Scalable Graph Convolutional Network Training on Distributed-Memory Systems

Gunduz Vehbi Demirci*
Imagination Technologies
United Kingdom
gunduz.demirci@imgtec.com

Aparajita Haldar
University of Warwick
United Kingdom
aparajita.haldar@warwick.ac.uk

Hakan Ferhatosmanoglu†
University of Warwick
United Kingdom
hakan.f@warwick.ac.uk

ABSTRACT

Graph Convolutional Networks (GCNs) are extensively utilized for deep learning on graphs. The large data sizes of graphs and their vertex features make scalable training algorithms and distributed memory systems necessary. Since the convolution operation on graphs induces irregular memory access patterns, designing a memory- and communication-efficient parallel algorithm for GCN training poses unique challenges. We propose a highly parallel training algorithm that scales to large processor counts. In our solution, the large adjacency and vertex-feature matrices are partitioned among processors. We exploit the vertex-partitioning of the graph to use non-blocking point-to-point communication operations between processors for better scalability. To further minimize the parallelization overheads, we introduce a sparse matrix partitioning scheme based on a hypergraph partitioning model for full-batch training. We also propose a novel stochastic hypergraph model to encode the expected communication volume in mini-batch training. We show the merits of the hypergraph model, previously unexplored for GCN training, over the standard graph partitioning model which does not accurately encode the communication costs. Experiments performed on real-world graph datasets demonstrate that the proposed algorithms achieve considerable speedups over alternative solutions. The optimizations achieved on communication costs become even more pronounced at high scalability with many processors. The performance benefits are preserved in deeper GCNs having more layers as well as on billion-scale graphs.

PVLDB Reference Format:
Gunduz Vehbi Demirci, Aparajita Haldar, and Hakan Ferhatosmanoglu. Scalable Graph Convolutional Network Training on Distributed-Memory Systems. PVLDB, 16(4): 711 - 724, 2022.
doi:10.14778/3574245.3574256

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/gunduzvd/Scalable-Graph-Convolutional-Network-Training-on-Distributed-Memory-Systems.

*Previously at the University of Warwick. This publication describes work performed at the University of Warwick and is not associated with Imagination Technologies.
†Also with Amazon Web Services. This publication describes work performed at the University of Warwick and is not associated with Amazon.

Proceedings of the VLDB Endowment, Vol. 16, No. 4 ISSN 2150-8097.
doi:10.14778/3574245.3574256

1 INTRODUCTION

Graph Convolutional Networks (GCNs) generalize the convolution operation, performed by convolutional neural networks on structured data (e.g., images, time-series), to graphs [41, 50]. GCNs are used in a wide range of data intensive graph applications such as node classification [25, 40], traffic forecasting on road networks [64], and recommender systems on user-item graphs [63].

While graph-based learning models have been highly successful, the scale of large graphs, including their multi-dimensional features for the vertices, necessitates the use of distributed memory systems [22, 37, 54, 69]. During the feedforward and backpropagation phases in GCN training, the graph convolution operation involves message passing and aggregation steps that induce irregular data accesses due to complex graph inter-connectivity. Existing systems use graph partitioning algorithms designed for traditional graph algorithm workloads (e.g., connected components, shortest paths), which do not take complex GCN data access patterns into consideration. Therefore, intelligent message passing strategies need to be employed to achieve a communication-efficient distributed-memory parallel inference and training solution.

Sparse-dense matrix multiplication (SpMM) and dense matrix multiplication (DMM) are core kernel operations in GCN training. SpMM achieves convolution whereas DMM corresponds to propagating vertex-feature vectors through a single layer neural network. There have been improved solutions proposed for parallel SpMM [28, 32, 47] and DMM [2] problems. However, the special requirements of combining SpMM and DMM for scalable GCN training and ensuring efficient forward propagation and backpropagation phases in GCNs remain under-explored. In particular, communications incur high latency and bandwidth costs to aggregate feature matrices during feedforward as well as to aggregate gradients and update parameter matrices during backpropagation.

While recent parallel/distributed algorithms achieve GCN training for GPU clusters and cloud systems [54, 69], these typically perform broadcast- and allreduce-type of collective communication operations. Sparse data communication and compression methods have been considered to alleviate the scalability issues of allreduce for larger models and processor counts [11, 14, 30, 33]. However, such redundant data and message transfer causes unnecessary communication overheads. Moreover, in GCN training, model parameter matrices are significantly smaller than the adjacency and vertex-feature matrices, so performance improvements in allreduce communication are not significant in the overall parallel execution time. Instead, efficient parallelization of SpMM performed on the large graph data can lead to higher performance increase. For example, Sancus [43] is a recent model that adaptively avoids broadcast

communications to reduce network traffic in data-parallel GNNs. However, parallel SpMM still requires broadcast-type communication, which is the main performance bottleneck in GCN training, due to its high memory and bandwidth costs. A parallel algorithm, CAGNET [54], performs broadcasts among processors turn-wise to transfer vertex-features in small portions but suffers significant latency overheads. Therefore, a viable alternative is to design a mechanism that can utilize non-blocking point-to-point communications that move only the necessary data among processors.

We introduce a highly parallel algorithm for training GCNs on distributed-memory systems. Our solution achieves scalability by replacing the blocking broadcast communications in existing approaches with non-blocking point-to-point communications for parallel SpMM and transferring only the necessary data with minimal number of messages between processors. The solution employs a one-dimensional (1D) partitioning on large adjacency, vertex-feature, and gradient matrices for parallel SpMM computations in feedforward and backpropagation phases. It replicates parameter matrices across processors due to their relatively smaller sizes. This enables data locality for performing DMM computations without any communication. Allreduce communication is needed for aggregating gradients, which has a negligible cost compared to the communication costs incurred in parallel SpMM.

The use of point-to-point communication operations in our solution enables communication to be reduced further via sparse matrix partitioning strategies [7]. We develop a sparse matrix partitioning scheme to distribute the adjacency, vertex-feature, and gradient matrices used in computations among processors, based on a hypergraph partitioning model for the original graph. We show that the hypergraph partitioning model encodes SpMM communication costs more accurately than the graph partitioning model which is in popular use (e.g., in DistDGL [69]). To capture the randomness when communication operations are performed for mini-batch training instead of full-batch training, we also introduce a novel stochastic hypergraph model. This model encodes the *expected* communication volume in parallel mini-batch training and can be utilized for any mini-batch sampling strategy.

We focus on large-scale CPU clusters, commonly used for big sparse problem instances in scientific computing, since research towards adapting existing, relatively inexpensive supercomputing systems towards deep learning is gaining attention. We also demonstrate our proposed solution on GPU clusters by replacing local computations with GPU kernels and using NCCL [1].

We perform extensive experiments on real-world network datasets. Experimental results show that our solution is highly efficient and scales to large processor counts. We show that the proposed distributed solution achieves considerable speedups over the single-node GCN implementation in Deep Graph Library (DGL) [56], especially on large graphs having low average node degrees. Our hypergraph model generally outperforms the graph model as it correctly encodes the tasks and data dependencies by exploiting sparsity in connectivity patterns. The time spent on communication operations decreases with the increasing number of processors. This provides a scalability advantage over current alternatives that use inefficient collective communications. Using the novel stochastic hypergraph partitioning algorithm, we achieve further reductions in the communication volume for mini-batch training.

A summary list of contributions of this paper is given below:

- We propose a highly parallel GCN training algorithm that exploits sparsity in communications and data locality in computations to scale the training process.
- We show the merits of our hypergraph-based data partitioning scheme over the more popular graph-based approach for distributed full-batch training of GCNs, to reduce communication overheads while satisfying load balance.
- We propose a novel stochastic hypergraph partitioning model which can be utilized in parallel mini-batch training.
- On a set of real-world graph datasets (e.g., citation graphs, social networks, road networks, co-purchasing networks), we evaluate the performance of the proposed algorithm and data partitioning models, and provide further insights for scalable data processing and training for GCNs.

2 RELATED WORK

2.1 Distributed Graph Processing

Distributed systems have been widely employed for graph analytics, from parallel processing to streaming graph updates and cloud-based graph engines [17, 23, 27, 38, 49, 51]. Several graph analytics APIs, such as GraphX [59], are built atop Apache Spark or similar frameworks. These systems face CPU utilization bottlenecks that can be avoided with our data parallelization solution, enabling better scaling. In contrast to vertex-centric models, which suffer high communication overheads [18, 36], graph- and block-centric models utilize local graph partition structure to reduce communication and scheduling [53, 60]. Our parallel solution instead exploits sparse connectivity patterns to achieve better data locality, enabling efficient communication across thousands of compute nodes.

Graph partitioning is widely employed for improving the efficiency of different types of queries [13, 45, 67], handling skewed workloads [62], reducing communication overheads [16], and scalability in network bound applications [9]. Methods that adaptively determine partitioning strategies at run time [12] or are application-driven [13] also motivate the need for our solution that employs considerations specific to GCNs during partitioning stage.

2.2 Distributed Systems for GNNs

Graph learning tasks perform both forward and backward propagation of model parameters, involving k-hop neighborhood aggregations, which require different considerations in partitioning compared to that of traditional graph processing. To efficiently train GCNs, methods have been devised to restrict the neighborhood considered by sampling, pruning, and caching [8, 39, 71].

Memory management and distributed training are essential for scalable deep neural networks [3, 11, 57, 66]. Various frameworks use distributed memory systems for parallel Graph Neural Network (GNN) training [29, 72]. On GPUs, NeuGraph [37] uses dataflow scheduling while ROC [22] utilizes dynamic regression-based partitioning to optimize communication, and G3 [35] leverages graph native operations. To reduce communication in full-batch GNN training, CAGNET [54] uses the aggregate memory of GPU clusters and the NCCL multi-GPU communication library. The DGCL [4] communication library instead reroutes communications to use fast links with vertex replication. Despite distributing the graph data,

none of these solutions adopt locality-aware partitioning to reduce communication overheads without replication as we do.

As an alternative to whole-graph training systems, sampling approaches overcome the coordination and communication overheads through mini-batches [48]. In DistDGL [69], reduction of network communication traffic is achieved by partitioning and co-locating the vertex/edge features with their corresponding local partition data for distributed CPU. Solutions like AliGraph [61], AGL [65], and PaGraph [34] all optimize the sampling step in different ways. We also make use of mini-batch sampling techniques. By integrating the sampling step into our stochastic hypergraph construction phase, we reflect the randomness in communication volumes more accurately. DistDGLv2 [70] recently achieves a hybrid design with asynchronous sampling to overlap CPU and GPU computations, which motivates a future blended approach with our CPU/GPU versions of our communication scheme.

The resource under-utilization problem is worse in GPUs since mini-batch sampling overshadows training time [48], especially for sparse graphs [68]. Hence, many works, including our own, focus on CPU implementations. The communication architectures on CPU also involve different optimization considerations compared to GPU-based systems [42]. For example, Dorylus explores a CPU-based serverless asynchronous pipeline for scalability [52]. ByteGNN [68] recently improves resource utilization on CPUs with a partitioning strategy tailored for GNN sampling, however does not account for sparsity as we do, which gives us better speedups.

