



# Scaling Large-scale GNN Training to Thousands of Processors on CPU-based Supercomputers

**Chen Zhuang**

Institute of Science Tokyo  
Tokyo, Japan  
Riken Center for Computational  
Science  
Kobe, Japan  
chen.zhuang@riken.jp

**Lingqi Zhang**

RIKEN Center for Computational  
Science  
Kobe, Japan  
lingqi.zhang@riken.jp

**Du Wu**

Institute of Science Tokyo  
Tokyo, Japan  
RIKEN Center for Computational  
Science  
Kobe, Japan  
du.wu@riken.jp

**Peng Chen**

RIKEN Center for Computational  
Science  
Kobe, Japan  
peng.chen@riken.jp

**Jiajun Huang**

University of South Florida  
Tampa, Florida, USA  
jiajun.huang.cs@gmail.com

**Xin Liu**

National Institute of Advanced  
Industrial Science & Technology  
Tokyo, Japan  
xin.liu@aist.go.jp

**Rio Yokota**

Institute of Science Tokyo  
Tokyo, Japan  
rioyokota@rio.ssrc.iir.isct.ac.jp

**Nikoli Dryden**

Lawrence Livermore National  
Laboratory  
Livermore, USA  
dryden1@llnl.gov

**Toshio Endo**

Institute of Science Tokyo  
Tokyo, Japan  
endo@ssrc.iir.isct.ac.jp

**Satoshi Matsuoka**

RIKEN Center for Computational  
Science  
Kobe, Japan  
Institute of Science Tokyo  
Tokyo, Japan  
matsu@acm.org

**Mohamed Wahib**

RIKEN Center for Computational  
Science  
Kobe, Japan  
mohamed.attia@riken.jp

## Abstract

Graph Convolutional Networks (GCNs), particularly for large-scale graphs, are crucial across numerous domains. However, training distributed full-batch GCNs on large-scale graphs suffers from inefficient memory access patterns and high communication overhead. To address these challenges, we introduce SuperGCN, an efficient and scalable distributed GCN

training framework tailored for CPU-powered supercomputers. Our contributions are threefold: (1) we develop general and efficient aggregation operators designed for irregular memory access, (2) we propose a hierarchical aggregation scheme that reduces communication costs without altering the graph structure, and (3) we present a communication-aware quantization scheme to enhance performance. Experimental results demonstrate that SuperGCN achieves a speedup of up to 6× compared with the SoTA implementations, and scales to 1000s of HPC-grade CPUs on the largest publicly available datasets, without sacrificing model convergence and accuracy. Moreover, due to the effective strong scaling of SuperGCN, we outperform SoTA GPU-based and CPU-based distributed full-batch GCN training frameworks, in absolute performance, for large-scale graphs.

Corresponding author: Peng Chen, Mohamed Wahib.



This work is licensed under a Creative Commons Attribution 4.0 International License.

ICS '25, Salt Lake City, UT, USA

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1537-2/25/06

<https://doi.org/10.1145/3721145.3730422>

## CCS Concepts

• **Computing methodologies** → **Machine learning**; **Parallel computing methodologies**.

## Keywords

Graph Neural Network, Supercomputer, Distributed Training

### ACM Reference Format:

Chen Zhuang, Lingqi Zhang, Du Wu, Peng Chen, Jiajun Huang, Xin Liu, Rio Yokota, Nikoli Dryden, Toshio Endo, Satoshi Matsuoka, and Mohamed Wahib. 2025. Scaling Large-scale GNN Training to Thousands of Processors on CPU-based Supercomputers. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3721145.3730422>

## 1 Introduction

Graph Convolutional Networks (GCNs) have been widely applied across various domains such as social networks analysis [23], biology [34, 45, 65], and chemistry [49, 52], due to their ability to process graph-structured data. Notably, GCNs play a crucial role in the recent surge of AI for Science. For instance, some recent studies [3, 8, 42, 48] have focused on transforming unstructured and adaptive mesh-based domains in physics simulations into graphs, enabling the replacement of traditional iterative solvers with GCNs. These approaches achieve comparable accuracy to conventional iterative methods while significantly reducing computational costs. The widespread presence of such large-scale graphs in real-world applications has driven the need for distributed GCN frameworks capable of efficiently handling large-scale graphs on large-scale computing systems.

Handling large-scale graphs in distributed GCN frameworks typically follows one of two types of training strategies: mini-batch and full-batch. The mini-batch method is widely supported in various studies, such as [9, 14, 25, 64]. In mini-batch training, the original graph is sampled into small subgraphs, which might destroy the original graph structure information during the training process, leading to potential accuracy degradation [12, 33]. Hence, the distributed full-batch training strategy, if designed to be scalable, is preferred for large-scale graphs, due to its ability to retain the original graph structure information.

Distributed full-batch GCN training is similar to model parallelism in conventional Deep Neural Networks (DNNs), it treats the full graph as the entire training dataset, and partitions the graph into multiple subgraphs to assign them to different workers for computation. After partitioning the graph, the distributed full-batch parallelism scheme applied on the graph is broken down to two distinctive components: local computation at the worker level, and remote communication to communicate data between the subgraphs. This

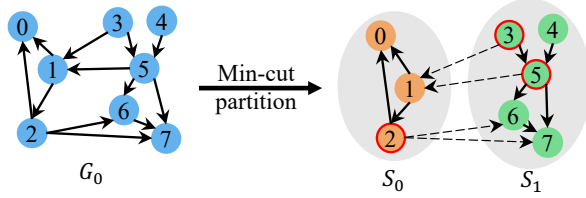
scheme presents two challenges for building an efficient and scalable GCNs training system: (1) Irregular memory access pattern and load imbalance caused by the sparsity and randomness of graphs. (2) Graph partitioning across workers creates numerous edge cuts between subgraphs, leading to significant imbalanced communication between different workers to preserve the original edge connections [57].

To address these two challenges of irregular memory access and imbalanced communication, clusters and supercomputers equipped with HPC-grade CPUs and high-performance interconnection networks represent a viable option over the more expensive and power-consuming GPU-accelerated systems. Therefore, it is extremely important to develop performant and scalable full-batch GCNs training solutions that are generic for different CPU platforms.

Currently, there are many existing distributed GCN training systems. For example, regarding the challenge of communication overhead, [57] reduces communication volume by sampling communication data, [44, 47, 53, 58, 66] use asynchronous communication to overlap with computation, and [56, 66] employ adaptive quantization to minimize communication costs. However, these approaches face several limitations: (1) they are not optimized for different CPU platforms, (2) sampling significantly modifies the graph structure, and leads to accuracy degradation similar to mini-batch training, (3) asynchronous communication breaks the dependency between epochs, introducing known issues such as data staleness and slower model convergence [15], (4) most importantly, none of the above approaches have been validated for extremely large graphs using distributed systems with a large number of machines.

In this paper, we design an efficient, scalable, and general distributed GCN training system for large-scale graphs on CPU-based supercomputers. The contributions of this paper are as follows:

- **General and Efficient Aggregation Operators.** We propose efficient CPU-level aggregation operators designed to support different CPU platforms (Arm and x86).
- **Hierarchical Aggregation Scheme:** We propose a novel hybrid pre-post aggregation method based on the minimum vertex cover algorithm, reducing communication volume while preserving the original graph structure.
- **Communication-Aware Quantization Scheme:** We leverage aggressive quantization communication to further reduce communication overhead. We additionally integrate masked label propagation and normalization to maintain model accuracy.
- Experimental results demonstrate that SuperGCN achieves up to 6× speedup compared to state-of-the-arts on CPU-based supercomputers, successfully trains on the largest



**Figure 1: Illustration of graph partitioning. Dotted lines indicate cut edges separating distinct subgraphs. Features of boundary nodes (red circles) need to be transferred across these cut edges via remote communication.**

publicly available GNN datasets using thousands of processors, and outperforms SoTA GPU-based and CPU-based distributed full-batch GCN training systems when operating at their respective maximum achievable performance.

## 2 Background

### 2.1 Graph Convolutional Networks

GCNs consist of multiple graph convolutional layers. GCNs take as input a graph and a feature matrix that stores the features of each node on the graph. The forward pass of GCN layers involves two stages: *Aggregation of neighbors' features* and *NN operations*. The stage of *Aggregation of neighbors' features*, represented as  $z_i^{(l-1)} = \text{AGGREGATE}(\{h_j^{(l-1)} \mid j \in N(i)\})$ , involves each node  $i$  on the graph aggregates the features of neighboring nodes  $N(i)$  using an *AGGREGATE* function (sum, mean, etc.). Following this is a neural network operation *UPDATE*, such as a linear transform, performed on node  $i$ 's  $l-1$ <sup>th</sup> layer features  $h_i^{(l-1)}$ , and aggregation result  $z_i^{(l-1)}$  to obtain the  $l$ <sup>th</sup> layer features  $h_i^{(l)}$ . This stage is referred to as *NN operations*,  $h_i^{(l)} = \text{UPDATE}(z_i^{(l-1)}, h_i^{(l-1)})$ . There are several GCN-related models, such as GCN [40] and GraphSAGE [25].

### 2.2 Distributed Full-batch GCNs Training

Distributed full-batch GCN training is crucial for addressing memory limitations and reducing training time. The input graph is divided into smaller subgraphs, each managed by a worker, similar to parallelism in traditional distributed neural networks [35]. Graph partitioning creates "cut edges" (dotted lines in Fig. 1), where the source and destination nodes belong to different subgraphs. These source nodes, known as *boundary nodes* (highlighted as red circles 2,3,5 in Fig. 1), require remote communication along cut edges to be transferred to the destination worker in aggregation of GCN layers. Several works have optimized distributed full-batch GCN training (Table 1). However, few studies target large-scale general CPU supercomputers (x86, Arm) and scale to thousands of processors.

