-graph training. Such manner has two issues: 1) It fails to exploit the overlapping opportunities between computation and communication; 2) It results in an out-of-memory error when training over large graphs, as the size of the whole subgraph can easily exhaust the GPU memory in each worker. For G3, the balanced partition strategy it adopts achieves both low global communication and balanced communication for each partition, resulting in a balanced subgraph distribution for training. Moreover, the multi-level pipelining achieves: 1) fine-grained bin-level overlapping of communication with computation; 2) training over large graphs without out-of-memory error as it supports swapping bins between CPU and GPU to relieve resource constraints. Though such memory swapping comes with a throughput slowdown, we believe it is necessary for the tradeoff between memory constraint and training efficiency. Overall, G3 outperforms all baselines, and the mean speedups over BNS-GCN and NeutronStar are 1.38×/1.20× and 1.84×/1.89× for GraphSAGE/GAT training, respectively. G3 speeds up less with GAT because of the higher computation workload in GAT and extra memory swapping overhead, as illustrated in §7.1.

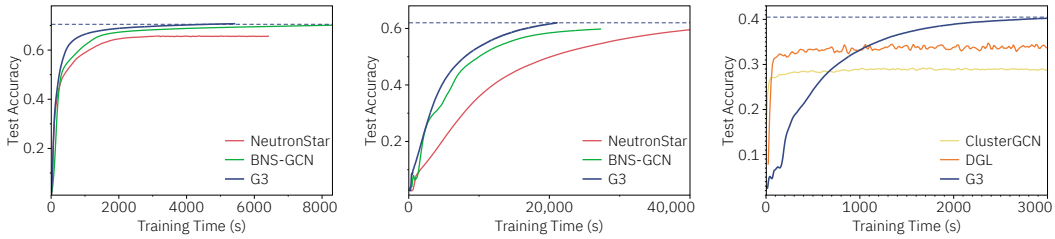
7.3 Time-to-Accuracy

In these experiments, we compare G3 with baselines in terms of time-to-accuracy. We conduct experiments using G3, BNS-GCN, and NeutronStar over Ogbn-products and Amazon, and using G3, DGL, and ClusterGCN over Ogbn-papers. We train GraphSAGE in all experiments in a 16-node cluster, and record the test accuracy during training.

The results are shown in Figure 12. The dashed lines are the approximate final accuracies that G3 achieves in each dataset, *i.e.*, 70.5% for Ogbn-products, 62% for Amazon, and 40.5% for Ogbn-papers, respectively.

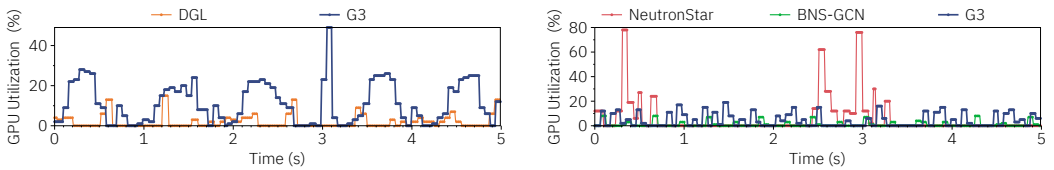
As Figure 12a and 12b for small graphs show, three full-graph training systems all converge to similar final accuracy. But G3 converges fastest among the three, *i.e.*, 1.8-2.3×, thanks to its higher training throughput, which can be attributed to the balanced partition and multi-level pipelining.

As Figure 12c for the large graph shows, initially, ClusterGCN converges the fastest among the three because it has no feature synchronization among workers, and hence has the minimal



(a) Ogbn-products (b) Amazon (c) Ogbn-papers

Fig. 12. Time-to-accuracy results over three datasets. G3 achieves 1.8-2.3× faster convergence speed than full-graph baselines, and 6%-10.7% higher final accuracy than mini-batch baselines.



(a) GPU utilization compared with mini-batch systems. The peak/average GPU utilizations are 49%/11.2% and 15%/2% for G3 and DGL, respectively.

(b) GPU utilization compared with other full-graph systems. The peak/average GPU utilizations are 19%/5.4%, 78%/4.8%, and 8%/1.1% for G3, NeutronStar, and BNS-GCN, respectively.