2.3 Data-Parallelization for GNNs

There are numerous studies on improving the efficiency of GNN computations, such as in cloud data processing systems on top of MapReduce [15] or Hadoop [20]. Ours is a data-parallel approach designed specifically for distributed training of GNNs, utilizing non-blocking parallel SpMM alongside local DMM. We make use of sparsity-based partitioning guided by a hypergraph model, to achieve non-blocking point-to-point communications for lowering communication costs. The potential of exploiting such data access patterns in GNN training has been recently highlighted as an open research question [31]. Graph convolution computations and GCN training depend highly on SpMM and DMM operations, therefore considering the access patterns in these computations is important in improving the resource efficiency and scalability of GCN training. The GE-SpMM algorithm [21] for GPUs allows integration with DGL for faster computation of GNNs by processing columns in parallel and ensuring coalesced access to sparse matrix data. Feat-Graph [19] co-optimizes graph traversal and feature dimension computation to offer efficient CPU/GPU implementations of sampled dense-dense matrix product (SDDMM) and SpMM in GNN training. FusedMM develops a general-purpose matrix multiplication kernel for graph embedding and GNN operations [44]. FusedMM unifies the matrix multiplications into a single operation since SpMM is frequently directly followed by DMM, but the approach is only applicable to shared-memory systems. In our algorithm, beyond point-to-point communications for SpMM and data locality for DMM, we pay special attention to the requirements of forward propagation and backpropagation during the training phase (aggregating features/gradients and updating parameters), and introduce a stochastic method to handle mini-batch sampling.

3 BACKGROUND

3.1 Graph Convolution

GCNs generalize the convolution operation to graphs having arbitrary size and topology, using an adjacency matrix to describe the (sparse) edge connections along which data aggregation takes place for every layer in the neural network.

Let $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ denote the adjacency matrix of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ which consists of $|\mathcal{V}| = n$ vertices. Vertex set $\mathcal{V}$ is associated with a feature matrix $\boldsymbol{H}^k \in \mathbb{R}^{n \times d_k}$ for every GCN layer, rows of which correspond to d_k-dimensional vertex features. Given an input feature matrix $\boldsymbol{H}^0$, feedforward of GCN is defined as

$$\begin{aligned} \boldsymbol{Z}^k &= \widehat{\boldsymbol{A}} \boldsymbol{H}^{k-1} \boldsymbol{W}^k \\ \boldsymbol{H}^k &= \sigma(\boldsymbol{Z}^k) \end{aligned} \tag{1}$$

for layers $k = 1, 2, \ldots L$. Matrix $\widehat{\boldsymbol{A}}$ is formed as $\widehat{\boldsymbol{A}} = \boldsymbol{D}^{-\frac{1}{2}} \widetilde{\boldsymbol{A}} \boldsymbol{D}^{-\frac{1}{2}}$ for normalization, where matrix $\widetilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$ corresponds to the adjacency matrix with self loops and matrix $\boldsymbol{D}(i,i) = \sum_j \widetilde{\boldsymbol{A}}(i,j)$ corresponds to the diagonal matrix of vertex degrees. To ease the notation, we will use $\boldsymbol{A}$ instead of $\widehat{\boldsymbol{A}}$ to denote the normalized adjacency matrix. In Equation (1), only the $\boldsymbol{A}$ matrix is sparse and the remaining matrices are dense. SpMM $\boldsymbol{A}\boldsymbol{H}^{k-1}$ combines feature vectors for each vertex (itself and neighbors). The resulting combined features are then involved in a DMM and multiplied by trainable parameter matrix $\boldsymbol{W}^k \in \mathbb{R}^{d_{k-1} \times d_k}$. Finally, a non-linear activation function $\sigma(\cdot)$ is applied to each element of matrix $\boldsymbol{Z}^k$ to compute $\boldsymbol{H}^k$.

The backpropagation phase requires a gradient matrix $\boldsymbol{G}^L \in \mathbb{R}^{n \times d_L}$ which is computed as

$$\boldsymbol{G}^L = \nabla_{\boldsymbol{H}^L} \mathbf{J} \odot \sigma'(\boldsymbol{Z}^L) \tag{2}$$

where $\nabla_{\boldsymbol{H}^L} \mathbf{J}$ denotes the matrix of derivatives of the loss function $\mathbf{J}$ with respect to output features in $\boldsymbol{H}^L$, $\sigma'(\cdot)$ denotes the derivative of the activation function, and symbol $\odot$ denotes element-wise multiplication (i.e., Hadamard product). Gradient matrices for the preceding layers for $k = L, L-1, \ldots, 1$ are recursively computed as

$$\begin{aligned} \boldsymbol{S}^k &= \boldsymbol{A}\boldsymbol{G}^k (\boldsymbol{W}^k)^T \\ \boldsymbol{G}^{k-1} &= \boldsymbol{S}^k \odot \sigma'(\boldsymbol{Z}^{k-1}) \end{aligned} \tag{3}$$

In Equation (3), SpMM is performed with matrices $\boldsymbol{A}$ and $\boldsymbol{G}^k$, and the resulting matrix is used in DMM with $(\boldsymbol{W}^k)^T$. Each gradient matrix $\boldsymbol{G}^k \in \mathbb{R}^{n \times d_k}$ is used to update parameter matrix $\boldsymbol{W}^k$ by the following set of gradient update rules

$$\Delta \boldsymbol{W}^k = (\boldsymbol{H}^{k-1})^T \boldsymbol{A}\boldsymbol{G}^k \tag{4}$$

$$\boldsymbol{W}^k \leftarrow \boldsymbol{W}^k - \eta \Delta \boldsymbol{W}^k \tag{5}$$

where $\Delta \mathbf{W}^k$ denotes the matrix of derivatives of the loss function $\mathbf{J}$ with respect to parameters in matrix $\boldsymbol{W}^k$, and η denotes the learning rate. It is important to note that, if the input graph is directed, transpose $\boldsymbol{A}^T$ is used instead of $\boldsymbol{A}$ in backpropagation (we refer the reader to [54] for a more detailed description).

3.2 Graph and Hypergraph Partitioning

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with vertex set $\mathcal{V}$ and edge set $\mathcal{E}$, a p-way partitioning of $\mathcal{G}$ is defined as $\Pi = \{\mathcal{V}_1, \mathcal{V}_2 \cdots \mathcal{V}_p\}$ consisting of subsets of vertices $\mathcal{V}_m \subset \mathcal{V}$ that are mutually disjoint ($\mathcal{V}_m \cap \mathcal{V}_n = \emptyset$ if $m \neq n$) and nonempty ($\mathcal{V}_m \neq \emptyset \; \forall \mathcal{V}_m \in \Pi$) where union of these subsets gives the vertex set ($\bigcup \mathcal{V}_m = \mathcal{V}$).

Each undirected edge $\{v_i, v_j\} \in \mathcal{E}$ between vertices $v_i, v_j \in \mathcal{V}$ is given a cost(v_i, v_j) and each vertex $v_i \in \mathcal{V}$ is associated with a weight $w(v_i)$, therefore the weight of a part $\mathcal{V}_m \in \Pi$ is defined as $W(\mathcal{V}_m) = \sum_{v_i \in \mathcal{V}_m} w(v_i)$. The partition Π is balanced if it satisfies the balancing constraint $W(V_m) \leq W_{avg}(1+\epsilon)$ for all $V_m \in \Pi$ where $W_{avg} = \sum_{v_i \in V} w(v_i)/p$ is the average part weight and ϵ is the maximum allowed imbalance ratio. Under a partitioning Π, an undirected edge $\{v_i, v_j\} \in \mathcal{E}$ is called cut edge if it connects vertices belonging two different parts. The p-way graph partitioning problem is defined as finding a partitioning Π such that the balancing constraint is satisfied and the total partitioning cost $\chi(\Pi) = \sum_{\{v_i, v_j\} \in \mathcal{E}_C} \text{cost}(v_i, v_j)$ is minimized where $\mathcal{E}_C$ denotes the set of cut edges.

Hypergraphs generalizes graphs by allowing hyperedges (nets) to connect more than two vertices. Let $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ denote a hypergraph consisting of vertex set $\mathcal{V}$ and net set $\mathcal{N}$, with Π defined as above. The set of vertices connected by a net $n_j \in \mathcal{N}$ is denoted by pins(n_j), where each net n_j is associated with cost(n_j). Under the partition Π, the connectivity set $\Lambda(n_j)$ is the set of parts that net n_j connects (i.e., pins$(n_j) \cap \mathcal{V}_m \neq \emptyset$). The number of such parts is called connectivity $\lambda(n_j) = |\Lambda(n_j)|$. If a net n_j connects to multiple parts (i.e., $\lambda(n_j) > 1$) it is said to be cut, and uncut otherwise. The connectivity cut size under Π is defined as $\chi(\Pi) = \sum_{n_j \in \mathcal{N}} \text{cost}(n_j) \times (\lambda(n_j) - 1)$. The hypergraph partitioning problem is therefore finding a p-way partition that satisfies the balancing constraint while minimizing the cut size, and is NP-Hard. There are successful tools that produce quality results for both graph and hypergraph partitioning problems [5, 24].

4 PARALLEL GCN TRAINING

We first present the feedforward and backpropagation steps of the proposed algorithm. Next, we describe a hypergraph partitioning model which reduces communication overheads over the graph model. We also propose a stochastic hypergraph model which encodes expected communication volume in mini-batch training.

4.1 Feedforward

The proposed parallel feedforward algorithm executes on p processors each of which is denoted by P_m for $m = 1, 2, \ldots p$. Adjacency matrix $\boldsymbol{A}$ and vertex feature matrices $\boldsymbol{H}^k$ for all layers $k = 0, 1, \ldots, L$ are 1D row-wise partitioned among processors where each processor P_m stores submatrices $\boldsymbol{A}_m \in \mathbb{R}^{n \times n}$ and $\boldsymbol{H}^k_m \in \mathbb{R}^{n \times d_k}$, which only contain subsets of rows of matrices $\boldsymbol{A}$ and $\boldsymbol{H}^k$. Adjacency matrix and feature matrices are conformably partitioned so that if row $\boldsymbol{A}(i,:)$ is assigned to submatrix $\boldsymbol{A}_m$, then the corresponding feature vectors $\boldsymbol{H}^k(i,:)$ for all layers k are assigned to submatrices $\boldsymbol{H}^k_m$, respectively (i.e., $\boldsymbol{A}(i,:) \in \boldsymbol{A}_m \Leftrightarrow \boldsymbol{H}^k(i,:) \in \boldsymbol{H}^k_m \; \forall k$). Parameter matrices $\boldsymbol{W}^k$ for all layers k are replicated and stored by all processors due to their relatively smaller sizes.