### 2.3 Full-batch vs. Mini-batch GCNs

Mini-batch GCNs training samples nodes from the original graph to form smaller batches, on which training is performed [14, 25]. In contrast, full-batch GCNs training uses the entire graph as the training data. While Mini-batch training reduces memory usage and computation, it can result in loss of graph information. Studies [9, 14, 25] suggest that sampling-based methods do not compromise accuracy on certain benchmarks, but there is no theoretical proof of their effectiveness across different graphs. The influence of graph properties on sampling in mini-batch training is still not well understood. For instance, Chen et al. [11] highlight that the popular neighbor sampling method, *GraphSAGE*, lacks convergence guarantees due to biased predictions. Other studies [33, 53] show that mini-batch GCNs perform worse on the Reddit dataset [25]. Moreover, traditional sampling methods [60] often fail to retain key graph structures, causing information loss in large-scale graphs.

### 2.4 Stochastic Integer Quantization

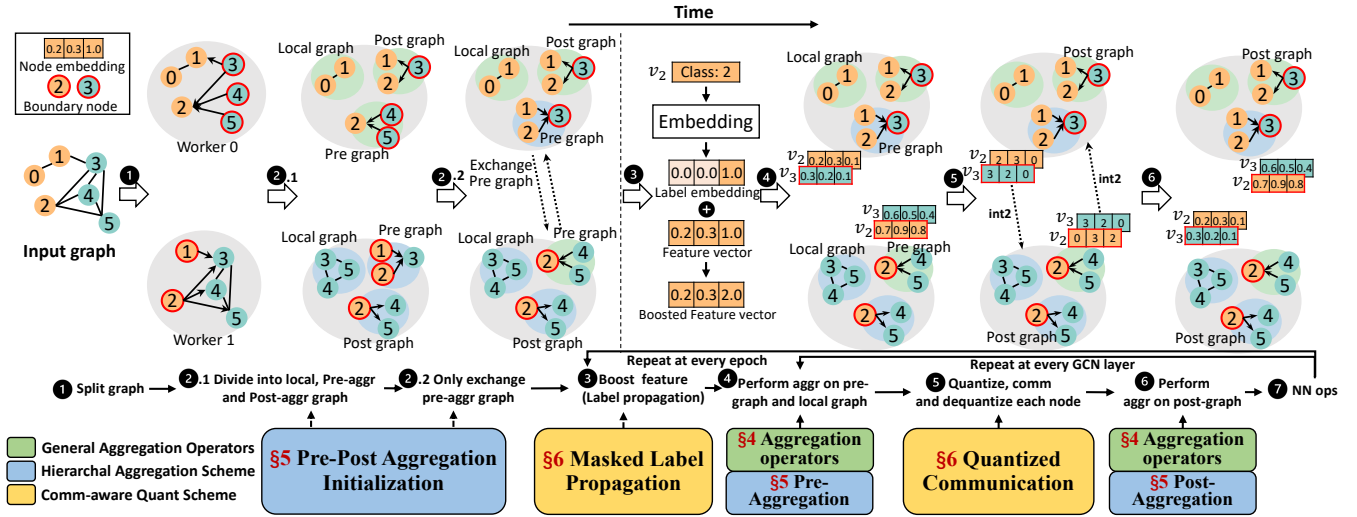
Stochastic integer quantization [10] is a fundamental technique in the field of machine learning. It is pivotal in optimizing various aspects of neural network training and communication efficiency [17, 56, 68]. This technique involves representing real-valued numbers as integers or real-valued numbers with reduced bit widths, thereby reducing memory footprint and accelerating computation. Quantization and dequantization methods can be written as  $h_{\text{quant}} = \text{round}((h - Z)/S)$  and  $h_{\text{dequant}} = h_{\text{quant}} * S + Z$ , where  $Z = \min(h)$ ,  $S = (\max(h) - \min(h)) / (2^b - 1)$ , and  $b$  is the bit width. The *round* function is the stochastic rounding operation [32].

### 2.5 GNNs with Masked Label Propagation

Label Propagation (LP) in Graph Neural Networks (GNNs) is a mechanism for distributing label information throughout a graph to assist in learning and prediction tasks. It improves the influence of nodes with the same label in the graph and improves model accuracy [59]. A specific type of label propagation in GNNs is Masked Label Propagation [51]. In GNN training with Masked Label Propagation, a portion of the labeled nodes' labels is randomly masked during each training step. The remaining unmasked labels are embedded into a vector that has the same dimension as the original feature vectors of nodes and added (sum) to the original feature vectors of nodes. With this step, in the message-passing operation (aggregation) of GNNs where the feature vectors are propagated in the graph, the unmasked label is also propagated in the graph. The masked labels not used in propagation are applied to compute loss and update model parameters. Masked Label Propagation works as label propagation but avoids label leakage and enhances the model's

**Table 1: Comparison of different distributed full-batch GNNs training solutions.**

Methods	Target Platform	Optimized operators	Preserve graph structure	Fresh boundary nodes	Reduce comm volume	Reported max workers number
ROC [33]	GPU	✗	✓	✓ (Synchronous)	✗	16
CAGNET [54]	GPU	✗	✓	✓ (Synchronous)	✓	100
BNS-GCN [57]	GPU	✗	✗	✓ (Synchronous)	✓	192
PipeGCN [58]	GPU	✗	✓	✗ (Asynchronous)	✗	32
Sancus [47]	GPU	✗	✓	✗ (Asynchronous)	✗	8
Dorylus [53]	Serverless	✗	✓	✗ (Asynchronous)	✗	32
DGCL [7]	GPU	✗	✓	✓ (Synchronous)	✗	8
AdapQ [56]	GPU	✗	✓	✓ (Synchronous)	✓	8
SAR [46]	x86 only	✗	✓	✓ (Synchronous)	✗	128
DistGNN [44]	x86 only	✓	✓	✗ (Asynchronous)	✗	256
<b>SuperGCN (ours)</b>	<b>x86 &amp; Arm</b>	✓	✓	✓ (Synchronous)	✓	<b>8,192</b>

**Figure 2: The flow of our proposed full-batch GCNs training system. Nodes with a red circle are boundary nodes obtained from other workers.**

ability to generalize [51]. Masked Label Propagation is implemented in popular GNN frameworks like PyG [18] and DGL [67], and it was demonstrated to boost model accuracy on some datasets (e.g. OGB dataset [4]).

### 3 SuperGCN

#### 3.1 Design Methodology

SuperGCN aims to minimize training time while maintaining high accuracy. SuperGCN consists of three components:

- **General and Efficient Aggregation Operators (Section 4):** Optimizes node-level performance.
- **Hierarchical Aggregation Scheme (Section 5):** Optimizes scalability, by reducing the communication volume, without affecting the algorithm or accuracy.
- **Communication-Aware Quantization Scheme (Section 6):** Employs aggressive quantization of communication to further improve scalability while preserving accuracy with algorithm design, as validated theoretically (Sec. 6.3) and empirically (Sec. 8.4).

#### 3.2 System Overview

Fig. 2 shows SuperGCN’s components and their interactions:

- 1 Partition input graph into subgraphs using METIS [38].
- 2 Split each subgraph into local, remote (pre-/post-aggregation) graphs based on communication reduction strategy (Sec. 5.2). Exchange pre-aggregation graph between workers.
- 3 Embed randomly selected nodes’ labels to their features (Sec. 6).
- 4 Aggregate boundary nodes’ features with pre-aggregation graph and local inner nodes’ features with the local graph using optimized operators (Sec. 4).
- 5 Quantize and transmit boundary nodes features (overlapped with local aggregation). Dequantize received features (Sec. 6).
- 6 Aggregate boundary nodes’ features with post-aggregation graph (Sec. 4).
- 7 Apply NN operations (e.g. linear transform). Repeat from step 4 for the next layer or 3 for the next epoch.

For message-passing-based GNN models such as GAT [55], GIN [61], and GraphSAGE [25], their main difference lies in how they calculate the weights of neighbors (for example,



GAT computes the weights through a self-attention mechanism). However, the core operation of these models remains neighbor aggregation (consisting of local aggregation and remote aggregation (communication)). SuperGCN primarily optimizes neighbor aggregation. Consequently, our system can be seamlessly applied to the distributed training of these message-passing-based GNN models.

## 4 General and Efficient Aggregation Operators

At the node-level, full-batch GCN training uses two aggregation operators: *Index\_add* and *SpMM*, which are similar to each other, to aggregate the neighbors' feature vectors. The following section uses *Index\_add* as an example.

In *Index\_add*, rows in *src* are added to *dst* at positions specified by *idx* (Fig. 3(a)). Unordered *idx*, incurred by the random connectivity of the graph, leads to a poor memory access pattern and locality. We improve *Index\_add*'s memory access pattern and locality:

- (1) *Clustering and sorting*: Sort *idx* and cluster *src* rows that aggregate to the same *dst* row (Fig. 3(b)). Though this might disrupt the access order of *src* rows, since each *src* row is still accessed sequentially and only once, the performance impact is minimal compared to improving *dst* reuse.
- (2) *Loop reordering*: Adjust memory access order on *src* to enable register-optimized kernel for data reuse on *dst*.
- (3) *Vector register-optimized kernel*: A shape-adaptive (aligned with cache line size) inner kernel with template-based code generation (Fig. 3(c)) to enhance memory access throughput and improve data reuse.