Fig. 13. G3 achieves 5.5×/2.9× higher average GPU utilization than mini-batch/full-graph baselines.

epoch time (~3.2 seconds). However, due to the lack of neighbor information from remote hosts during training, ClusterGCN can only converge to as much as 29.8%. Afterward, we see no accuracy improvement with more training epochs. DGL converges faster but can only converge to low final accuracy, *i.e.*, ~6% lower than the value G3 achieves. The faster convergence speed that DGL achieves is due to the property of mini-batch training. Such paradigm updates the model during every iteration, which is a higher update frequency than that of full-graph training that G3 adopts (*e.g.*, 75 vs. 1 in Ogbn-papers). As a result, the frequent update manner converges faster, but at the cost of significant final accuracy loss due to the sampling in training (§2.3). For G3, it initiates the slowest among all due to the property of full-graph training. But then it converges rapidly and finally achieves the best accuracy thanks to its high efficiency and full-graph training. Overall, G3 achieves the best final accuracy than DGL and ClusterGCN. The higher accuracy of G3 can be attributed to the full-graph training adopted by G3 than the mini-batch training adopted by DGL [18].

7.4 GPU Utilization

We report the GPU utilization of G3 compared with baselines when training GraphSAGE over Ogbn-products in a 16-node cluster. We measure the GPU utilization every 10 milliseconds. We use different initial bin size settings for G3 when compared to mini-batch and full-graph systems, as specified in §7.1 and §7.2. The results in a five-second window are shown in Figure 13.

As Figure 13a shows, we find that G3 achieves both higher peak GPU utilization (49% vs. 15%) and higher average GPU utilization (11.2% vs. 2%). Note that the small utilization value is due to the sparse computations in GNN models that fail to leverage GPU efficiency [11]. DGL achieves lower GPU utilization because its sampling becomes the bottleneck and limits the GPU utilization, as elaborated in §2.3. For G3, however, it remains high GPU utilization thanks to the balanced workload across workers and the multi-level pipelining to exploit overlap opportunities.

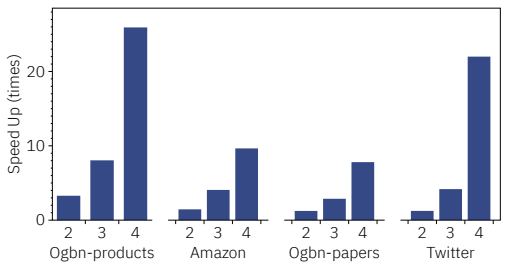


Fig. 14. The throughput advantage of G3 over DGL increases with more GNN layers.

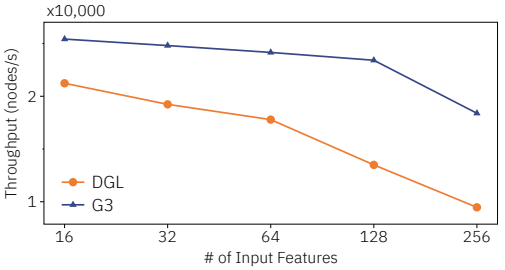


Fig. 15. G3 achieves better throughput performance when the number of features increases.

As seen in Figure 13b, NeutronStar has a higher peak GPU utilization compared to G3 and BNS-GCN, but also experiences the most GPU idle time. This is because NeutronStar loads the entire graph to GPU memory before processing, which leads to high GPU usage but more idle time (76.6% of the time) waiting for communication. Additionally, NeutronStar uses libtorch instead of DGL to implement GNN operators, which may also contribute to the difference in peak GPU utilization. BNS-GCN performs the worst peak and average GPU utilization due to the extra sampling steps during training. Overall, G3 has the best average GPU utilization among all systems, with 2.9× higher utilization on average than BNS-GCN and NeutronStar. As a result, G3 experiences almost no GPU idle time during the time span. This is due to the balanced workload that eliminates straggler issues and the multi-level pipelining that maximizes the overlap of communication and computation.

We also observe that with a smaller initial bin size setting, G3 achieves better GPU utilization. This is because, with a finer-grained bin packing, G3 can achieve better pipelining, but at the cost of more CPU-GPU memory swapping. We leave the optimal bin size configuration that balances the tradeoff between the fine-grained pipelining and minimal memory swapping between CPU and GPU as future work.