The matrix partitioning scheme encodes a vertex-partitioning on graph $\mathcal{G}$, since rows $\boldsymbol{A}(i,:)$ and $\boldsymbol{H}^k(i,:)$ denote the adjacency list and features of vertex $v_i \in \mathcal{V}$. Moreover, this partitioning also induces a task partitioning in feedforward phase: If a vertex v_i is assigned to a processor P_m, the task of computing row $\boldsymbol{Z}(i,:)^k$ of intermediate matrix $\boldsymbol{Z}^k$ in layer k is performed by processor P_m where row $\boldsymbol{Z}(i,:)^k$ is computed as

$$\boldsymbol{Z}^k(i,:) = \left(\sum_{j \in \text{cols}(\boldsymbol{A}(i,:))} \boldsymbol{A}(i,j) \boldsymbol{H}^{k-1}(j,:) \right) \boldsymbol{W}^k. \tag{6}$$

Hence, to compute submatrix $\boldsymbol{Z}^k_m$, processor P_m needs to receive all $\boldsymbol{H}^{k-1}$-matrix rows corresponding to all nonzero column indices in $\boldsymbol{A}_m$, which are not locally stored in $\boldsymbol{H}^{k-1}_m$. Let $\boldsymbol{H}^{k-1}_{nm} \in \mathbb{R}^{n \times d_{k-1}}$ denote the submatrix consisting of rows that are needed to be transferred from processor P_n to P_m. That is, submatrix $\boldsymbol{H}^{k-1}_{nm}$ contains subset of rows of $\boldsymbol{H}^{k-1}_n$ corresponding to the intersection of nonzero row indices of $\boldsymbol{H}^{k-1}_n$ and column indices of $\boldsymbol{A}_m$. More formally, $\exists i \in \text{rows}(\boldsymbol{H}^{k-1}_{nm})$ if $i \in \text{cols}(\boldsymbol{A}_m) \cap \text{rows}(\boldsymbol{A}_n)$. We use non-blocking point-to-point communications to transfer these submatrices between processors. After processor P_m receives submatrix $\boldsymbol{H}^{k-1}_{nm}$ from each processor P_n for all $n \neq m$ such that $\boldsymbol{H}^{k-1}_{nm} \neq \boldsymbol{0}$, P_m performs multiplication

$$\boldsymbol{Z}^k_m = (\boldsymbol{A}_m \boldsymbol{H}^{k-1}_m + \sum_{n \neq m} \boldsymbol{A}_m \boldsymbol{H}^{k-1}_{nm}) \boldsymbol{W}^k \tag{7}$$

to compute submatrix $\boldsymbol{Z}^k_m \in \mathbb{R}^{n \times d_k}$. Then, P_m applies the nonlinear activation function $\boldsymbol{H}^k_m = \sigma(\boldsymbol{Z}^k_m)$ to proceed to the next layer.

To manage sparse point-to-point communication operations, each processor P_m is provided with sets $\mathcal{S}_m$ and $\mathcal{R}_m$ which are computed before training with respect to the partitioning of adjacency matrix $\boldsymbol{A}$ among processors. Set $\mathcal{S}_m$ is composed of diagonal matrices $\boldsymbol{X}_{mn} \in \mathbb{R}^{n \times n}$ for each processor $P_n \neq P_m$. Matrix $\boldsymbol{X}_{mn}$ is used in a special matrix multiplication to determine which local $\boldsymbol{H}^{k-1}_m$-rows to be sent by processor P_m to P_n. Formally,

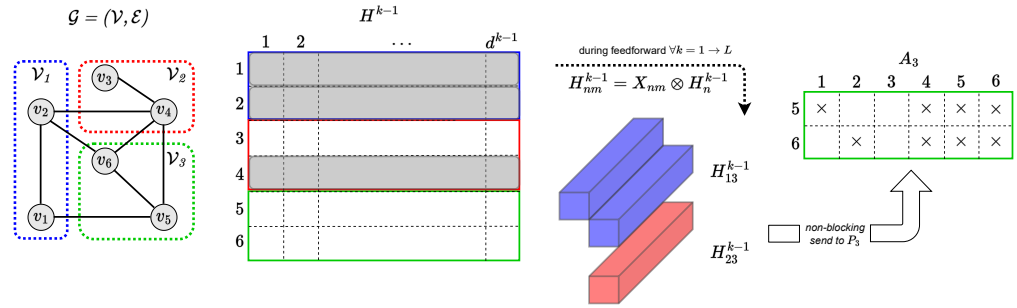
$$\begin{aligned} \mathcal{S}_m = \{ \boldsymbol{X}_{mn} \mid \boldsymbol{X}_{mn} \neq \boldsymbol{0} \wedge \boldsymbol{X}_{mn}(i,i) = 1 \\ \forall i \in \text{cols}(\boldsymbol{A}_n) \cap \text{rows}(\boldsymbol{A}_m) \}. \end{aligned} \tag{8}$$

That is, the ith diagonal entry $\boldsymbol{X}_{mn}(i,i) = 1$ if the intersection of nonzero row and column indices of matrices $\boldsymbol{A}_m$ and $\boldsymbol{A}_n$ contains index i, otherwise it is set to zero. Set $\mathcal{R}_m$ is composed of processors from which P_m receives at least one message. Formally,

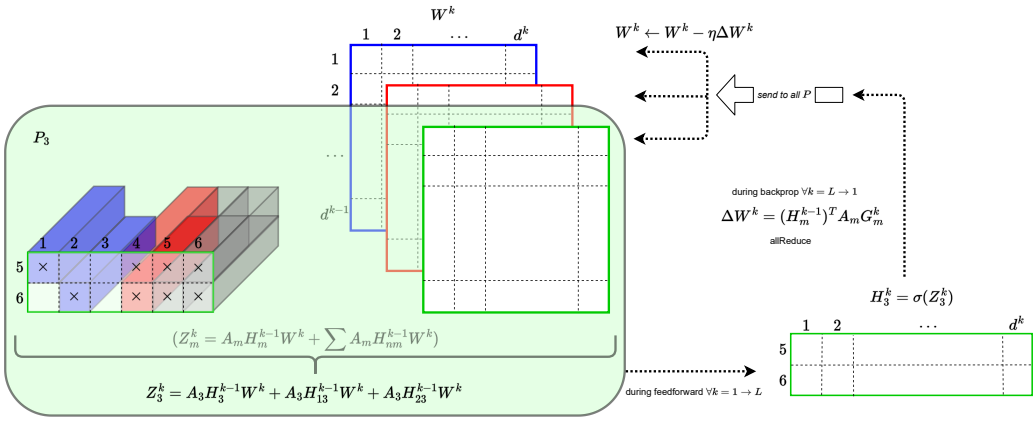
$$\mathcal{R}_m = \{ P_n \mid P_n \neq P_m \wedge cols(\boldsymbol{A}_m) \cap rows(\boldsymbol{A}_n) \neq \emptyset \}. \tag{9}$$

That is, processor P_n is included in $\mathcal{R}_m$ if the intersection of nonzero row and column indices of matrices $\boldsymbol{A}_m$ and $\boldsymbol{A}_n$ is nonempty, and processor P_m receives at least one row of $\boldsymbol{H}^{k-1}_n$ from P_n.

Algorithm 1 describes the proposed parallel feedforward algorithm. We used SuiteSparse:GraphBLAS (GB) [10] library to perform the sparse matrix operations. In lines 3–5, to overlap communication by computation, a non-blocking communication is performed for each diagonal matrix $\boldsymbol{X}_{mn} \in \mathcal{S}_m$ by processor P_m to send required $\boldsymbol{H}^{k-1}_m$-matrix rows to processor P_n. Matrix $\boldsymbol{H}^{k-1}_{mn}$ is formed through a specialized matrix multiplication $\boldsymbol{H}^{k-1}_{mn} = \boldsymbol{X}_{mn} \otimes \boldsymbol{H}^{k-1}_m$. By this matrix multiplication, if the ith diagonal entry is $\boldsymbol{X}_{mn}(i,i) = 1$, then the ith row $\boldsymbol{H}^{k-1}_m$ is copied into matrix $\boldsymbol{H}^{k-1}_{mn}$. Operator $\otimes$



(a) For graph $\mathcal{G}$, the feature matrix H^{k-1} for layer $k-1$ has been conformably partitioned along with A while the weight matrix W^k has been duplicated across all 3 processors. Processor P_3 (in green) requires H^{k-1}_{13} and H^{k-1}_{23} from the other two processors (in blue and red respectively).



(b) Computations are demonstrated here on the processor P_3 (in green) which stores A_3 and H^{k-1}_3 locally but must receive H^{k-1}_{13} and H^{k-1}_{23}. The resulting H^k_3 features are similarly computed for all layers ($k = 1 \to L$) during the feedforward phase. Subsequently, during backpropagation ($k = L \to 1$), these are used to generate ΔW^k for updation of the weight matrices W^k on all processors.

Figure 1: Communication and computation processes in feedforward and backpropagation of the distributed GCN algorithm.

denotes that the matrix multiplication is performed under semiring GxB_PLUS_SECOND, defined by GB library, to replace the multiplication operator with a copy operator that will directly carry the second operand to the resulting variable without multiplying (i.e., $z = x \times y \implies z = y$). In line 6, local matrix multiplication $\boldsymbol{Z}^k_m = \boldsymbol{A}_m \boldsymbol{H}^{k-1}_m \boldsymbol{W}^k$ is performed without waiting for the non-blocking communication operations to complete. Matrix $\boldsymbol{Z}^k_m$ is incomplete at this stage and its computation is finalized after receiving all necessary data. In lines 7–9, processor P_m receives $\boldsymbol{H}^{k-1}_{nm}$ from each $P_n \in \mathcal{R}_m$, and performs multiplication and addition $\boldsymbol{Z}^k_m = \boldsymbol{Z}^k_m + \boldsymbol{A}_m \boldsymbol{H}^{k-1}_{nm} \boldsymbol{W}^k$ to compute the final matrix $\boldsymbol{Z}^k_m$.

Figure 1 displays a sample execution of the feedforward phase. The adjacency matrix $\boldsymbol{A}$, and feature matrix $\boldsymbol{H}^k$ for each layer k are conformably partitioned among the three processors. Thus, each processor P_m only stores submatrices $\boldsymbol{A}_m$ and $\boldsymbol{H}^{k-1}_m$. For instance, the computation of matrix $\boldsymbol{Z}^k_3$ by processor P_3 (in green) requires the other two processors to send $\boldsymbol{H}^{k-1}_{13}$ and $\boldsymbol{H}^{k-1}_{23}$ corresponding to nonzero indices, and local matrix multiplication is performed without waiting for the completion of these non-blocking communications, to compute $\boldsymbol{H}^k_3$. Processor P_3 retrieves features of v_1 and v_4 for convolution on vertex v_5, and features of v_2, v_4 for vertex v_6. Hence, $\boldsymbol{H}^{k-1}_{13}$ contains rows 1 and 2 of $\boldsymbol{H}^{k-1}$ while $\boldsymbol{H}^{k-1}_{23}$ contains row 4, since these are the nonzero indices of $\boldsymbol{A}_m$ where the feature matrix rows are not locally stored. Note that row 4 is only transferred once to avoid a redundant communication.