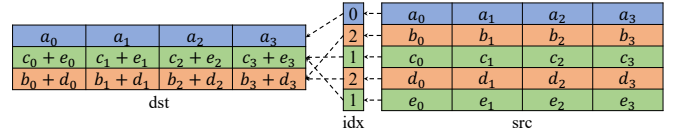
We also employ other optimizations including 2D dynamic parallelism and FLOPS-based load balancing (Fig. 3(d)) to boost performance further.

## 5 Hierarchical Aggregation Scheme

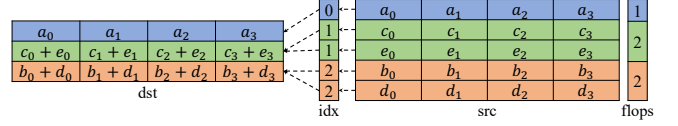
In distributed full-batch GCN training, communication performance is mainly affected by graph partitioning and the aggregation method. In this section, we first describe our selected graph partitioning algorithm (Sec. 5.1), followed by our novel *pre- and post-aggregation* method that reduces communication overhead based on partitioning results (Sec. 5.2). We then analyze the relationship between *pre- and post-aggregation* and minimum vertex cover problem (Sec. 5.3), and provide a performance model characterizing communication costs (Sec. 5.4).

### 5.1 Graph partitioning algorithm selection

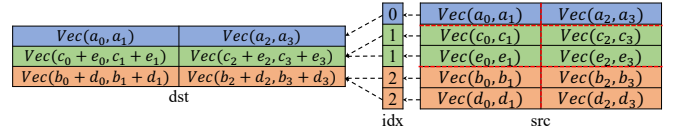
Regarding graph partitioning, we employ METIS [38] min-cut algorithm, which is widely adopted in popular GNN frameworks such as DGL [67], PyG [18], and so on [7, 43,



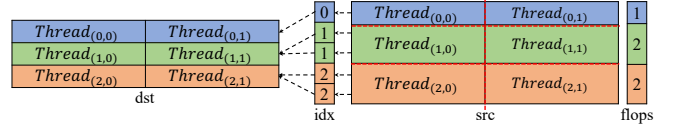
(a) Baseline vanilla implementation of *index\_add*.



(b) Cluster *src* based on sorted *idx* to improve memory access of *dst*.



(c) Invoking a kernel optimized for vector register reuse on *dst*.



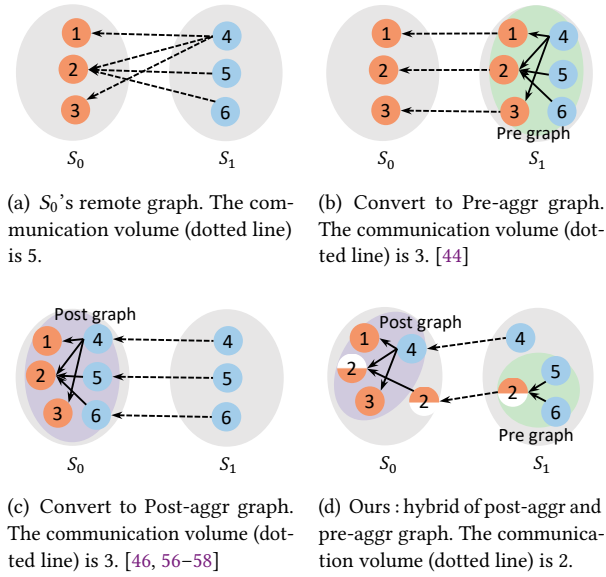
(d) A 2D dynamic parallelism strategy to determine the thread number on each dimension and improve load balance.

**Figure 3: Steps of the proposed algorithm for optimizing the neighbor aggregation operator *index\_add* on a single CPU.**

47, 56–58, 66]. METIS minimizes the total communication volume by minimizing the total number of cut edges. Additionally, it maximizes communication locality (neighboring subgraphs have higher communication volume). This locality property is beneficial for computing clusters and supercomputers where intra-node communication is significantly faster than inter-node communication.

### 5.2 Pre- and Post-aggregation

**5.2.1 Constructing Pre- and Post-aggregation Subgraph.** After graph partitioning, each worker is allocated a subgraph. Each subgraph consists of two components: a local graph for message passing on inner nodes, and a remote graph for message passing on boundary nodes received from other subgraphs through the cut edges. To reduce communication overhead, the remote graph can be transformed into either a post-aggregation or pre-aggregation graph. *Pre* denotes performing the aggregation operation before communication, while *post* refers to performing the aggregation operation after communication. For instance, consider  $S_0$ 's perspective of the remote graph shown in Fig. 4. In this scenario, the



**Figure 4: Strategies to construct a remote graph.**

communication volume along the cut edges (dotted line) is 5 nodes' features. By converting the remote graph to either the pre-aggr graph (Fig. 4(b)) or the post-aggr graph (Fig. 4(c)), the communication volume is reduced from 5 node features to 3 node features. To our knowledge, most existing solutions choose either the pre-aggregation method [44] or the post-aggregation method [46, 56–58].

However, the two above solutions are suboptimal. To achieve a more aggressive reduction in communication volume, we propose a simple yet effective method that categorizes each edge in the remote graph into either the pre-aggr graph or the post-aggr graph in Alg. 1. Initially, we treat the remote graph as a bipartite graph and identify all connected components in this graph. Subsequently, we introduce the minimum vertex cover algorithm to find the nodes that cover all cut edges (line 1-3 in Alg. 1). In Fig. 4(a), nodes 2 and 4 form the minimum vertex cover set. Then we traverse all edges in the remote graph (cut edges) (line 4-8 in Alg. 1). If the source node of the cut edges belongs to the minimum vertex cover (node 4 and its cut edges), it's better to delay aggregation on these edges until the source node reaches the destination worker, as performing aggregation first would involve the transfer of multiple destination nodes and cause redundant communication. Conversely, when the destination node of cut edges is part of the minimum vertex cover (node 2 and its cut edges), aggregation on these edges occurs prior to the transfer of results to the destination worker can reduce redundant communication (pre-aggr). Based on this strategy, a remote graph is transformed into a hybrid of pre-aggr graph and post-aggr graph, as shown in Fig. 4(d).

**Algorithm 1:** A method for transforming a remote graph into a hybrid of pre- post-aggregation graph.

**Input:** list of edges in remote graph  $r\_edges$   
**Output:** list of edges in pre-aggr graph  $pre\_edges$ , list of edges in post-aggr graph  $post\_edges$

```

1  $r\_edges \leftarrow \text{to\_bipartite}(r\_edges)$   $v\_cover \leftarrow \text{set}()$ 
2 for  $subgraph$  in  $\text{find\_connected\_components}(r\_edges)$  do
3    $v\_cover.add(\text{minimum\_vertex\_cover}(subgraph))$ 
4 for  $edge$  in  $r\_edges$  do
5   if  $edge.src$  in  $v\_cover$  then
6      $post\_edges.add(edge)$ 
7   else
8      $pre\_edges.add(edge)$ 

```

**5.2.2 Performing Pre- and Post-aggregation.** Our proposed approach involves the aggregation and communication stages of boundary nodes, including:

- (1) Perform pre-aggregation on node 5, 6 at  $S_1$  to obtain the partial result of node 2.
- (2) Send node 4 and the partial result of node 2 to  $S_0$ .
- (3) Conduct post-aggregation on received node 4 and the partial result of node 2 at  $S_0$  to get the aggregation result of node 1, 2, 3.

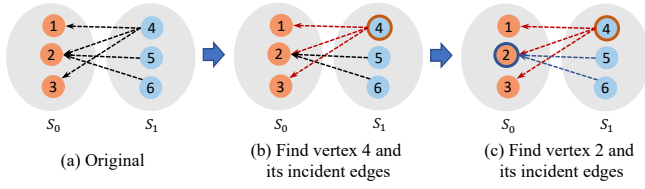
With our proposed strategy, the communication volume can be shrunk from 3 nodes to 2 nodes.

### 5.3 Constructing Pre- and Post-aggregation Subgraph is Finding Minimum Vertex Cover

To minimize communication overhead, we formulate the problem as a Minimum Vertex Cover (MVC) in a bipartite graph. This section explains how MVC ensures an optimal communication volume.

**5.3.1 Minimum Vertex Cover (MVC) in bipartite graphs.** In a bipartite graph  $G = (U, V, E)$ , a vertex cover is a subset of vertices  $C \subseteq U \cup V$  such that every edge in  $E$  has at least one endpoint in  $C$ . The **minimum vertex cover** is the smallest such set. König's theorem [41] establishes that in bipartite graphs, the size of a minimum vertex cover equals the size of a maximum matching, allowing polynomial-time solutions.