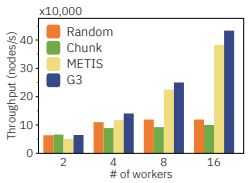
7.5 Impact of Layers

In this experiment, we compare G3 with DGL when training GraphSAGE with different numbers of layers over four datasets in a 16-node cluster. We create three variants for each GNN model with 2, 3, and 4 layers. The sampling strategies for DGL are (25, 10), (25, 15, 10), and (25, 20, 15, 10) for 2-, 3-, and 4-layer models, respectively. We report the throughput speedup of G3 against DGL.

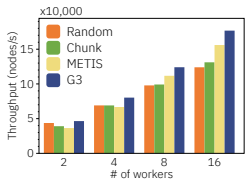
Figure 14 shows the results. We observe that the advantage of G3 over DGL increases drastically with the number of layers when training over all datasets. Specifically, G3 outperforms DGL by up to 25.9× in the 4-layer model. As elaborated in §2.3, DGL replicates all out-of-partition nodes for each mini-batch, whose size grows exponentially with the number of layers. Therefore, the efficiency of DGL degrades drastically with the increasing number of layers. By comparison, G3 alleviates duplicate computation and communication due to its hybrid parallelism design and thus shows a greater advantage over DGL with deeper GNNs. Specifically, G3 accelerates the training process over the four datasets by 25.9×, 9.6×, 7.8×, and 22.0×, respectively. The significant improvement of G3 than DGL demonstrates the potential of G3 to support exploration for deeper and more complex GNN models.

7.6 Impact of Input Features

In this experiment, we compare G3 and DGL using Amazon dataset with varying input feature dimensions in a 16-node cluster. We train GraphSAGE on Amazon and vary the input feature dimensions from 16 to 256.

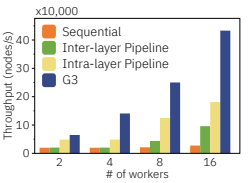


(a) Ogbn-products

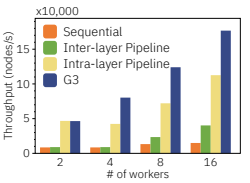


(b) Amazon

Fig. 16. Effectiveness of balanced partitioning.



(a) Ogbn-products



(b) Amazon

Fig. 17. Effectiveness of multi-level pipeline.

Figure 15 shows that G3 outperforms DGL in all dimensions and has a more graceful decrease in throughput as the feature dimension increases. DGL's heavy degradation in performance with increased feature dimension (*e.g.*, G3 outperforms DGL by 1.94× in a 256-dimensional graph) is due to the increased redundant communication, while G3 eliminates this redundant communication and uses multi-level pipeline scheduling to overlap computation and communication. We observe a slight decrease in G3's throughput when the input feature dimension increases from 128 to 256, possibly due to the large communication volume not being fully overlapped. Overall, the results demonstrate G3's versatility over a wider range of graph datasets.

7.7 Microbenchmarks

Balanced Work Partitioning. We evaluate the effectiveness of G3's locality-aware iterative partitioning by replacing it in G3 with the following partitioning methods.

- **Random**. Nodes are assigned to partitions at random.
- **Chunk-based** [52]. Nodes are split into contiguous chunks, with each chunk representing a partition.
- **METIS** [21]. A widely used graph partitioning library that generates partitions with minimum edge-cuts.
- **G3** with locality-aware iterative graph partitioning.

Figure 16 shows the training throughput using the above algorithms when the cluster scales out. Algorithms such as random and chunk-based partitioning perform poorly as they do not take into account all edges in the graphs. METIS and balanced partitioning, which aim to minimize the number of edges across subgraphs, perform better. Balanced partitioning, in particular, by balancing communication and computation workload, improves the straggler issue and is important for multi-level pipeline scheduling (as discussed in §3.2). Overall, balanced partitioning leads to a maximum 4.3× improvement in speed compared to other partition algorithms.

Multi-level Pipeline Scheduling. We evaluate the effectiveness of multi-level pipeline scheduling by performing experiments on the following variants of G3.

- **Sequential**. G3 with no inter- or intra-layer pipeline mechanism. All workers process computation and communication sequentially during training, which is the same manner as [13, 20]'s bin-packing performs.
- **Inter-layer**. G3 that pipelines only across layers (§5.2). Each worker employs the inter-layer pipeline but naively packs nodes with consecutive node IDs into bins.
- **Intra-layer**. G3 that pipelines only within the same layer (§5.2). Each worker waits for all required intermediate results and employs the intra-layer pipeline with adaptive bin packing.
- **G3** with the multi-level pipeline scheduling.