Algorithm 1: Parallel Feedforward

```
1  forall processors P_m in parallel do
2      for k = 1 to L do
3          foreach X_mn ∈ S_m do
4              H^{k-1}_mn = X_mn ⊗ H^{k-1}_m
5              Non-blocking send H^{k-1}_mn to processor P_n
6          Z^k_m = A_m H^{k-1}_m W^k
7          foreach P_n ∈ R_m do
8              Receive H^{k-1}_nm from processor P_n
9              Z^k_m = Z^k_m + A_m H^{k-1}_nm W^k
10         H^k_m = σ(Z^k_m)
```

Algorithm 2: Parallel Backpropagation

1 **forall** processors P_m **in parallel do**
2 $\quad \boldsymbol{G}_m^L = \nabla_{\boldsymbol{H}_m^L}\mathbf{J} \odot \sigma'(\boldsymbol{Z}_m^L)$
3 $\quad$ **for** $k = L$ **to** 1 **do**
4 $\quad\quad$ **foreach** $\boldsymbol{X}_{mn} \in \mathcal{S}_m$ **do**
5 $\quad\quad\quad \boldsymbol{G}_{mn}^k = \boldsymbol{X}_{mn} \otimes \boldsymbol{G}_m^k$
6 $\quad\quad\quad$ Non-blocking send $\boldsymbol{G}_{mn}^k$ to processor P_n
7 $\quad\quad \boldsymbol{S}_m^k = \boldsymbol{A}_m \boldsymbol{G}_m^k (\boldsymbol{W}^k)^T$
8 $\quad\quad$ **foreach** $P_n \in \mathcal{R}_m$ **do**
9 $\quad\quad\quad$ Receive $\boldsymbol{G}_{nm}^k$ from processor P_n
10 $\quad\quad\quad \boldsymbol{S}_m^k = \boldsymbol{S}_m^k + \boldsymbol{A}_m \boldsymbol{G}_{nm}^k (\boldsymbol{W}^k)^T$
11 $\quad\quad \boldsymbol{G}_m^{k-1} = \boldsymbol{S}_m^k \odot \sigma'(\boldsymbol{Z}_m^{k-1})$
12 $\quad\quad \Delta\mathbf{W}_m^k = (\boldsymbol{H}_m^{k-1})^T (\boldsymbol{A}_m \boldsymbol{G}^k)$
13 $\quad\quad \Delta\mathbf{W}^k$ = Allreduce-sum($\Delta\mathbf{W}_m^k$)
14 $\quad\quad \mathbf{W}^k \leftarrow \mathbf{W}^k - \eta \Delta\mathbf{W}^k$

4.2 Backpropagation

In the backpropagation phase, similar to the vertex feature matrices, gradient matrices $\boldsymbol{G}^k$ for each layer k are row-wise partitioned among processors where each processor P_m holds submatrix $\boldsymbol{G}_m^k \in \mathbb{R}^{n \times d_k}$ in each layer k. Gradient matrix $\boldsymbol{G}^k$ and adjacency matrix $\boldsymbol{A}$ are conformably partitioned so that if row $\boldsymbol{A}(i,:)$ is assigned to submatrix $\boldsymbol{A}_m$, then row $\boldsymbol{G}^k(i,:)$ is assigned to submatrix $\boldsymbol{G}_m^k$ (i.e., $\boldsymbol{A}(i,:) \in \boldsymbol{A}_m \Leftrightarrow \boldsymbol{G}^k(i,:) \in \boldsymbol{G}_m^k \; \forall k$). Hence, the task of computing row $\boldsymbol{S}(i,:)^k$ of intermediate matrix $\boldsymbol{S}^k$ is given to processor P_m if row $\boldsymbol{A}(i,:)$ and corresponding vertex v_i is assigned to P_m. So, the same row-wise partitioning is induced on matrix $\boldsymbol{S}^k$ as with matrices $\boldsymbol{A}$ and $\boldsymbol{G}^k$. Matrix $\boldsymbol{S}^k$ is computed by following similar steps of computation of $\boldsymbol{Z}^k$ in feedforward phase. Then, P_m performs element-wise multiplication $\boldsymbol{G}_m^{k-1} = \boldsymbol{S}_m^k \odot \sigma'(\boldsymbol{Z}_m^{k-1})$.

Algorithm 2 gives the proposed parallel backpropagation algorithm. In line 2, each processor P_m computes submatrix $\boldsymbol{G}_m^L$ by using the local vertex-feature matrix $\boldsymbol{H}_m^L$ in the final layer. Here, $\nabla_{\boldsymbol{H}_m^L}\mathbf{J}$ denotes the matrix of partial derivatives of the loss function with respect to $\boldsymbol{H}_m^L$, and its formulation depends on the definition of the loss function. In lines 4–10, matrix $\boldsymbol{S}^k$ is computed in a similar way to computation of $\boldsymbol{Z}^k$ in Algorithm 1. In line 11, gradient matrix $\boldsymbol{G}^{k-1}$ for the preceding layer is computed via element-wise multiplication of matrices $\boldsymbol{S}_m^k$ and $\sigma'(\boldsymbol{Z}_m^{k-1})$. In line 12, each processor P_m computes partial results for gradient matrix $\Delta\mathbf{W}^k$ of the loss function $\mathbf{J}$ with respect to parameter matrix $\boldsymbol{W}^k$.

In the computation of $\Delta\mathbf{W}^k$, matrix $(\boldsymbol{H}_m^{k-1})^T$ is computed in feedforward phase, whereas $(\boldsymbol{A}_m\boldsymbol{G}^k)$ part is computed as a by-product in lines 7 and 10. Here, if column $(\boldsymbol{H}^{k-1})^T(:,i)$ is stored in $(\boldsymbol{H}_m^{k-1})^T$, then the corresponding row $(\boldsymbol{A}\boldsymbol{G}^k)(i,:)$ is also stored in $(\boldsymbol{A}_m\boldsymbol{G}^k)$. Therefore, multiplication $(\boldsymbol{H}_m^{k-1})^T(\boldsymbol{A}_m\boldsymbol{G}^k)$ by processor P_m produces matrix $\Delta\mathbf{W}_m^k$ of partial products where each nonzero $\Delta\mathbf{W}_m^k(i,j)$ contributes to the corresponding nonzero

$$\Delta\mathbf{W}^k(i,j) = \sum_m \Delta\mathbf{W}_m^k(i,j)$$

in the final matrix $\Delta\mathbf{W}^k$. In line 13, the final gradient matrix $\Delta\mathbf{W}^k$ is computed via an allreduce-type communication operation which combines (sums) partial matrices from all processes and distributes the result back to all processes. In line 14, gradient update on $\boldsymbol{W}^k$ is performed by all processors on their local copies.

Figure 1 also displays the additional computations performed in the backpropagation phase. As seen in the figure, the relatively smaller-sized weight matrices $\boldsymbol{W}^k$ for each layer k are replicated among all processors. The computation of matrix $\boldsymbol{S}^k$ is identical with the computation of matrix $\boldsymbol{H}^k$ and requires the same communication steps which are determined by the partitioning on the adjacency matrix $\boldsymbol{A}$. Matrix $\boldsymbol{S}^k$ is used together with matrix $\boldsymbol{Z}^{k-1}$ to compute gradient matrix $\boldsymbol{G}^{k-1}$. The figure also shows the allreduce operation performed on locally computed matrices $\Delta\boldsymbol{W}_m^k$ to compute the final matrix $\Delta\boldsymbol{W}^k$ for gradient update operations.

4.3 Partitioning Models

Different partitioning models may be used for partitioning the adjacency matrix among processors. We compare the graph and hypergraph models and highlight how the hypergraph model correctly encodes the total communication volume during the message-passing operations. Finally, we present our novel stochastic hypergraph model which encodes expected communication volume instead of exact values, and therefore supports mini-batch training.

4.3.1 *Graph Model*.

In a graph model, a p-way partitioning $\Pi_p = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_p\}$ over vertex set $\mathcal{V}$ induces a row-wise partitioning on matrix $\boldsymbol{A}$ among p processors. If a vertex v_i is assigned to part $\mathcal{V}_m \in \Pi_p$, then row $\boldsymbol{A}(i,:)$ is assigned to processor P_m. Note that the input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in GCN training can be directed or undirected, but graph partitioning tools (e.g., METIS) assume that the graph to be partitioned is undirected, edges have integer costs, and vertices have integer weights. Therefore, we build an undirected graph $\mathcal{G}' = (\mathcal{V}, \mathcal{E}')$ where we use vertex set $\mathcal{V}$ as is, but replace each directed edge $(v_i, v_j) \in \mathcal{E}$ with an undirected edge $\{v_i, v_j\} \in \mathcal{E}'$.

Under a partition Π_p, each undirected cut edge $\{v_i, v_j\}$ represents the communication of $\boldsymbol{H}^{k-1}(i,:)$- and $\boldsymbol{H}^{k-1}(j,:)$-matrix rows between respective processors during feedforward phase, and communication of $\boldsymbol{G}^k(i,:)$- and $\boldsymbol{G}^k(j,:)$-matrix rows during backpropagation phase. Since we have d-dimensional vertex feature matrix $\boldsymbol{H}^k \in \mathbb{R}^{n \times d_k}$, each undirected edge encodes a total communication volume of $\sum_k 2(d_{k-1}+d_k)$ nonzero entries over all layers k. Because the communication volume encoded by each edge is the same constant value, each undirected edge $\{v_i, v_j\} \in \mathcal{E}'$ can be associated with a unit cost$(v_i, v_j) = 1$. Each vertex v_i is associated with a computational weight $w(v_i) = |\text{cols}(\boldsymbol{A}(i,:))|$. DistDGL [69] also utilizes this partitioning scheme and partitions the input graph via METIS, only considering undirected graphs.

What makes the graph model less accurate compared to the hypergraph model is that the former overestimates the total communication volume between processors. This deficiency of the graph model can be seen in two ways: (i) When both of the directed edges (v_i, v_j) and (v_j, v_i) are not simultaneously present in the input graph $\mathcal{G}$, the graph model still considers an undirected edge $\{v_i, v_j\}$ that sees communication in both ways although the

communication is actually one-way. (ii) If a vertex v_i is connected to vertices v_j and v_k that are stored together but on a different processor from v_i, the graph model assumes that the features of v_i are sent twice. However, these features are sent to that processor once in a single message. These two cases cause the partitioning cut size to be higher than the actual communication volume.

4.3.2 *Hypergraph Model*.

We model one-dimensional (1D) row-wise partitioning of adjacency matrix as a hypergraph partitioning problem [5] since the hypergraph model can encode the exact communication volume of parallel GCN. The connectivity cut size of the hypergraph model encodes the total communication volume among processors, while weights of partitions encode the associated computational load for processors. Hence, minimization of the connectivity cut size under weight-balancing constraints achieves minimization of the total communication volume while achieving computational-load balance. During the feedforward phase, the hypergraph model encodes the total communication volume on $\boldsymbol{H}^{k-1}$-matrix rows for parallel SpMMs $\boldsymbol{A}\boldsymbol{H}^{k-1}$ among processors in each layer k. The model also encodes the total communication volume on $\boldsymbol{G}^k$-matrix rows for parallel SpMMs $\boldsymbol{A}\boldsymbol{G}^k$ during backpropagation phase.