**5.3.2 Achieving Optimal Communication Volume via Minimum Vertex Cover.** Let the remote graph (Fig. 4(a)) induce a bipartite graph  $G = (U, V, E)$ , where each edge  $e = (u, v) \in E$  represents a data transfer between vertices  $u$  and  $v$ . If data were transferred along every edge, a vertex connected to multiple edges might be transmitted repeatedly, leading to redundancy. To eliminate this redundancy, we seek a set of vertices  $C \subseteq U \cup V$  such that every edge is "covered" by at



**Figure 5: Achieving optimal communication volume via finding minimum vertex cover.**

least one vertex in  $C$ ; mathematically, we require that

$$\forall(u, v) \in E, \quad u \in C \quad \text{or} \quad v \in C.$$

This set  $C$  effectively “absorbs” the data transfers for all its incident cut edges, meaning that transferring the data corresponding to the vertices in  $C$  is sufficient to account for all communications. Our goal is to minimize the communication volume, which is directly proportional to the number of vertices transmitted. Thus, we formulate the problem as

$$\min_{C \subseteq U \cup V} |C| \quad \text{subject to} \quad \forall(u, v) \in E, \quad u \in C \quad \text{or} \quad v \in C. \quad (1)$$

Take Fig. 5(a) as an example, the goal is to find a vertex set (vertex 2 and 4) that covers all edges while minimizing the number of vertices. Eqn 1 is exactly the definition of the MVC problem [16] on a bipartite graph. So we conclude that **finding the optimal communication volume is equivalent to solving the MVC problem on  $G$** . The optimal communication volume corresponds to  $|MVC|$ : the minimal number of vertices needed to cover all edges. Also, the MVC problem on bipartite graphs has an optimal solution (by König’s theorem [41] and Hopcroft–Karp algorithm [27]).

After identifying the minimum vertex cover set  $C$  (e.g. vertex 2 and 4 in Fig. 5(c)), we construct pre- and post-aggregation graph based on  $C$ , transforming the original edges communication into vertices communication on  $C$ , thereby achieving the optimal communication volume.

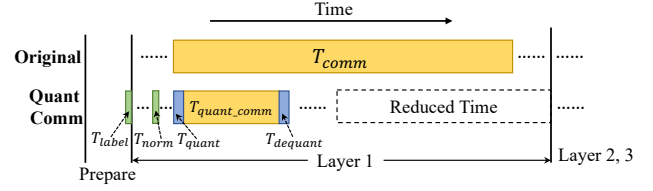
## 5.4 Communication Performance Model

After partitioning the graph and aggregating the nodes, the communication cost required in distributed full-batch GCN training can be estimated. We use  $Comm^{i,j}$  to represent the communication volume from process  $i$  to process  $j$ . For distributed training with  $P$  processes, the communication time  $T_{comm}$  is expressed as follows:

$$T_{comm}^{i,j} = \frac{Comm^{i,j} \times BIT_{fp32}}{BW_{comm}} + L_{comm} \quad (2)$$

$$T_{comm} = \max_{i=0}^P \left( \sum_{j=0}^P T_{comm}^{i,j} \right)$$

The upper equation computes the communication time as the sum of the data transfer time, given by  $\frac{V^{i,j}}{BW_{comm}}$ , and the fixed communication latency  $L_{comm}$ . Since the communication pattern in distributed full-batch GCN training is



**Figure 6: Traditional communication (top) and our communication quantization scheme (bottom) in GCN training.**

highly unbalanced, to account for this imbalance, the lower equation selects the communication time of the process with the longest communication time (serves as the bottleneck) as the global communication time  $T_{comm}$ .

## 6 Communication-Aware Quantization Scheme

To further reduce the communication volume, in this section we introduce a communication-aware quantization scheme that leverages quantization in the communication of distributed full-batch GCN training, while maintaining model accuracy. We illustrate the workflow of our scheme in Sec. 6.1. We then provide a thorough performance analysis in Sec. 6.2 and accuracy analysis in Sec. 6.3.

### 6.1 Workflow of quantization communication

The workflow of our scheme (as shown in Fig. 6) is

- (1) **Masked label propagation.** At the beginning of each epoch, we convert the labels of randomly selected nodes into embedding and add them to the initial feature vectors, which allows the labels to propagate during neighbor aggregation (③ in Fig. 2). Unselected labels are used to update model parameters.
- (2) **Normalization.** Large-magnitude outliers introduce large quantization errors. Therefore, we apply LayerNormalization on the embedding table before each GCN layer to remove outliers and smoothen the data distribution.
- (3) **Quantize, comm and dequantize.** Quantize the embedding table with quantization parameters (zero-point and scale). Transfer the quantized embedding table and quantization parameters to other workers. After communication, dequantize received embedding tables with the received quantization parameters. (⑤ in Fig. 2)

### 6.2 Performance analysis

In this section, we present the performance improvements achieved by our communication-aware quantization scheme through theoretical analysis. The key takeaway is that our

approach effectively reduces communication volume and enhances communication efficiency.

**6.2.1 Performance model of our scheme.** Quantization reduces the communication volume yet requires additional overhead to preserve model accuracy. Accounting for these overheads, we restructure our performance model (Eqn 2) into three components:

1)  $T_{pre\_quant}$ : Time for Masked label propagation at the start of each epoch and LayerNormalization before each GCN layer. These operations are performed on the local subgraph and do not increase the communication volume:

$$T_{pre\_quant}^i = \frac{SubGraph^i \times BIT_{fp32}}{TH_{cal}} \quad (3)$$

2)  $T_{quant}$  and  $T_{dequant}$ : Time for quantizing and dequantizing the communication data. We support quantizing the FP32 data into intX (where X can be 2, 4, or 8 bits). The communication volume can be reduced by a factor of  $\frac{BIT_{fp32}}{BIT_{intX}}$ , but this reduction introduces additional computational cost:

$$T_{quant}^{i,j} = T_{dequant}^{j,i} = \frac{Comm^{i,j} \times (BIT_{fp32} + BIT_{intX})}{TH_{cal}} \quad (4)$$

3)  $T_{quant\_comm}$  with Params: Quantization introduces quantization-related parameters (zero-point and scale), which are stored as FP32 values. These parameters need to be transferred to other workers for dequantization. Therefore, the time for quantized communication (transferring data and parameters) from process  $i$  to process  $j$  is updated as:

$$T_{quant\_comm}^{i,j} = \frac{(Comm^{i,j} \times BIT_{intX}) + (Params^{i,j} \times BIT_{fp32})}{BW_{comm}} + L_{comm} \quad (5)$$

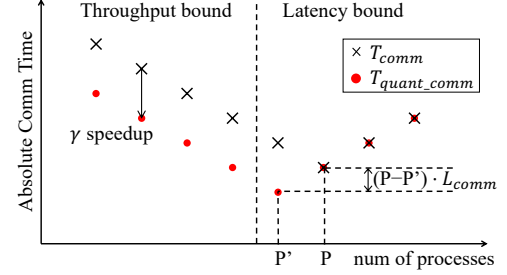
Taking all these components into account, the total quantized communication time is estimated to be:

$$T_{quant\_comm} = \max_{i=0}^P (T_{pre\_quant}^i + \sum_{j=0}^P (T_{quant}^{i,j} + T_{quant\_comm}^{i,j} + T_{dequant}^{j,i})) \quad (6)$$

**6.2.2 Performance gain of our scheme.** In this section, we analyze the performance gain of our scheme under different scales (bottlenecks in throughput or latency). To facilitate analysis, we define four ratios  $\alpha$ ,  $\beta$ ,  $\gamma$  and  $\delta$  where

$$\begin{aligned} \alpha &= \frac{Comm^{i,j}}{Params^{i,j}} \sim O(10^2) \\ \beta &= \frac{TH_{cal}}{BW_{comm}} \sim O(10^2) \\ \gamma &= \frac{BIT_{fp32}}{BIT_{intX}} = \frac{32}{X} \\ \delta &= L_{comm} : \frac{Comm^{i,j} \times BIT_{intX}}{BW_{comm}} \end{aligned} \quad (7)$$

where  $\alpha$  represents the ratio of communication data volume to quantization-related parameters volume,  $\beta$  is the ratio of computing throughput to communication bandwidth of the hardware,  $\gamma$  is the ratio of bit-width before and after quantization, and  $\delta$  is the ratio of communication latency to



**Figure 7: Performance analysis of quantized communication.**

data transfer time. The performance gain of our scheme can be expressed as follows:

$$Speedup = \frac{T_{comm}}{T_{quant\_comm}} = \frac{\alpha\beta(\gamma + \delta)}{(1 + \delta)\alpha\beta + 2\alpha(1 + \gamma) + \beta\gamma} \approx \frac{\gamma + \delta}{1 + \delta} \quad (8)$$

Note that  $T_{pre\_quant}$  is omitted in Eqn 8, as it is negligible compared to the communication time with processes  $P$ . **In the case of medium scales, communication is throughput-bound.** This implies that  $\delta \rightarrow 0$ . Our scheme reduces communication volume  $\gamma$  times and can achieve nearly  $\gamma$  speedup (left side in Fig. 7). For example, with Int2 quantized communication, the speedup approaches 16 $\times$ . **As the number of processes ( $P$ ) increases, the communication bottleneck gradually changes to latency.** This is reflected in the increase of  $\delta$  until  $\delta$  approaches infinity ( $\delta \rightarrow \infty$ ). According to Eqn 8, the performance improvement brought about by our scheme diminishes as the system increases the scale, yet it does not have any negative impact. Furthermore, our scheme can make the communication shift to latency bound with fewer processes ( $P'$ ), resulting in an absolute time reduction of  $(P - P') \cdot L_{comm}$  compared to the case without our scheme (as shown in Fig. 7).

### 6.3 Accuracy analysis

In this subsection, we analyze the methods we use from the perspective of model accuracy.

**Lemma 1.** With the GNNs model, given a global minimum of loss  $\mathcal{L}(W^*)$ , we make the following assumptions:

(1) the gradient  $\nabla \tilde{\mathcal{L}}(W_t)$  is  $\rho$ -Lipschitz continuous, (2) the approximate gradient is unbiased,  $\mathbb{E}[\nabla \tilde{\mathcal{L}}(W_t)] = \nabla \mathcal{L}(W_t)$ , (3) the variance of the approximate gradient is bounded,  $\text{Var}[\|\nabla \tilde{\mathcal{L}}(W_t) - \nabla \mathcal{L}(W_t)\|_2] = \mathbb{E}[\|\delta_t\|_2] \leq K$ ,  $K$  is a constant and  $K > 0$ .