Figure 17 shows the training throughput of various G3 variants. The sequential method has the lowest throughput. Intra-layer and inter-layer methods both improve through partial pipelining,

with the intra-layer performing better, but this advantage decreases as the cluster size increases. This is because (1) the initial bin size of 11 allows for more opportunities for intra-layer pipelining, and (2) the inter-layer pipeline takes advantage of more network resources with a larger cluster size. The multi-level pipeline scheduling in G3 provides the best scalability and can achieve a maximum 11.66× improvement in performance speed.

8 RELATED WORK

GNN Frameworks. DGL [39, 50] supports distributed GNN training and is widely used in both academia and industry. ROC [18] proposes online learning-based graph partitioning for a balanced workload and uses dynamic programming for efficient memory management. NeuGraph [28] bridges graph processing with DNN and support efficient multi-GPU training. However, these frameworks suffer from either layer-wise communication barriers or cross-partition neighbor replications, as elaborated in §2.3. PyG [10], AliGraph [51], Euler [2], and AGL [49] adopts mini-batch training with sampling, resulting in sub-optimal final accuracy [18].

GNN Training Optimization. GNNAdvisor [41] presents optimizations such as workload management and GPU memory customizations. DGCL [5] proposes a communication planning algorithm to optimize communication during training. PaGraph [27] proposes a caching policy that reduces data movement between CPU and GPU for the frequently visited nodes. These works assume that graphs are stored in one machine. BNS-GCN [34] advocates full-graph training and proposes a simple yet effective sampling method to relieve the communication and memory overhead. PipeGCN [35] defers the communication to the next iteration's computation, which introduces staleness and results in a lower theoretical convergence speed. Sancus [31] is also aware of embedding staleness and uses historical embeddings with cache to avoid communications adaptively. SALIENT [19] optimizes mini-batch training with a fast sampling approach, shared-memory parallelization, and pipelining of batch transfer. Dorylus [32] adopts pipeline parallelism [17, 30] to maximize the resource utilization in the serverless scenario. The above works may still potentially compromise their final accuracies. SAR [29] is a CPU-only GNN training system that proposes a distributed sequential rematerialization scheme for efficient memory usage. It does not support GPU training.

9 CONCLUSION

This paper tackles the scalability challenge in distributed GNN training. We propose GNN hybrid parallelism in G3 to scale out GNN training with carefully scheduled peer-to-peer intermediate data sharing, enabling scalable GNN training on large graphs. G3 accelerates the training process by balancing workload across workers with locality-aware iterative partitioning, and overlaps communication with computation using a multi-level pipeline scheduling algorithm. We evaluate G3 with extensive experiments on large graphs, and the results demonstrate up to 2.24× improvement in training throughput and better final accuracy compared to previous systems in a 16-node cluster.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments. This work is supported in part by the Key-Area R&D Program of Guangdong Province (2021B0101400001), the Hong Kong RGC TRS T41-603/20-R, GRF-16213621, ITF ACCESS, the NSFC Grant 62062005, the NSFC grant 62172008, the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas), and the Turing AI Computing Cloud (TACC) [45]. Xin Jin is with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education. Kai Chen is the corresponding author.

REFERENCES

[1] Ravichandra Addanki, Peter W. Battaglia, David Budden, Andreea Deac, Jonathan Godwin, Thomas Keck, Wai Lok Sibon Li, Alvaro Sanchez-Gonzalez, Jacklynn Stott, Shantanu Thakoor, and Petar Velickovic. 2021. Large-scale graph representation learning with very deep GNNs and self-supervision. In *arXiv*.

[2] Alibaba. 2020. *Euler*. https://github.com/alibaba/euler

[3] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proc. WWW*.

[4] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. WWW*.

[5] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: An efficient communication library for distributed GNN training. In *Proc. EuroSys*.

[6] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. Fastgcn: fast learning with graph convolutional networks via importance sampling. In *arXiv*.

[7] 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 *Proc. SIGKDD*.

[8] Matthew T. Dearing and Xiaoyan Wang. 2021. Analyzing the Performance of Graph Neural Networks with Pipe Parallelism. In *arXiv*.

[9] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. 2012. High performance network virtualization with SR-IOV. *J. Parallel and Distrib. Comput.* (2012).

[10] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. In *arXiv*.

[11] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *Proc. OSDI*.