To partition adjacency matrix $\boldsymbol{A}$, we first build a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ where for each matrix row $\boldsymbol{A}(i,:)$ there exists one vertex $v_i \in \mathcal{V}$ and for each column $\boldsymbol{A}(:,j)$, there exists one net $n_j \in \mathcal{N}$. Similar to the graph model, a partitioning obtained on the vertex set of the input graph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ also induces a 1D row-wise partitioning on the adjacency matrix. That is, a p-way partitioning $\Pi_p = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_p\}$ over vertex set $\mathcal{V}$ induces a row-wise partitioning on matrix $\boldsymbol{A}$ among p processors, since each vertex v_i corresponds to row $\boldsymbol{A}(i,:)$. Additionally, each vertex $v_i \in \mathcal{V}$ also represents the task of computing rows $\boldsymbol{Z}^k(i,:)$ and $\boldsymbol{S}^k(i,:)$ in each layer k. Therefore, each vertex v_i is associated with weight $w(v_i) = |\text{cols}(\boldsymbol{A}(i,:))|$, i.e., the number of nonzero column indices in the ith row of matrix $\boldsymbol{A}$, to encode the computational load of the task represented by vertex v_i. Note that the number of nonzero arithmetic operations required to compute rows $\boldsymbol{Z}^k(i,:)$ and $\boldsymbol{S}^k(i,:)$ is proportional to the number of nonzero column indices in row $\boldsymbol{A}(i,:)$. So, satisfying the balancing constraints in hypergraph partitioning achieves computational-load balance.

Net set $\mathcal{N}$ encodes task dependencies on rows of matrices $\boldsymbol{H}^{k-1}$ and $\boldsymbol{G}^k$ during feedforward and backpropagation phases for each layer k. Each net $n_j \in \mathcal{N}$ connects all vertices $v_i \in \mathcal{V}$ for which the corresponding row $\boldsymbol{A}(i,:)$ has a nonzero entry in the jth column. For computing rows $\boldsymbol{Z}(i,:)^k$ and $\boldsymbol{S}(i,:)^k$, the processor that owns row $\boldsymbol{A}(i,:)$ needs all $\boldsymbol{H}^{k-1}$- and $\boldsymbol{G}^k$-matrix rows, corresponding to nonzero column indices $\text{cols}(\boldsymbol{A}(i,:))$, respectively. Therefore, pins of a net n_j denotes the tasks that require row $\boldsymbol{H}^{k-1}(j,:)$ and $\boldsymbol{G}^k(j,:)$. Formally, pins of a net n_j can be written as

$$\text{pins}(n_j) = \{v_i \in V \mid \exists j \in \text{cols}(\boldsymbol{A}(i,:))\}. \tag{10}$$

Under a partitioning Π_p, a net $n_j \in \mathcal{N}$ with connectivity set $\Lambda(n_j)$ encodes the total communication volume on rows $\boldsymbol{H}^{k-1}(j,:)$ and $\boldsymbol{G}^k(j,:)$ in each layer k. Here, at least one part in $\Lambda(n_j)$ stores vertex v_j since each diagonal entry contains a nonzero entry in adjacency matrix $\boldsymbol{A}$. That is, for all net $n_j \in \mathcal{N}$, vertex $v_j \in \text{pins}(n_j)$. Therefore, a part $V_m \in \Lambda(n_j)$ stores vertex v_j and hence, processor P_m stores rows $\boldsymbol{H}^{k-1}(j,:)$ and $\boldsymbol{G}^k(j,:)$ in its local submatrices $\boldsymbol{H}^{k-1}_m$ and $\boldsymbol{G}^k_m$, respectively. Due to the task dependencies encoded by net n_j, processor P_m sends row $\boldsymbol{H}^{k-1}(j,:)$ to all processors corresponding to parts in $\Lambda(n_j) \setminus \mathcal{V}_m$ during feedforward phase, i.e., $\lambda(n_j) - 1$ communications. Similarly, processor P_m sends row $\boldsymbol{G}^k(j,:)$ to all processors in $\Lambda(n_j) \setminus \mathcal{V}_m$ during backpropagation phase. If a processor $P_n \in \Lambda(n_j) \setminus \mathcal{V}_m$ has multiple vertices connecting to net n_j, processor P_n receives row $\boldsymbol{H}^{k-1}(j,:)$ and row $\boldsymbol{G}^k(j,:)$ only once. So, net n_j incurs a communication volume of $\text{cost}(n_j) \times \big(\lambda(n_j) - 1\big)$ where the cost of net n_j is denoted as $\text{cost}(n_j) = \sum_k d_{k-1} + d_k$ since all nonzero entries in rows $\boldsymbol{H}^{k-1}(j,:)$ and $\boldsymbol{G}^k(j,:)$ are communicated in each layer k. Since the cost of each net is the same constant value, we can also associate each net with a unit cost $cost(n_j) = 1$. Therefore, the total communication volume can be written as

$$\sum_{n_j \in \mathcal{N}} 2 \times \text{cost}(n_j) \times \big(\lambda(n_j) - 1\big) \tag{11}$$

which indicates that minimizing the connectivity cut size corresponds to minimizing the total communication volume.

Figure 2 displays an illustrative example of the proposed hypergraph partitioning model on a sample graph $\mathcal{G}$ having adjacency matrix $\boldsymbol{A}$. The hypergraph $\mathcal{H}$ is constructed having parts $\mathcal{V}_1$ (blue), $\mathcal{V}_2$ (red), and $\mathcal{V}_3$ (green) each containing two vertices, with a net n_j for every column $\boldsymbol{A}(:,j)$. According to the hypergraph partitioning model, rows of $\boldsymbol{A}$ are assigned to processors based on the hypergraph vertex partitioning. For example, row $\boldsymbol{A}(i,:)$ will be stored on processor P_1 as vertex v_1 is assigned to $\mathcal{V}_1$. Since v_1 represents a task, the computational load is proportional to the number of non-zero columns in row 1 and is encoded by its weight $w(v_1) = 3$. Each net connects non-zero entries in a row. For example net n_2 connects $pins(n_2) = \{v_1, v_2, v_4, v_6\}$ with connectivity set $\Lambda(n_2) = \{\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3\}$. Its connectivity is therefore $\lambda(n_2) = 3$. The net set $\mathcal{N}$ thus encodes task dependencies during the feedforward and backpropagation phases since communication operations on matrices $\boldsymbol{H}^k$ and $\boldsymbol{G}^{k-1}$ are identical and determined by the partitioning on matrix $\boldsymbol{A}$. The feature matrix $\boldsymbol{H}^{k-1}$ is conformably partitioned with $\boldsymbol{A}$ for each layer of the GCN, while the weight matrix $\boldsymbol{W}^k$ is replicated across the three processors.

Figure 2 also depicts how the graph model overestimates the communication volume. Features of vertex v_4 must be fetched by vertices v_2, v_3, v_5, and v_6. For example, according to the graph model, the feature vector of v_4 is encoded as if it were sent from processor P_2 to processor P_3 twice, but should only be sent once. Therefore, cut edges connecting to vertex v_4 in the graph encodes a communication volume of 3 instead of the true value of 2. On the other hand, the hypergraph model shown on $\mathcal{H}$ uses net n_4 to encode communications from vertex v_4. Since the connectivity of n_4 is $\lambda(n_4) = 3$ and hypergraph partitioning minimizes connectivity−1 metric, net n_4 encodes the true communication volume as $\lambda(n_4) - 1 = 2$.

4.3.3 *Stochastic Hypergraph Model*.

In mini-batch training, a stochastic sampling is applied to the input graph to produce subgraphs on which convolutions are performed. We propose a novel stochastic hypergraph model which encodes and minimizes the *expected* communication volume in mini-batch

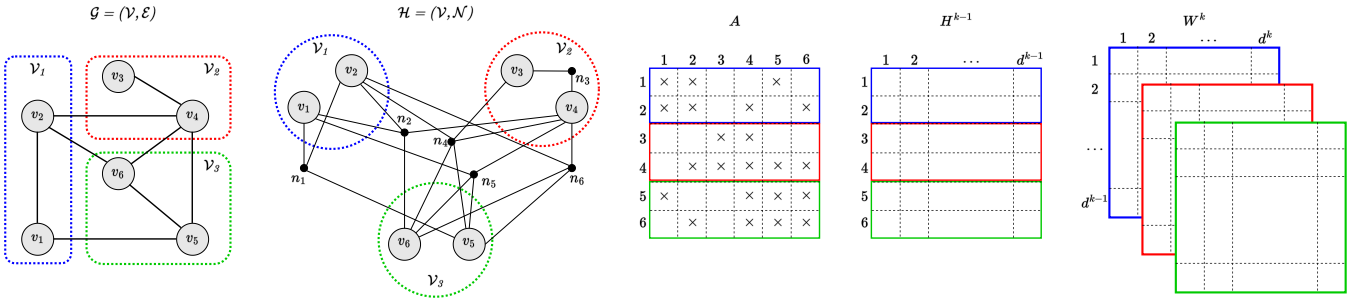


Figure 2: Hypergraph partitioning of graph $\mathcal{G}$ having adjacency matrix A (including self loops), by constructing the corresponding hypergraph $\mathcal{H}$ where every net n_j connects nonzero entries of the column i in A. The feature matrix H^k for layer k is conformably partitioned along with A, while the weight matrix W^k is duplicated across all processors.

Algorithm 3: Stochastic Hypergraph Partitioning

1 Generate b subgraphs $G_i = (V_i', E_i')$ of G for $i = 1, 2, \ldots, b$
2 Build hypergraph $H_i = (V_i', N_i')$ for each $G_i = (V_i', E_i')$
3 Build stochastic hypergraph $H = (V = \bigcup_{i=1}^{b} V_i', N = \bigcup_{i=1}^{b} N_i')$
4 Partition p-way hypergraph H to obtain partitioning $\Pi = \{V_1, V_2, \ldots, V_p\}$
5 **Return** Π

training. Note that the hypergraph/graph models described earlier encode the communication volume in full-batch training.

We first randomly generate mini-batches (i.e., subgraphs) using a sampling technique. Next, for each subgraph, we build a hypergraph that encodes the total communication volume for the mini-batch. By merging all hypergraphs generated (one per mini-batch), we build a larger hypergraph that can encode the *expected* connectivity of any randomly generated net. Partitioning the resulting merged stochastic hypergraph minimizes the expected connectivity of a random net, and thus minimizes the expected total communication volume for any randomly generated mini-batch.

More formally, given an input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, each mini-batch corresponds to a subgraph $\mathcal{G}' = (\mathcal{V}' \subset \mathcal{V}, \mathcal{E}' \subset \mathcal{E})$. We generate b mini-batches, each corresponding to a subgraph $\mathcal{G}_i' = (\mathcal{V}_i', \mathcal{E}_i')$ for $i = 1, 2, \ldots, b$. For each such subgraph $\mathcal{G}'$ a hypergraph $\mathcal{H}' = (\mathcal{V}', \mathcal{N}')$ is built in the same way as in full-batch training. The stochastic hypergraph $\mathcal{H} = (\mathcal{V} = \bigcup \mathcal{V}_i', \mathcal{N} = \bigcup \mathcal{N}_i')$ is formed by merging all vertex and net sets into the corresponding sets for the merged hypergraph. The proposed stochastic hypergraph partitioning process is described in Algorithm 3 which returns a partitioning Π of $\mathcal{H}$ to determine the row-wise partitioning of the adjacency matrix.