Then select any  $t$  in epoch $\{1, \dots, T\}$ , we have

$$\mathbb{E}[\|\nabla \tilde{\mathcal{L}}(W_t)\|_2^2] \leq \frac{2(\mathcal{L}(W_1) - \mathcal{L}(W^*))}{T(2\eta - \rho\eta^2)} + \frac{\eta\rho}{2 - \eta\rho} \cdot K^2 \quad (9)$$

The proof of Lemma 1 follows the analysis in [56], adapted to our setting. The first term of the right side will go to 0 while  $T \rightarrow \infty$ . This means under the quantization scheme we



maintain  $O(\frac{1}{\gamma})$  convergence rate, and model will converge to the neighborhood of a stationary point, the radius of the neighborhood is determined by the gradient variance  $K$ .

**Lemma 2.** With the GCNs model, by embedding label into the vector space of feature, label propagation and feature propagation are approximately unified into message passing operations in GCNs. That is to say, we have:

$$\begin{aligned} H^{(l+1)} &= AH^{(l)}W^{(l)} = A^l(X + Y_{embed})(W^{(0)}W^{(1)} \dots W^{(l)}) \\ &= A^lX(W^{(0)}W^{(1)} \dots W^{(l)}) + A^lY_{embed}(W^{(0)}W^{(1)} \dots W^{(l)}) \end{aligned} \quad (10)$$

where  $H^{(l)}$  is the embedding after GCN layer  $l$ ,  $A$  is the adjacent matrix,  $W^l$  is the weight matrix in GCN layer  $l$ ,  $X$  is the initial feature vectors,  $Y_{embed}$  is the embedding from label, while  $Y_{embed} = YW_{embed}$ .

**Proposition 1.** Lemma 1 shows that with quantized communication, the model still maintains the same convergence rate. According to Lemma 2, adding embedded labels to features can be treated as a type of label propagation. Based on [51], integrating label propagation into feature propagation can enhance the connectivity between nodes with the same label. This enhanced connectivity brings the embeddings of nodes with the same label closer in the high-dimensional latent space, partially mitigating the issue where quantization shortens the distances between the embedding of nodes with different labels. This enables the model to more precisely separate nodes with different labels. Therefore, this enhances the model's tolerance to precision loss.

## 7 Implementation

Our framework is built on top of PyTorch and PyTorch Geometric (PyG) [18]. Currently, PyG lacks support for distributed full-batch GNNs training. Consequently, we implement the entire training process including preprocessing, graph partitioning, subgraph construction, and so on. For communication primitives, we employ MPI\_Alltoallv.

### 7.1 Single node level

Unlike typical CPUs that are optimized for latency, A64FX is a CPU optimized for throughput. It features a larger cache line size, higher memory bandwidth, more compute cores, wider SIMD instruction (Arm SVE 512), and longer operation latency. Therefore, our optimization goal is to provide more parallelism (e.g. aggressive software prefetch) to hide long latency. For x86 CPUs, we implement and optimize the aggregation operators with Intel AVX-512.

### 7.2 Distributed level

For graph partitioning, to ensure a balanced distribution of training samples and prevent load imbalances among different workers, we assign node weights based on in-degree of nodes and training masks when configuring METIS [38].

For the minimum vertex cover in pre- and post-aggregation, we optimize the implementation of the NetworkX library [24]. This significantly reduces the preprocessing time.

## 7.3 Quantization of Communication

In our implementation, we fix the bit-width of quantization to Int2. We implemented the quantization process based on [56] and made further optimizations:

(1) Decentralized int2 quantization communication scheme. In our proposed scheme, each worker quantizes and dequantizes the message without any synchronization or communication with a centralized master.

(2) Fusion of quantization parameters calculation and quantization kernels. To reduce redundant memory access, our fused kernel first retrieves 4 rows of the embedding table (necessary for packing four int2 values into one int8 for compatibility) and calculates quantization parameters for these rows. Then, it reuses data from cache and quantizes them using the newly calculated parameters.

(3) Latency reduction in quantization kernel. We improve performance by replacing the long latency division operation (98 cycles on Fujitsu A64FX [20]) with reciprocal estimate and multiplication operations and eliminating random number generation to shorten instruction dependency chains and improve throughput.

(4) Vectorization for quantization and dequantization kernel on different CPUs.

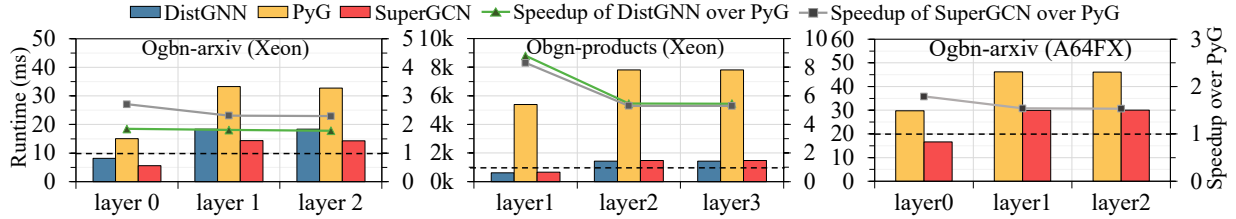
## 8 Evaluation

### 8.1 Experimental Setup

**Hardware.** We conducted experiments on two supercomputers: ABCI [1] and Fugaku [50]. ABCI's compute nodes are connected with InfiniBand and contain two Intel Xeon Gold 6148 CPUs, each with 20 cores. Fugaku's compute nodes are connected via the Tofu interconnection network and feature Fujitsu Arm A64FX processors. These processors have 48 cores, organized into 4 CMGs with each CMG containing 12 cores. **Datasets.** Table 2 shows the graph datasets in the experiments. The datasets are a) part of the "Open Graph Benchmark" suite [28, 29]; b) the widely used real dataset "Reddit" [25]; c) two large-scale graphs "Proteins" [2] and "UK-2007-05" [5, 6]; d) the latest and largest publicly available dataset "IGB260M" [39]. **Models.** We use a three layers GraphSAGE [25] model for the experiments. GraphSAGE is the most commonly used GCN model to assess the performance of GNN frameworks. The model settings for different datasets are shown in Table 2. **Baselines.** We choose the state-of-the-art system *DistGNN* as the baseline on ABCI. *DistGNN* is optimized for Intel clusters. It proposes delayed

**Table 2: The graph datasets and model hyperparameters used in experiments.**

Datasets	#Vertex	#Edges	#Feat	#Class	#Hidden	#Epochs	Dropout	Learning Rate	Norm Type
Ogbn-arxiv	169,343	1,166,243	128	40	256	250	0.5	0.01	LayerNorm
Reddit	232,965	114,615,892	602	41	256	250	0.5	0.01	LayerNorm
Ogbn-products	2,449,029	61,859,140	100	47	256	250	0.5	0.01	LayerNorm
Proteins	8,745,542	1,309,240,502	128	256	256	200	0.5	0.01	LayerNorm
Ogbn-papers100M <sup>1</sup>	111,059,956	1,615,685,872	128	172	256	200	0.5	0.005	LayerNorm
Ogb-lsc-mag240M <sup>2</sup>	121,751,666	2,593,241,212	768	153	256	300	0.5	0.005	LayerNorm
UK-2007-05	105,896,555	3,738,733,648	128	172	128	200	0.5	0.01	LayerNorm
IGB260M	269,346,174	3,995,777,033	1024	19	256	200	0.5	0.01	LayerNorm

<sup>1</sup>We convert this directed graph into an undirected graph on Fugaku.<sup>2</sup>We extract the undirected homogeneous papers citation graph**Figure 8: Performance comparisons of aggregation operators for different datasets on a single Intel Xeon Gold 6148 processor and one CMG of an Arm Fujitsu A64FX processor. A64FX processor is divided into four CMGs, each appearing as a single NUMA node.**

communication to reduce communication overhead. Following their paper’s suggestion, we set the delayed communication epoch number to 5 (cd-5). Since currently there is no proper baseline optimized for Arm processors, we report the performance of our framework on Fugaku as: (1) *SuperGCN* (w/o comm optimizations), which is our implementation before applying the proposed communication optimization, and (2) *SuperGCN* (w/ comm optimizations), which includes all the proposed communication optimization (pre-post aggregation and quantization communication). We also compare our methods with baselines optimized for full-batch distributed GCN training on GPU clusters. The results of these baselines are collected from their original papers.

## 8.2 Performance of Aggregation on a Single CPU

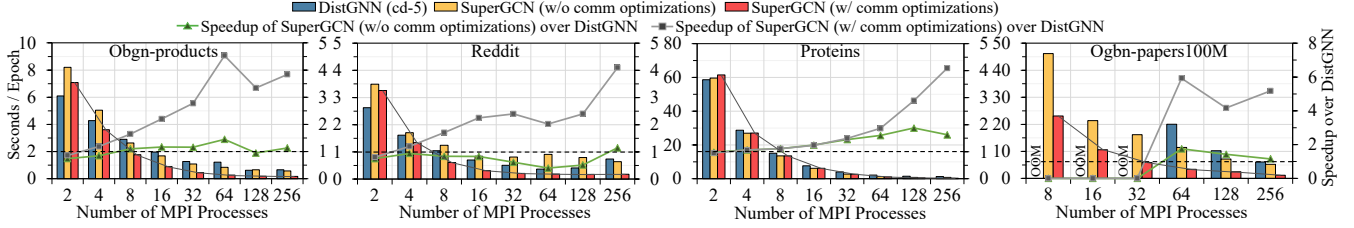
This section reports the performance of the proposed aggregation operators as these operators are the most time-consuming of full-batch GCN training on a single CPU. Fig. 8 shows the performance comparison of aggregation operators on a single Intel Xeon Gold 6148 processor. Due to the limitations of memory capacity, we only show results on a subset of the dataset. On the Xeon, SuperGCN achieves a speedup of 1.8× to 8.4× on different model layers over PyG, and ~1.22× over Intel’s library DistGNN. When processing larger datasets, such as ogbn-products, SuperGCN eliminates more redundant memory accesses. As a result, SuperGCN achieves higher speedup on larger datasets. We also compare SuperGCN with PyG on one CMG of an Arm Fujitsu A64FX

processor ( $\frac{1}{4}$  of the processor). We achieve a speedup ranging from 1.5× to 1.8× on the Ogbn-arxiv dataset.