[12] Andrew Gibiansky. 2017. Bringing HPC techniques to deep learning. *Baidu Research Technology Report* (2017).

[13] Yuntao Gui, Yidi Wu, Han Yang, Tatiana Jin, Boyang Li, Qihui Zhou, James Cheng, and Fan Yu. 2022. HGL: accelerating heterogeneous GNN training with holistic representation and optimization. In *Proc. SC*.

[14] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proc. NeurIPS*.

[15] 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. In *arXiv*.

[16] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and Bridging the Gaps in Current GNN Performance Optimizations. In *Proc. PPoPP*.

[17] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *arXiv*.

[18] 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 *Proc. MLSys*.

[19] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E Leiserson, and Jie Chen. 2022. Accelerating training and inference of graph neural networks with fast sampling and pipelining. In *Proc. MLSys*.

[20] Anuradha Karunarathna, Dinika Senarath, Shalika Madhushanki, Chinthaka Weerakkody, Miyuru Dayarathna, Sanath Jayasena, and Toyotaro Suzumura. 2020. Scalable graph convolutional network based link prediction on a distributed graph database server. In *Proc. CLOUD*.

[21] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SISC* (1998).

[22] George Karypis and Vipin Kumar. 1998. Multilevel Algorithms for Multi-Constraint Graph Partitioning. In *Proc. SC*.

[23] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. (2016).

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Proc. NeurIPS*.

[25] Guohao Li, Matthias Müller, Bernard Ghanem, and Vladlen Koltun. 2021. Training graph neural networks with 1000 layers. In *ICML*.

[26] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *Proc. OSDI*.

[27] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN Training on Large Graphs via Computation-Aware Caching. In *Proc. SoCC*.

[28] Lingxiao 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 *Proc. ATC*.

[29] Hesham Mostafa. 2021. Sequential Aggregation and Rematerialization: Distributed Full-batch Training of Graph Neural Networks on Large Graphs. In *arXiv*.

[30] 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 *Proc. SOSP*.
[31] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. Sancus: sta le n ess-aware c omm u nication-avoiding full-graph decentralized training in large-scale graph neural networks. In *Proc. VLDB*.
[32] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. 2021. Dorylus: affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *Proc. OSDI*.
[33] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. (2017).
[34] 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 *Proc. MLSys*.
[35] 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 *arXiv*.
[36] Xinchen Wan, Kai Chen, and Yiming Zhang. 2022. DGS: Communication-Efficient Graph Sampling for Distributed GNN Training. In *Proc. ICNP*.
[37] Xinchen Wan, Hong Zhang, Hao Wang, Shuihai Hu, Junxue Zhang, and Kai Chen. 2020. Rat-resilient allreduce tree for distributed machine learning. In *Proc. APNet*.
[38] Hao Wang, Jingrong Chen, Xinchen Wan, Han Tian, Jiacheng Xia, Gaoxiong Zeng, Weiyan Wang, Kai Chen, Wei Bai, and Junchen Jiang. 2020. Domain-specific communication optimization for distributed DNN training. In *arXiv*.
[39] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2020. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. In *arXiv*.
[40] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. Neutronstar: distributed GNN training with hybrid dependency management. In *Proc. SIGMOD*.
[41] 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 *Proc. OSDI*.
[42] Yiding Wang, Decang Sun, Kai Chen, Fan Lai, and Mosharaf Chowdhury. 2022. Efficient dnn training with knowledge-guided layer freezing. In *arXiv*.
[43] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. 2019. Simplifying graph convolutional networks. In *ICML*.
[44] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *arXiv*.
[45] Kaiqiang Xu, Xinchen Wan, Hao Wang, Zhenghang Ren, Xudong Liao, Decang Sun, Chaoliang Zeng, and Kai Chen. 2021. Tacc: A full-stack cloud computing infrastructure for machine learning tasks. In *arXiv*.
[46] Liang Yao, Chengsheng Mao, and Yuan Luo. 2019. Graph convolutional networks for text classification. In *Proc. AAAI*.
[47] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proc. SIGKDD*.
[48] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. Graphsaint: Graph sampling based inductive learning method. In *arXiv*.
[49] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. 2020. AGL: A Scalable System for Industrial-Purpose Graph Machine Learning. In *Proc. VLDB*.
[50] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *arXiv*.
[51] 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. In *arXiv*.
[52] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proc. OSDI*.

Received October 2022; revised January 2023; accepted February 2023