Under a p-way vertex partition Π of the stochastic hypergraph $\mathcal{H}$, let λ denote the expected connectivity of a randomly generated net. By using Hoeffding's inequality, the value of λ can be estimated within its θ error with a probability of at least $1-\delta$. That is, let λ_i be a random variable that denotes the connectivity of a randomly generated net where $1 \leq \lambda_i \leq p$ (since a net connects at least one part and at most p parts). Let $\lambda' = \frac{1}{|N|} \sum \lambda_i$ be the estimation for λ where $|N|$ denotes the total number of nets obtained in the stochastic hypergraph. By Hoeffding inequality,

$$Pr[|\lambda' - \lambda| \geq \theta] \leq 2 \exp(\frac{-2|N|\theta^2}{(p-1)^2}) \tag{12}$$

To achieve $1 - \delta$ confidence,

$$2 \exp(\frac{-2|N|\theta^2}{(p-1)^2}) \leq \delta \tag{13}$$

must be achieved. Hence, solving this equation for $|N|$ gives

$$|N| \geq \frac{(p-1)^2}{2\theta^2} \ln \frac{2}{\delta} \tag{14}$$

which denote the smallest number of nets needed to achieve the θ error with $1-\delta$ confidence.

As shown, if an adequate number of nets are generated, the stochastic hypergraph model encodes the expected communication volume with low error with high probability. Since the expected connectivity λ is determined by the partitioning Π over the hypergraph, the stochastic hypergraph partitioning can minimize this objective. Additionally, if each vertex is equally likely to be selected in a mini-batch, then the same vertex weighting and balancing constraint in hypergraph model for full-batch training can be applied here to achieve computational load balance.

4.4 Extension to GNNs

The main difference between general GNN models [26, 55, 58] and GCNs is in the way messages are created and combined between vertices. In some GNN models, DMM is performed first and messages are created, which is followed by a specialized SpMM for message passing and combining. For example, in a GAT [55], first each vertex feature is transformed with a local parameter matrix (i.e., DMM), and the resulting feature is transmitted to neighbor vertices using the same communication pattern as in SpMM. At the destination vertex, features are concatenated and then multiplied with an attention vector. That is, the order of SpMM and DMM can be changed and additional mathematical operations can be applied to their outputs, without affect the message directions and communication patterns between vertices. Therefore, our proposed partitioning method can be directly used for other GNN models, and simple modifications to the proposed GCN algorithm can support the additional computations necessary alongside the same communication scheme as before.

Table 1: Dataset properties

Dataset	Vertices	Edges	Type
amazon0601	403,394	3,387,388	Directed
cit-Patents	3,774,768	16,518,948	Directed
coPapersDBLP	540,486	30,491,458	Undirected
com-Amazon	334,863	1,851,744	Undirected
com-Youtube	1,134,890	5,975,248	Undirected
flickr	820,878	9,837,214	Directed
roadNet-CA	1,971,281	5,533,214	Undirected
soc-Slashdot0902	82,168	948,464	Directed
Cora	2708	10556	Undirected
ogbn-Papers100M	111,059,956	1,615,685,872	Directed
Reddit	232,965	114,615,892	Undirected

5 EXPERIMENTAL RESULTS

We evaluate the performance of the proposed parallel GCN training algorithm on a diverse set of real-world graphs from popular applications that use GCN models such as citation networks, social networks, road networks, and product co-purchasing networks. Properties of these graphs are displayed in Table 1.

We use DGL (with PyTorch v1.6 backend) implementation of GCN as the baseline, and compute speedup values according to its single-node CPU performance. We also compare our performance against CAGNET [54] which is the algorithm most related to our own, by using both the original GPU implementation and our own CPU implementation of CAGNET. We omit comparisons against Neugraph [37] and ROC [22] as they are not compatible with CPU clusters, and CAGNET already provides much more scalability. To the best of our knowledge, our algorithm is the first parallel GCN training algorithm designed for CPU clusters.

We evaluate the improvements in performance of the proposed parallel GCN training algorithm with both hypergraph partitioning (HP) and graph partitioning (GP) models used to partition the input matrices. We also evaluate our novel stochastic hypergraph partitioning (SHP) model for mini-batching. Additionally, we report results for random partitioning (RP) as a baseline, which evenly splits the adjacency matrix by assigning rows to processors uniformly at random, and is a competitive method for balancing computational load and communications.

We run our CPU experiments on a cluster of 180 compute nodes with 2x Intel Xeon Platinum 8268 2.9 GHz 24-core processors (48 cores per node) and 4GB RAM per core. Our GPU experiments use the Sulis cluster of 30 nodes each with 3x NVIDIA A100 GPUs and 4GB RAM per core. Both use InfiniBand interconnect (100 Gbit/s) and Slurm Workload Manager. The single-node DGL implementation requires a server with a better hardware configuration. We use a 16-core Intel Xeon 3.90GHz processor with 500 GB memory.

Our CPU code is in C++, using SuiteSparse:GraphBLAS library for local sparse matrix operations and MPI for point-to-point communication operations. The GPU version in Python uses PyTorch with NCCL backend to perform communication operations [1]. Incorporating future support for asynchronous communication (as in our CPU implementation) may help overcome the limitations of NCCL to overlap communication and computation for better performance gains on GPU. We used PaToH [6] hypergraph partitioning tool and METIS [24] graph partitioning tool. For `ogbn-Papers100M`, we used KaHyPar [46] which can handle massive-scale graphs. We used both partitioning tools with their default parameters and set the maximum imbalance ratio as $\epsilon = 0.01$.

Table 2: Performance comparison with HP, GP, and RP on $P = 512$ processors

			Volume		Messages		
		R	Avg	Max	Avg	Max	S
amazon0601	HP	0.63	0.12	0.29	0.22	0.51	10.88
	GP	0.65	0.18	0.31	0.30	0.62	10.55
	HP/GP	0.97	0.67	0.92	0.74	0.82	
cit-Patents	HP	0.77	0.17	0.29	0.70	0.89	8.48
	GP	0.80	0.19	0.50	0.77	0.94	8.10
	HP/GP	0.95	0.88	0.57	0.91	0.95	
coPapersDBLP	HP	0.32	0.07	0.08	0.42	0.71	10.93
	GP	0.69	0.07	0.16	0.57	0.77	5.04
	HP/GP	0.46	0.97	0.49	0.74	0.92	
com-Amazon	HP	0.32	0.09	0.20	0.14	0.32	14.31
	GP	0.37	0.14	0.27	0.19	0.42	12.37
	HP/GP	0.86	0.60	0.73	0.72	0.75	
com-Youtube	HP	0.40	0.36	0.52	0.72	0.97	10.85
	GP	1.45	0.37	2.60	0.90	0.99	3.01
	HP/GP	0.28	0.98	0.20	0.81	0.98	
flickr	HP	0.81	0.45	0.60	0.79	1.00	9.59
	GP	11.13	0.38	6.89	0.96	1.00	0.70
	HP/GP	0.07	1.19	0.09	0.82	1.00	
roadNet-CA	HP	0.19	0.01	0.01	0.01	0.03	30.32
	GP	0.20	0.01	0.02	0.01	0.03	29.08
	HP/GP	0.96	0.78	0.67	1.03	1.00	
soc-Slashdot0902	HP	0.75	0.74	0.69	0.86	0.92	3.50
	GP	2.02	0.85	4.38	0.93	1.00	1.30
	HP/GP	0.37	0.86	0.16	0.92	0.92	
	mean HP	0.47	0.13	0.21	0.29	0.48	10.60
	mean GP	0.98	0.15	0.56	0.35	0.52	5.04
	mean HP/mean GP	0.48	0.87	0.37	0.83	0.92	

Communication Costs. Table 2 compares HP, GP, and RP in terms of the communication volume and message counts metrics they incur on 512 processors (i.e., MPI processes). For each partitioning method, we ran the parallel GCN algorithm with random vertex features and label data for five epochs and measured the running time average and total communication cost metrics. These metrics respectively relate to bandwidth and latency costs induced by different partitioning strategies on parallelization costs. In the table, both the average and maximum volume/number of messages sent by a processor are displayed. Average values are proportional to the total message volume/count values, and are used to display how much the maximum values deviate from the mean.

For each input graph, the first and second rows denote the respective values attained by HP and GP, where these values are normalized with respect to values attained by RP. The third row denotes the ratios of values attained by HP and GP (i.e., HP/GP). The first column (i.e., "R") in the table indicates the ratio of the parallel running time of HP and GP to that of RP. The last column (i.e., "S") denotes the speedup values attained by HP and GP with respect to single-node running time performance of DGL. At the end of

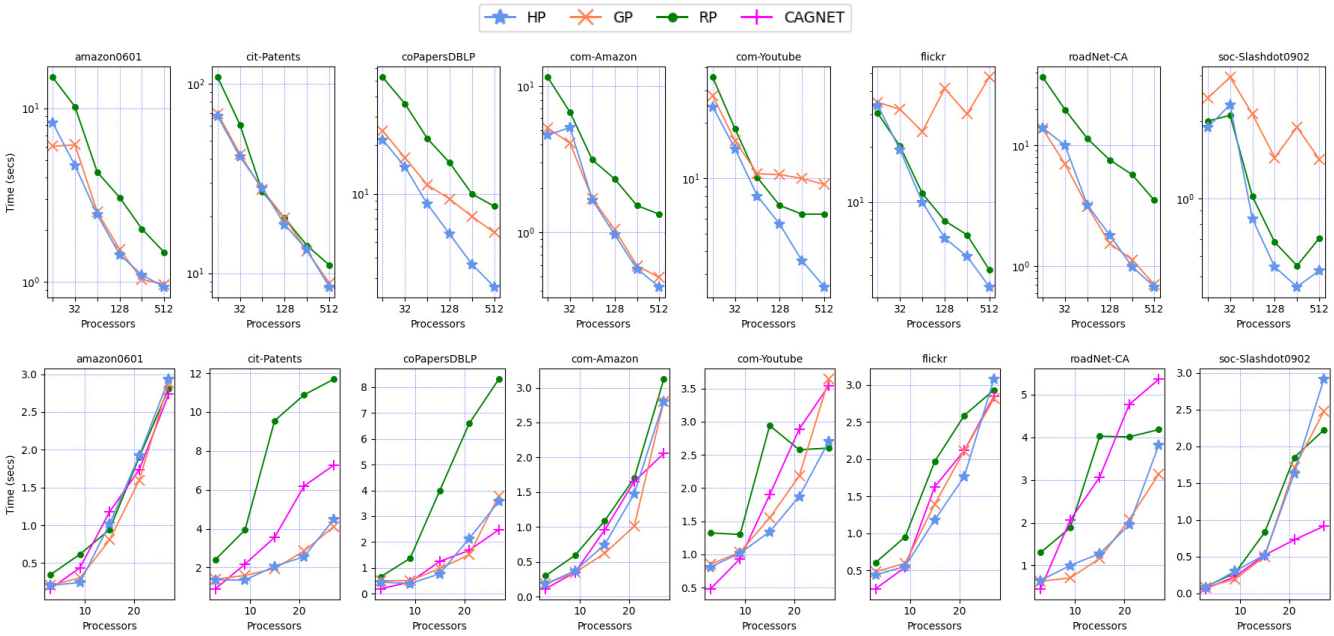


Figure 3: Strong scaling for full-batch training with HP, GP, and RP on $P = 16$ to $P = 512$ CPUs (top row) and with HP, GP, RP, and CAGNET (CN) on $P = 3$ to $P = 27$ GPUs (bottom row).

the table, the geometric means of the normalized values for HP and GP are given where the ratios of these values are given in the last row. For instance, the "R" and "S" columns for `amazon0601` are interpreted as follows: Parallel running times of HP and GP divided by that of RP is 0.63 and 0.65, respectively. The parallel running time of HP divided by that of GP is 0.97. Speedups achieved by HP and GP with respect to DGL are displayed under column "S" as 10.88 and 10.55, respectively.