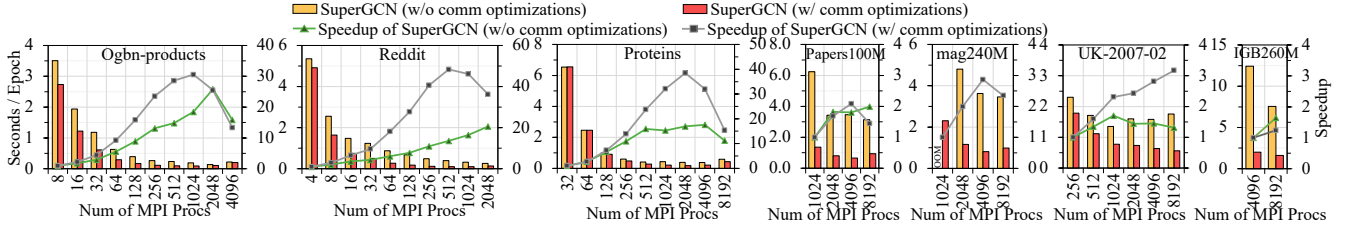
## 8.3 Performance and Scaling on Multiple CPUs

**ABCI supercomputer.** We conduct a performance and scaling comparison between SuperGCN and DistGNN on multiple CPUs using ABCI (Xeon CPUs) across various datasets. Fig. 9 illustrates the performance and speedup achieved over DistGNN on four different datasets. Due to the memory capacity limitation, we begin with 8 MPI processes for ogbn-papers100M. SuperGCN achieves speedups ranging from 0.9× to 6.0× compared to DistGNN. The results show that speedup becomes more significant as the number of processes increases, due to communication becoming the main bottleneck. These experimental results confirm the efficacy of our approach in reducing communication overhead. It is worth noting that those speedups are on the Xeon-based supercomputer, for which DistGNN is particularly designed.

The scaling of SuperGCN (w/o and w/ comm optimizations) and DistGNN on ABCI are shown in Fig. 9. DistGNN suffers from out-of-memory issues on Ogbn-papers100M. Across all datasets, SuperGCN (w/ comm optimizations) demonstrates superior strong scalability when compared with DistGNN. Notably, on Ogbn-papers100M, SuperGCN achieves a speedup that is close to linear scaling. On proteins, SuperGCN even obtains a superlinear speedup.



**Figure 9: Performance comparison with DistGNN and scaling for different datasets on ABCI (Intel).** OOM refers to the out-of-memory runs. The results in the bar chart correspond to the left y-axis (performance), while the results in the line chart correspond to the right y-axis (speedup).



**Figure 10: Performance and scaling for different datasets on Fugaku (Arm).** The results in the bar chart correspond to the left y-axis (performance), while the results in the curves correspond to the right y-axis (speedup). The speedup is over the smallest number of MPI processes the dataset can be executed on. Each A64FX in Fugaku runs four MPI ranks: the number of processors used in each run becomes the number of MPI ranks / 4.

**Fugaku supercomputer.** Fig. 10 shows the performance and scaling of SuperGCN on Fugaku. These results demonstrate the scalability of SuperGCN on Arm-based supercomputers. In summary, SuperGCN scales to a maximum of 8,192 MPI ranks, especially on the largest publicly available dataset IGB260M. *To the best of our knowledge, this stands as the highest scalability achieved by a full-batch GNNs training system.*

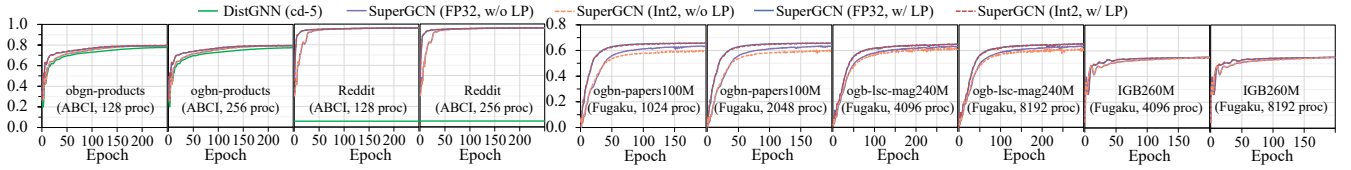
Furthermore, as shown in Fig. 10, compared to the performance at small-scale and large-scale, SuperGCN (w/ comm opt.) achieves a higher speedup over SuperGCN (w/o comm opt.) at medium-scale cases as communication is throughput-bound. In large-scale cases where communication becomes latency-bound, the speedup decreases. Nevertheless, the best performance of SuperGCN (w/ comm) still surpasses that of SuperGCN (w/o comm). These experimental results are consistent with the conclusion of our theoretical analysis presented in Sec. 6.2.2 and Fig. 7.

## 8.4 Accuracy Evaluations on Multiple CPUs

To assess the impact of our proposed communication-aware quantization scheme on model convergence and accuracy, we report the model convergence and accuracy on real large-scale datasets. We measure model convergence by monitoring changes in accuracy over training epochs, in all datasets. Additionally, we use the final test accuracy to report the

model accuracy. For Ogb-lsc-mag240M, we use the validation accuracy, since this dataset has no test label. The results can be divided into two parts: results of DistGNN, SuperGCN for Reddits and Ogbn-products on ABCI, and results of SuperGCN for Ogbn-papers100M, Ogb-lsc-mag240M, and IGB260M on Fugaku. For SuperGCN, We employ four settings to check the impact of Int2 quantization communication and masked label propagation on model accuracy: FP32 without masked label propagation (w/o LP), FP32 with masked label propagation (w/ LP), Int2 w/o LP, and Int2 w/ LP. We compared our final model accuracy after convergence with other baselines (shown in Table 4) and the OGB leaderboard [4] across various datasets, demonstrating that our model accuracy is within a reasonable range.

Accuracy for different datasets is shown in Fig. 11. In comparison to DistGNN, on Ogbn-products, even if our method converges at the same rate as DistGNN, our approach achieves higher accuracy on the test dataset after model convergence. On Reddit, there are some issues with the implementation of DistGNN, which result in abnormal training curves. On Ogbn-products, Reddit, and IGB260M, we found that Int2 closely matches FP32 regardless of whether label propagation is used. However, enabling masked label propagation allows the model to converge faster. On large-scale datasets Ogbn-papers100M and Ogb-mag240M, Int2 converges to lower accuracy than FP32 without masked label propagation, yet matches FP32 when label propagation



**Figure 11: Accuracy of test datasets (products, Reddit, papers100M and IGB260M) and validation dataset (Ogb-lsc-mag240M). W/o LP refers to training without masked label propagation, while w/ LP is to train with masked label propagation.**

**Table 3: Accuracy of different datasets. The N/A refers to the abnormal accuracy. SoTA accuracy of mini-batch (neighbor sampling) with the same model as SuperGCN is collected from Ogb-leaderboard [4] and [57] [37] [36] [39].**

Method	Oggn-products (ABCI)			Reddit (ABCI)			Oggn-papers100M (Fugaku)		Ogb-lsc-mag240M (Fugaku)		IGB260M (Fugaku)	
	64 procs	128 procs	256 procs	64 procs	128 procs	256 procs	1024 procs	2048 procs	4096 procs	8192 procs	4096 procs	8192 procs
DistGNN (cd-5)	78.16	77.76	77.27	N/A	N/A	N/A	-	-	-	-	-	-
SuperGCN (FP32, w/o LP)	79.14	79.15	79.17	96.14	96.12	96.16	63.58	63.62	63.33	63.33	54.90	54.89
SuperGCN (Int2, w/o LP)	79.42	79.24	79.33	96.10	96.17	96.16	60.19	60.42	62.73	60.91	<b>54.96</b>	<b>54.96</b>
SuperGCN (FP32, w/ LP)	79.64	<b>79.62</b>	<b>79.61</b>	<b>96.28</b>	<b>96.27</b>	96.23	65.62	<b>65.82</b>	<b>64.99</b>	<b>65.29</b>	54.39	54.37
SuperGCN (Int2, w/ LP)	<b>79.68</b>	<b>79.62</b>	79.26	96.26	<b>96.27</b>	<b>96.27</b>	<b>65.71</b>	65.70	64.87	65.23	54.51	54.93
Mini-batch (GraphSAGE)	78.70 [4]			95.40 [57]			64.91 [37]		65.10 [36]		54.95 [39]	

is enabled. This empirically validates that the masked label propagation we introduce mitigates the accuracy drop caused by Int2 communication on large-scale datasets.