As seen in Table 2, both HP and GP provide significant improvements over communication volume and message count metrics. On average, HP and GP incur 87% and 85% less average communication volume than RP respectively, with HP performing 15% better than GP. In terms of maximum communication volume, HP consistently outperforms RP, providing 79% improvement on average over RP. HP performs 63% better than GP and provides better communication balance. Even though GP provides 44% improvement on average over RP, its performance significantly degrades for graphs `com-Youtube`, `flickr` and `soc-Slashdot0902` where for instance, GP performs 6.89x worse than RP for `flickr`. Although both partitioning methods provide significant improvement in average communication volume, graph partitioning can disrupt the communication balance between processors. For the message count metrics, on average, HP and GP reduce the total number of messages by 71% and 65% as compared to RP while HP performs 17% better than GP. Similarly, maximum message count is respectively reduced by 52% and 48% by HP and GP while HP performs 9% better than GP.

Parallel Running Times. Improvements in communication costs by HP and GP considerably reduce parallel running of RP. On all graphs, HP provides an average of 2.12x speedup over RP. GP runs 20%–80% faster than RP for most graphs. However, it is slower than RP on `flickr`, `com-Youtube` and `soc-Slashdot0902`. The reason for this is the communication volume imbalance, as can be seen from the maximum and average communication volumes achieved on these graphs. Note that RP achieves a good communication and computation balance. The best performance is achieved by HP and GP for `roadNet-CA` where both partitioning methods provide approximately 99% improvement in communication volume and message count metrics compared to RP and runs 5x faster.

As seen in the speedup column, on average, HP and GP provide 10.60x and 5.04x speedup respectively over the DGL implementation. The best speedup is achieved for `roadNet-CA` where HP and GP approximately provide 30.32x and 29.08x speedup. This is because road networks are relatively more sparse as compared to the other social networks and hence the amount of data transferred between processors reduces in such cases. As the graph sizes increase and graphs become more sparse, partitioning tools usually perform better optimizations.

Figure 3 displays strong scaling of HP, GP, and RP on CPU (first row) and GPU (second row) clusters. As seen here, on the CPU cluster, HP achieves almost linear speedup up to 512 cores on all input graphs. Additionally, HP either matches or outperforms GP, and always outperforms RP. The reason for the speedup loss of HP on 512 processors for `soc-Slashdot0902` is the relatively smaller size of the graph as compared to the others. In general, GP performs better than RP except for `flickr`, `com-Youtube` and `soc-Slashdot0902` graphs due to degradation of communication balance as also shown in Table 2. Here, we omit plot for our CPU implementation of CAGNET since HP and GP significantly outperform it.

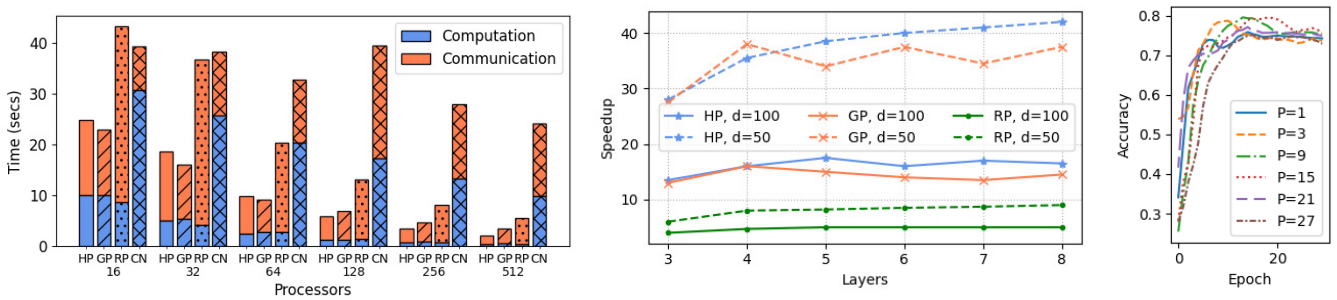


Figure 4: Performance comparisons for full-batch training. (a) Communication time and computation time split with HP, GP, RP, and CAGNET (CN) on `coPapersDBLP` for $P = 16$ to $P = 512$ CPUs. (b) Speedup with increasing layers ($L = 3, 4, \ldots, 8$) and dimensions ($d = 50, 100$) on `roadnet-CA` for $P = 512$ CPUs. (c) GNN model accuracy with HP on `Cora` for $P = 1$ to $P = 27$ GPUs.

We also demonstrate that algorithms that are focused on optimizing communication operations are better suited to CPU clusters over GPUs, and where the sparsity of the problem is important to exploit. As seen in Figure 3, for GPU cluster experiments, the PyTorch implementations (with NCCL backend) of both CAGNET and our parallel GCN (i.e. HP, GP and RP) do not scale well (up to 27 GPUs). The reason for this is the high communication efficiency needed to attain speedup on GPUs. In comparison to MPI, the NCCL backend cannot provide the necessary efficiency. On GPUs, the proportion of total running time that is spent on local computation is small, therefore the gains obtained via parallelization do not amortize the time spent for communication on larger GPU counts. In addition, despite the optimizations obtained in the communication volume from our algorithm, with the NCCL backend these are not as effective as with MPI. This results in limited performance improvement on the overall parallel run time due to the higher latency costs. HP and GP continue to be faster than CAGNET for most datasets and settings. In addition, the performance improvement is expected to be more stark at higher GPU counts, as can be seen from the CPU cluster results, but no suitable larger GPU clusters were available for our experiments. Moreover, we find that our parallel CPU implementation for HP is able to outperform the GPU version in many cases. For example, on `amazon0601`, the running time is 0.94 seconds on 512 CPUs, while on 15 GPUs it takes as long as 1.02 seconds. On the larger `roadNet-CA` dataset, the same setting takes only 0.67 seconds on CPU and twice as long (1.28 seconds) on GPU.

Communication and Computation Times. Figure 4a analyses the breakdown of communication and local computation times in the total parallel CPU running time of HP, GP, RP, and CAGNET (CN) for `coPapersDBLP`. On all processor counts, HP and GP consistently perform better than CAGNET, with HP being the best method at high processor counts. On the largest processor count, HP runs nearly 12x faster than CAGNET. Even though RP performs worse than CAGNET on $P = 16$ processors, its performance becomes better as the number of processors increases. As seen in the figure, the total communication time decreases with the total computation time for HP, GP and RP as the number of processors increases, whereas the communication time of CAGNET increases. This is because point-to-point communication necessitates each processor to communicate only with a small subset of processors and thus incurs lower communication volume and latency costs, whereas broadcast communication involves all processors and incurs higher communication overheads due to the unnecessary data and message transfer. The redundant computations in CAGNET are also visible in its higher local computation times. Moreover, better optimizations are achieved by HP than GP, which is evident from the communication time of GP being 1.7x higher and CAGNET being 8.3x higher than that of HP on 512 processors. HP shows between 2.4x to 3x better communication efficiency over RP from low processor counts to high, while the communication benefit of GP over RP drops slightly from 2.7x to 1.8x.

Scalability for Deeper Networks. Figure 4b shows speedup performance of HP, GP, and RP when varying the number of layers and the dimensionality of features. The number of dimensions is chosen as $d = 50$ and $d = 100$, and the number of layers is increased from 2 to 8. The speedup is computed by dividing the running time of DGL by the running time of HP, GP, or RP respectively under the same GCN configuration. When the number of layers increases and d is kept constant, there is no loss of speedup in any algorithm, and speedup in fact increases for HP. The speedups decrease as d increases because of the rise in total communication volume, which reduces the parallelization efficiency. For example, the speedup of HP decreases approximately from 40x to 17x when the number of features is increased from $d = 50$ to $d = 100$, for an 8-layer GCN. On the other hand, the performance of HP increases from approximately 28x to 40x when the number of layers is increased from 3 to 8, for $d = 100$. We observe the same behavior across different datasets, on both CPU and GPU versions of our algorithm, and present the plots on `roadNet-CA` for 512 CPUs.

Predictive Performance. We also examine the effect on predictive performance of the GCN model when parallelized using our training algorithm. We use the `Cora` dataset since our large-scale networks do not have training labels. We run the parallel training algorithm for 30 epochs on up to 27 GPUs, and compare their accuracy performance with the serial training algorithm. Figure 4c shows that the parallel training algorithm does not have any negative impact on the accuracy performance, with approximately 75% accuracy achieved in all settings.

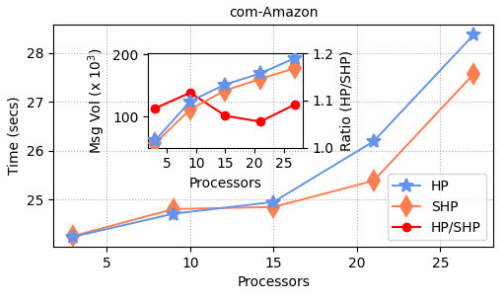


Figure 5: Performance comparisons for mini-batch training. Running time and communication volume ("Msg Vol") with HP and SHP on `com-Amazon` for $P = 3$ to $P = 27$ GPUs.

Table 3: Performance comparison with HP and RP ($d = 1, 2, 5$) on `ogbn-Papers100M` for $P = 27$ GPUs.

Partitioning model	Running time (secs) $d=1$	$d=2$	$d=5$	Communication volume
HP	24.46	25.00	29.73	1.2 billion
RP	34.70	42.88	65.14	13 billion

Stochastic Hypergraph Model. Figure 5 shows the relative performance improvement of stochastic hypergraph model (SHP) over HP in mini-batch training. 10K random mini-batches of size 20K vertices are generated. The total communication volume they induce under partitionings obtained by HP and SHP on `com-Amazon` graph using GPU are measured. We set $\theta = 0.1$ and $\delta = 0.5$ to run SHP and we set the same maximum imbalance ratio ($\epsilon = 0.01$) for both SHP and HP. In the figure (inset), the relative improvement of SHP over HP (in red along the secondary y-axis) shows that HP induces 10% more communication volume than SHP on average. The performance difference in favor of SHP is even more pronounced at higher processor counts. We see that SHP provides a greater benefit of shorter running time with more processors.

Scalability to Billion-scale Datasets. We also test our algorithm on a billion-scale `ogbn-Papers100M` dataset, which is only feasible when partitioned onto 27 GPUs due to memory limitations. Table 3 shows that our methods is scalable not only to high processor counts but also to very large graphs. RP slows down significantly with increasing dimensionality of features. On the other hand, the communication benefit of HP, reducing communication volume approximately by a factor of 10x, allows it to scale better.

Comparison against SOTA. Our optimizations are proposed for large-scale CPU clusters, since the improvement of point-to-point communication overheads over broadcast is more pronounced in such cases. We nonetheless compare the running time of our GPU implementation (HP) against state-of-the-art distributed GPU systems. All systems use the same GCN architecture and report results on the `Reddit` dataset that is common among them. The reported algorithms (except CAGNET) use methods that affect training and predictive performance, such as caching, vertex replication, and asynchronous parameter updates, whereas HP performs full-batch training. As seen from the results, HP achieves considerable relative performance even on small GPU counts.

Table 4: Comparison of running time (per epoch) on `Reddit`.

Method	Running time (per epoch)	Setup	Reference
HP	0.67	A100*3	-
CAGNET	0.11	V100*4	Fig 1 (c=1) [54]
ROC	1/5 = 0.20	P100*4	Fig 5 [22]
Sancus	97.4/1000 = 0.09	V100*4	Table 4 (SCS-A) [43]
PaGraph	≈ 1.00	1080Ti*1	Fig 9 [34]
Dorylus	162.9/120 = 1.36	V100*2	Fig 5, Table 4 [52]
DGCL	0.15	V100*4	Fig 8(a) [4]

6 CONCLUSION

We proposed a highly parallel algorithm for GCN training on large-scale distributed-memory systems. For scalability, all matrices except parameter matrices are row-wise partitioned between processors. The algorithm achieves further communication cost reduction by capturing the sparsity pattern of the adjacency matrix to perform point-to-point communications, via the use of a sparse matrix partitioning scheme based on an intelligent hypergraph model.

Our solution is scalable on a CPU cluster with MPI backend, with the proposed hypergraph partitioning based approach providing significant speedups. The latency between hidden layers are considerably amortized, allowing deeper GCN models to be trained, and with no impact on accuracy. We also performed experiments on a GPU cluster with NCCL backend which provide useful insights on large scale GNN training. All tested algorithms demonstrated less scalability in GPUs compared to the CPU based versions. We also observed that on some instances CPU implementation runs faster than the GPU implementation, besides being more scalable. To further improve the mini-batch training, we proposed a novel stochastic hypergraph model that successfully captures the randomness of communication operations in parallel mini-batch training and achieves improvements over the hypergraph model.

The proposed algorithm is adaptable to other GNNs by changing only the local computations without requiring any changes in terms of the communication operations, which opens up many future directions of research. Additionally, there is scope for exploring several optimizations in the mini-batch sampling strategy and in the GPU communications. The use of 2D and 3D partitioning schemes is another promising avenue for further research.

ACKNOWLEDGMENTS

Aparajita is supported by the Feuer International Scholarship in Artificial Intelligence. Computing resources used were provided by the Scientific Computing Research Technology Platform at the University of Warwick.

REFERENCES

[1] Ammar Ahmad Awan, Khaled Hamidouche, Akshay Venkatesh, and Dhabaleswar K Panda. 2016. Efficient large message broadcast using NCCL and CUDA-aware MPI for deep learning. In *Proceedings of the 23rd European MPI Users' Group Meeting*. 15–22.

[2] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. 2012. Communication-optimal parallel algorithm for strassen's matrix multiplication. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. 193–204.

[3] Neil Band. 2020. MemFlow: Memory-Aware Distributed Deep Learning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2883–2885.

[4] 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.

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

[6] Ümit V Çatalyürek and Cevdet Aykanat. 2011. Patoh (partitioning tool for hypergraphs). In *Encyclopedia of Parallel Computing*. Springer, 1479–1487.

[7] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and Harold Kuhn. 2006. MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In *Proceedings of the 20th annual international conference on Supercomputing*. 353–360.

[8] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247* (2018).

[9] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1804–1815.

[10] Timothy A Davis. 2019. Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)* 45, 4 (2019), 1–25.

[11] Gunduz Vehbi Demirci and Hakan Ferhatosmanoglu. 2021. Partitioning sparse deep neural networks for scalable training and inference. In *Proceedings of the ACM International Conference on Supercomputing*. 254–265.

[12] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. 2021. GraphScope: a unified engine for big graph processing. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2879–2892.

[13] Wenfei Fan, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. 2022. Application-driven graph partitioning. *The VLDB Journal* (2022), 1–24.

[14] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. 2021. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 676–691.

[15] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. 2011. SystemML: Declarative machine learning on MapReduce. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 231–242.

[16] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. 2018. A study of partitioning policies for graph analytics on large-scale distributed platforms. *Proceedings of the VLDB Endowment* 12, 4 (2018), 321–334.

[17] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. {PowerGraph}: Distributed {Graph-Parallel} Computation on Natural Graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 17–30.

[18] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M Tamer Özsu, Xingfang Wang, and Tianqi Jin. 2014. An experimental comparison of pregel-like graph processing systems. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1047–1058.

[19] 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.

[20] Botong Huang, Shivnath Babu, and Jun Yang. 2013. Cumulon: Optimizing statistical data analysis in the cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1–12.

[21] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[22] 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.

[23] Roozbeh Karimi, David M Koppelman, and Chris J Michael. 2019. GPU road network graph contraction and SSSP query. In *Proceedings of the ACM International Conference on Supercomputing*. 250–260.

[24] George Karypis. 1998. hMETIS 1.5: A hypergraph partitioning package. *http://www. cs. umn. edu/˜ metis* (1998).

[25] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[26] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. 2018. Predict then propagate: Graph neural networks meet personalized pagerank. *arXiv preprint arXiv:1810.05997* (2018).

[27] Seongyun Ko and Wook-Shin Han. 2018. Turbograph++ a scalable and fast graph analytics system. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*. 395–410.

[28] Penporn Koanantakool, Ariful Azad, Aydin Buluç, Dmitriy Morozov, Sang-Yun Oh, Leonid Oliker, and Katherine Yelick. 2016. Communication-avoiding parallel sparse-dense matrix-matrix multiplication. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 842–853.

[29] Adrian Kochsiek and Rainer Gemulla. 2021. Parallel Training of Knowledge Graph Embedding Models: A Comparison of Techniques. *Proceedings of the VLDB Endowment* 15, 3 (nov 2021), 633–645. https://doi.org/10.14778/3494124.3494144

[30] Kelly Kostopoulou, Hang Xu, Aritra Dutta, Xin Li, Alexandros Ntoulas, and Panos Kalnis. 2021. DeepReduce: A Sparse-tensor Communication Framework for Distributed Deep Learning. *arXiv preprint arXiv:2102.03112* (2021).

[31] Arun Kumar, Supun Nakandala, Yuhao Zhang, Side Li, Advitya Gemawat, and Kabir Nagrecha. 2021. Cerebro: A Layered Data Platform for Scalable Deep Learning. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*.

[32] Rakshith Kunchum, Ankur Chaudhry, Aravind Sukumaran-Rajam, Qingpeng Niu, Israt Nisa, and P Sadayappan. 2017. On improving performance of sparse matrix-matrix multiplication on gpus. In *Proceedings of the International Conference on Supercomputing*. 1–11.

[33] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

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

[35] Husong Liu, Shengliang Lu, Xinyu Chen, and Bingsheng He. 2020. G3: when graph neural networks meet parallel graph processing systems on GPUs. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2813–2816.

[36] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment* 8, 3 (2014), 281–292.

[37] 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.

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

[39] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. 2021. HET: scaling out huge embedding model training via cache-enabled distributed framework. *Proceedings of the VLDB Endowment* 15, 2 (2021), 312–320.

[40] Xupeng Miao, Wentao Zhang, Yingxia Shao, Bin Cui, Lei Chen, Ce Zhang, and Jiawei Jiang. 2021. Lasagne: A multi-layer graph convolutional network framework via node-aware deep architecture. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[41] Alessio Micheli. 2009. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks* 20, 3 (2009), 498–511.

[42] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021. Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2087–2100.

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

[44] Md Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. 2021. FusedMM: A Unified SDDMM-SpMM Kernel for Graph Embedding and Graph Neural Networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 256–266.

[45] Mohamed Sarwat, Sameh Elnikety, Yuxiong He, and Mohamed F Mokbel. 2013. Horton+ a distributed system for processing declarative reachability queries over partitioned graphs. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1918–1929.

[46] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. 2022. High-Quality Hypergraph Partitioning. *ACM J. Exp. Algorithmics* (mar 2022). https://doi.org/10.1145/3529090

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

[48] Marco Serafini. 2021. Scalable Graph Neural Network Training: The Case for Sampling. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 68–76.

[49] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 505–516.

[50] David I Shuman, Sunil K Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. 2013. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE signal processing magazine* 30, 3 (2013), 83–98.

[51] Jiawen Sun, Hans Vandierendonck, and Dimitrios S Nikolopoulos. 2017. Graphgrind: Addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*. 1–10.

[52] 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.

[53] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From" think like a vertex" to" think like a graph". *Proceedings of the VLDB Endowment* 7, 3 (2013), 193–204.

[54] Alok Tripathy, Katherine Yelick, and Aydin Buluc. [n.d.]. Reducing Communication in Graph Neural Network Training. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 987–1000.

[55] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[56] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315* (2019).

[57] Wei Wang, Meihui Zhang, Gang Chen, HV Jagadish, Beng Chin Ooi, and Kian-Lee Tan. 2016. Database meets deep learning: Challenges and opportunities. *ACM SIGMOD Record* 45, 2 (2016), 17–22.

[58] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. 2019. Simplifying graph convolutional networks. In *International conference on machine learning*. PMLR, 6861–6871.

[59] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. In *First international workshop on graph data management experiences and systems*. 1–6.

[60] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.

[61] 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.

[62] Shengqi Yang, Xifeng Yan, Bo Zong, and Arijit Khan. 2012. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 517–528.

[63] 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.

[64] Bing Yu, Haoteng Yin, and Zhanxing Zhu. 2017. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. *arXiv preprint arXiv:1709.04875* (2017).

[65] 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. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3125–3137.

[66] Yuhao Zhang, Frank Mcquillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano, and Arun Kumar. 2021. Distributed Deep Learning on Data Systems: A Comparative Analysis of Approaches. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1769–1782.

[67] Xiang Zhao, Chuan Xiao, Xuemin Lin, Qing Liu, and Wenjie Zhang. 2013. A partition-based approach to structure similarity search. *Proceedings of the VLDB Endowment* 7, 3 (2013), 169–180.

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

[69] 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.

[70] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. 2022. Distributed Hybrid CPU and GPU training for Graph Neural Networks on Billion-Scale Heterogeneous Graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4582–4591.

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

[72] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. 2022. TGL: A General Framework for Temporal GNN Training on Billion-Scale Graphs. *Proceedings of the VLDB Endowment* 15, 8 (apr 2022), 1572–1580. https://doi.org/10.14778/3529337.3529342