Furthermore, Table 3 presents a comparison of test accuracy between DistGNN and the different settings of SuperGCN after model convergence. Similar to the result shown in Fig. 11, masked label propagation does not impact model accuracy for Oggn-products, Reddit, and IGB260M. However, on Oggn-papers100M and Ogb-mag240M, masked label propagation reduces the accuracy loss by Int2. Empirical results demonstrate that our communication-aware algorithm design maintains consistent training convergence and model accuracy.

**SuperGCN (full-batch) versus SoTA Mini-batch Training** In Table 3 we also report the highest reported mini-batch accuracy in literature using exactly the same model (GraphSAGE). SuperGCN consistently achieves higher (or similar) accuracy to SoTA mini-batch solutions.

## 8.5 Performance and Accuracy comparison with GPU baselines

In this section, we compare our framework on supercomputer Fugaku with the current state-of-the-art GPU baseline in terms of performance and model accuracy across different datasets. We collect the performance (seconds per epoch) and model accuracy for the highest achieved performance on the largest machine scale runs reported by the original papers. The GPU baselines are optimized for distributed full-batch GCN training, including AdaptQ [56] and SYLVIE [66] that employ adaptive quantization communication for improving framework’s performance and scalability. We increase our epoch number to align with their setting for this comparison.

The results are listed in Table 4. We can summarize the results: (1) Scaling to the highest performance allowed before performance regresses: our method (SuperGCN) achieves the best performance on the oggn-products and Reddit datasets while maintaining model accuracy within a normal range. (2) On the Oggn-papers100M dataset, converting the directed graph to undirected improves accuracy but increases the computation and communication costs. Our approach achieves high accuracy using an undirected graph while maintaining near-best performance. Unlike with other datasets, our experiments (Table 3) show lossy communication reduces accuracy on this dataset. However, our method preserves model accuracy even when using lossy communication techniques. (3) Our approach also demonstrates performance and model accuracy on the mag240M and IGB260M datasets. To the best of our knowledge, they are the largest publicly available GNN training datasets. All these results further demonstrate that, compared to state-of-the-art baselines, our framework achieves faster training while maintaining similar model accuracy and remains effective on large-scale datasets.

## 8.6 Runtime Breakdown on Multiple CPUs

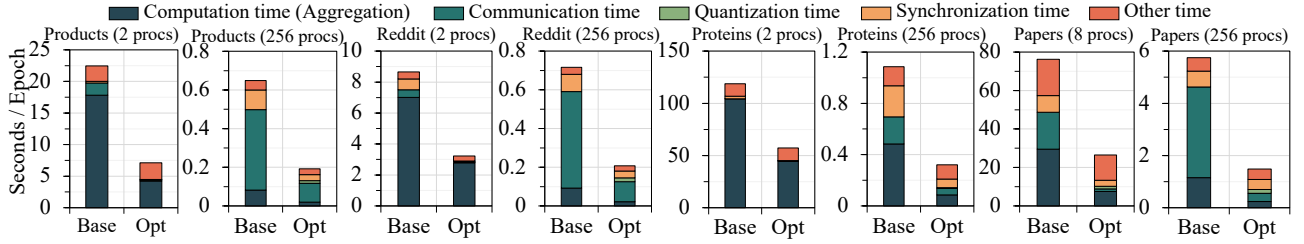
To understand performance bottlenecks when using SuperGCN for full-batch GCNs training, we collect the breakdown of training time before and after using our proposed optimization method on both small-scale and large-scale runs (shown in Fig. 12). We divide the training process into 5 components: (a) *Aggr time*, time of aggregation operation in GCN layers, (b) *Comm time*, time of communication in GCN layers, (c) *Quant time*, time of quantization and dequantization in GCN layers, (d) *Sync time*, time of synchronization in GCN layers (check load imbalance), (e) *Other time*, time spent on other



**Table 4: Performance and accuracy comparison of different GPU baselines on large-scale datasets. The results are for the highest achieved performance on the largest machine scale runs reported by the original papers. The size of datasets increases from left (Ogbn-products) to right (IGB260M).**

Methods	Platform	Ogbn-products		Reddit		Ogbn-papers100M		Ogb-lsc-mag240M		IGB260M	
		Time (s)	Acc. (%)	Time (s)	Acc. (%)	Time (s)	Acc. (%)	Time (s)	Acc. (%)	Time (s)	Acc. (%)
DGL [67]	GPU	0.99	79.19	7.28	97.1	17.0	–	–	–	–	–
PipeCCN [58]	GPU	0.43	78.77	0.43	97.10	6.70	–	–	–	–	–
BNSGCN [57]	GPU	0.28	79.30	0.19	<b>97.15</b>	<b>0.59</b>	–	–	–	–	–
AdapQ [56]	GPU	0.47	78.90	0.38	96.53	–	–	–	–	–	–
SYLVIE [66]	GPU	0.23	78.85	0.50	96.87	1.30	–	–	–	–	–
<b>SuperGCN (ours)</b>	CPU	<b>0.07</b>	<b>80.24</b>	<b>0.13</b>	96.55	0.65	65.63	<b>0.80</b>	<b>65.29</b>	<b>1.62</b>	<b>54.93</b>

Time (s): Epoch time in seconds    Acc.: Test accuracy (Val accuracy for mag240M)    The results of DGL are collected from [66].

**Figure 12: Time breakdown of different datasets on ABCI (Intel x86). *Base* represents vanilla PyG implementation, while *Opt* refers to SuperGCN with all our proposed optimizations.****Table 5: Communication volume and time in 1 GCN layer under different communication methods. This test is conducted for Ogb-lsc-mag240M dataset on Fugaku (2048 procs).**

Methods	Comm volume (GB)	Comm time (ms)
SuperGCN (pre_aggr)	1934.8559	1094.35
SuperGCN (post_aggr)	1934.8559	1131.62
SuperGCN (pre_post_aggr)	1269.5784	730.792
SuperGCN (pre_post_aggr+Int2) data	80.4770	47.0393
params	1.6530	4.82566

components of training. To collect precise results, we switch off the overlapping of computation and communication.

For small graphs, the performance bottleneck is the aggregation operation within the GCN layer for Ogbn-products, Reddit, and Proteins. Therefore, when using our proposed method designed for a single CPU, the time spent on the aggregation operation is significantly reduced, and the proportion of aggregation operation time in the total training time also decreases significantly. For large-scale runs, the performance bottleneck shifts to communication. After employing our proposed optimizations for reducing communication volume, the communication time drops significantly.

## 8.7 Efficacy of Communication Optimizations

In this section, we examine the effectiveness of our proposed optimizations on communication. Various configurations are evaluated, including (1) *Pre*: solely applying pre-aggregation,

(2) *Post*: solely applying post-aggregation, (3) *Pre-post*: applying the hybrid method of pre-aggr and post-aggr, (4) *Pre-post+Int2*: combining *Pre-post* with Int2 quantization. Here, *data* refers to the communication of quantized feature vectors, while *param* refers to the communication of parameters (zero point and scale) utilized for dequantization. Table 5 shows that our proposed hybrid of pre-aggr and post-aggr (pre-post-aggregation) method reduces communication volume and time by approximately 1.5× in comparison to *Pre* or *Post*. Despite the extra communication overhead introduced for dequantization, it still decreases the communication volume and time by about 15×. These results demonstrate the efficacy of our proposed optimizations on communication.

## 9 Related Work

There are several GCN training frameworks optimized for mini-batch training, such as Aligraph [62], P3 [22], DistDGL [67], etc. However, a detailed study [33] indicates sampling-based methods suffer from model accuracy loss.

On GPU platforms, several optimization methods have been presented for aggregation operators [19, 21, 31]. On CPU platforms, DistGNN [44] optimizes aggregation operators (SpMM) for Intel CPUs using Intel LibXSMM [26]. However, it primarily targets Intel x86 CPUs. In addition, there are approaches proposed for both CPUs and GPUs [30, 63] based on DL compilers (e.g. TVM [13]).

We categorize the communication optimization methods into three main approaches: (1) some methods [44, 47, 53, 58]

aim to mitigate the communication overhead by overlapping asynchronous communication with computation in the subsequent epoch. However, asynchronous communication introduces staleness for nodes' features, resulting in slower training convergence [15]. (2) optimizing communication paths based on network topology. DGCL [7] constructs a weighted graph to characterize the network topology and employs a tree-based algorithm to identify the optimal communication path. (3) reducing communication volume to mitigate the communication cost. CAGNET [54] employs a communication-avoiding algorithm to reduce communication volume. BNS-GCN [57] lowers communication volume by randomly sampling the boundary nodes to transfer. However, this approach modifies the graph structure. AdapQ [56], SYLVIE [66] and [69] introduce stochastic integer quantization [10] to compress boundary nodes' features. However, to obtain a trade-off between accuracy and performance, they have extra overhead to select an appropriate combination of 2, 4, and 8 bits for quantization, whereas our method applies a uniform 2-bit quantization communication, hereby reducing the communication volume to minimum, while also avoiding the high cost of adaptive quantization.

## 10 Conclusion

We present SuperGCN, a distributed training framework for graph convolutional networks designed specifically for CPU-based supercomputing systems. Our framework addresses the critical challenge of irregular memory access and communication overhead in distributed full-batch GCN training through three major contributions: general and efficient aggregation operators, a hierarchical aggregation scheme that reduces communication costs while preserving graph structure, and a communication-aware quantization scheme that effectively utilizes quantized communication while maintaining model accuracy. SuperGCN achieves a speedup of up to 6× in comparison to SoTA CPU-based implementations, scales to thousands of processors on the largest publicly available datasets, and outperforms SoTA GPU-based distributed full-batch GCN training frameworks at peak performance.

## Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-2005627). Nikoli Dryden was supported LLNL LDRD #24-SI-008. The authors wish to express their sincere gratitude to Tal Ben-Nun from LLNL for his helpful advice in this work.

## References

- [1] Ryousei Takano, Shinichiro Takizawa, Yusuke Tanimura, Hidemoto Nakada, Hirotaka Ogawa. 2024. *ABCI 3.0: Evolution of the Leading AI Infrastructure in Japan*. arXiv preprint [arXiv:2411.09134](https://arxiv.org/abs/2411.09134).
- [2] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos K Kyrpidis, and Aydin Buluç. 2018. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic acids research* 46, 6 (2018), e33–e33.
- [3] Filipe De Avila Belbute-Peres, Thomas Economou, and Zico Kolter. 2020. Combining differentiable PDE solvers and graph neural networks for fluid flow prediction. In *international conference on machine learning*. PMLR, 2402–2411.
- [4] Open Graph Benchmark. 2024. OGB-Leaderboards for Node Property Prediction. [https://ogb.stanford.edu/docs/leader\\_nodeprop/](https://ogb.stanford.edu/docs/leader_nodeprop/)
- [5] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.
- [6] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.
- [7] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: an efficient communication library for distributed GNN training. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 130–144.
- [8] Yadi Cao, Menglei Chai, Minchen Li, and Chenfanfu Jiang. 2023. Efficient learning of mesh-based physical simulation with bi-stride multi-scale graph neural network. In *International Conference on Machine Learning*. PMLR, 3541–3558.
- [9] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247* (2018).
- [10] Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael Mahoney, and Joseph Gonzalez. 2021. Actnn: Reducing training memory footprint via 2-bit activation compressed training. In *International Conference on Machine Learning*. PMLR, 1803–1813.
- [11] Jianfei Chen, Jun Zhu, and Le Song. 2017. Stochastic training of graph convolutional networks with variance reduction. *arXiv preprint arXiv:1710.10568* (2017).
- [12] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *International Conference on Machine Learning*. PMLR, 942–950.
- [13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.
- [14] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Chao-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.
- [15] Wei Dai, Yi Zhou, Nanqing Dong, Hao Zhang, and Eric P Xing. 2018. Toward understanding the impact of staleness in distributed machine learning. *arXiv preprint arXiv:1810.03264* (2018).
- [16] Reinhard Diestel. 2017. *Graph Theory* (5th ed.). Graduate Texts in Mathematics, Vol. 173. Springer.
- [17] Boyuan Feng, Yuke Wang, Xu Li, Shu Yang, Xueqiao Peng, and Yufei Ding. 2020. Sqquant: Squeezing the last bit on graph neural networks with specialized quantization. In *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 1044–1052.
- [18] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428*

- (2019).
- [19] Qiang Fu, Yuede Ji, and H Howie Huang. 2022. TLPINN: 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.
  - [20] Fujitsu. 2023. A64FX Microarchitecture Manual. [https://github.com/fujitsu/A64FX/blob/master/doc/A64FX\\_Microarchitecture\\_Manual\\_en\\_1.8.1.pdf](https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.8.1.pdf) Accessed: 2025-02-21.
  - [21] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.
  - [22] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale.. In *OSDI*. 551–568.
  - [23] Zhijiang Guo, Yan Zhang, and Wei Lu. 2019. Attention guided graph convolutional networks for relation extraction. *arXiv preprint arXiv:1906.07510* (2019).
  - [24] Aric Hagberg, Pieter J Swart, and Daniel A Schult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Laboratory (LANL), Los Alamos, NM (United States).
  - [25] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).
  - [26] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 981–991.
  - [27] John E Hopcroft and Richard M Karp. 1973. An  $n^5/2$  algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing* 2, 4 (1973), 225–231.
  - [28] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. Ogb-lsc: A large-scale challenge for machine learning on graphs. *arXiv preprint arXiv:2103.09430* (2021).
  - [29] 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.
  - [30] 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 *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.
  - [31] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. Gspmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.
  - [32] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2704–2713.
  - [33] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.
  - [34] Hao Jiang, Peng Cao, Mingyi Xu, Jinzhu Yang, and Osmar Zaiane. 2020. Hi-GCN: A hierarchical graph convolution network for graph embedding learning of brain network and brain disorders prediction. *Computers in Biology and Medicine* 127 (2020), 104096.
  - [35] Albert Njoroge Kahira, Truong Thao Nguyen, Leonardo Bautista-Gomez, Ryousei Takano, Rosa M. Badia, and Mohamed Wahib. 2021. An Oracle for Guiding Large-Scale Model/Hybrid Parallel Training of Convolutional Neural Networks. In *HPDC*. ACM, 161–173.
  - [36] 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. *Proceedings of Machine Learning and Systems* 5 (2023).
  - [37] 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. *Proceedings of Machine Learning and Systems* 4 (2022), 172–189.
  - [38] George Karypis, Kirk Schloegel, and Vipin Kumar. 1997. Parmetis: Parallel graph partitioning and sparse matrix ordering library. (1997).
  - [39] Arpandee Khatua, Vikram Sharma Mailthody, Bhagyashree Taleka, Tengfei Ma, Xiang Song, and Wen-mei Hwu. 2023. Igb: Addressing the gaps in labeling, features, heterogeneity, and size of public graph datasets for deep learning research. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4284–4295.
  - [40] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).
  - [41] Dénes König. 1931. Gráfok és mátrixok. *Matematikai és Fizikai Lapok* 38 (1931), 116–119.
  - [42] Remi Lam, Alvaro Sanchez-Gonzalez, Matthew Willson, Peter Wirsberger, Meire Fortunato, Ferran Alet, Suman Ravuri, Timo Ewalds, Zach Eaton-Rosen, Weihua Hu, et al. 2023. Learning skillful medium-range global weather forecasting. *Science* 382, 6677 (2023), 1416–1421.
  - [43] 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 *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 443–458.
  - [44] Vasmuddin 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.
  - [45] Jintao Meng, Peng Chen, Mohamed Wahib, Mingjun Yang, Liangzhen Zheng, Yanjie Wei, Shengzhong Feng, and Wei Liu. 2022. Boosting the predictive performance with aqueous solubility dataset curation. *Scientific Data* 9, 1 (2022), 71.
  - [46] Hesham Mostafa. 2022. Sequential aggregation and rematerialization: Distributed full-batch training of graph neural networks on large graphs. *Proceedings of Machine Learning and Systems* 4 (2022), 265–275.
  - [47] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. Sancus: stateless-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1937–1950.
  - [48] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W Battaglia. 2020. Learning mesh-based simulation with graph networks. *arXiv preprint arXiv:2010.03409* (2020).
  - [49] Seongok Ryu, Yongchan Kwon, and Woo Youn Kim. 2019. A Bayesian graph convolutional network for reliable prediction of molecular properties with uncertainty quantification. *Chemical science* 10, 36 (2019), 8438–8446.
  - [50] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki,

- Naoyuki Shida, Ikuo Miyoshi, et al. 2020. Co-design for a64fx many-core processor and "fugaku". In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.
- [51] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjin Wang, and Yu Sun. 2020. Masked label prediction: Unified message passing model for semi-supervised classification. *arXiv preprint arXiv:2009.03509* (2020).
- [52] Mengying Sun, Sendong Zhao, Coryandar Gilvary, Olivier Elemento, Jiayu Zhou, and Fei Wang. 2020. Graph convolutional networks for computational drug development and discovery. *Briefings in bioinformatics* 21, 3 (2020), 919–935.
- [53] 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 *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 495–514.
- [54] Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2020. Reducing communication in graph neural network training. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.
- [55] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. 2017. Graph attention networks. *stat* 1050, 20 (2017), 10–48550.
- [56] Borui Wan, Juntao Zhao, and Chuan Wu. 2023. Adaptive Message Quantization and Parallelization for Distributed Full-graph GNN Training. *Proceedings of Machine Learning and Systems* 5 (2023).
- [57] 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. *Proceedings of Machine Learning and Systems* 4 (2022), 673–693.
- [58] 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*. <https://openreview.net/forum?id=kSwqMH0zn1F>
- [59] Hongwei Wang and Jure Leskovec. 2020. Unifying graph convolutional neural networks and label propagation. *arXiv preprint arXiv:2002.06755* (2020).
- [60] Tianyi Wang, Yang Chen, Zengbin Zhang, Tianyin Xu, Long Jin, Pan Hui, Beixing Deng, and Xing Li. 2011. Understanding graph sampling algorithms for social network analysis. In *2011 31st international conference on distributed computing systems workshops*. IEEE, 123–128.
- [61] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).
- [62] Hongxia Yang. 2019. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 3165–3166.
- [63] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 660–678.
- [64] 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 *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 974–983.
- [65] Ye Yuan and Ziv Bar-Joseph. 2020. GCNG: graph convolutional networks for inferring gene interaction from spatial transcriptomics data. *Genome biology* 21, 1 (2020), 1–16.
- [66] Meng Zhang, Qinghao Hu, Cheng Wan, Haozhao Wang, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2024. Sylvie: 3d-adaptive and universal system for large-scale graph neural network training. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3823–3836.
- [67] 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 *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 36–44.
- [68] Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhe-long Li, Xiuqi Yang, and Junjie Yan. 2020. Towards unified int8 training for convolutional neural network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1969–1979.
- [69] Chen Zhuang, Peng Chen, Xin Liu, Toshio Endo, Satoshi Matsuoka, and Mohamed Wahib. 2024. Communication Optimization for Distributed GCN Training on ABCI Supercomputer. In *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*. IEEE, 160–161.