UC Berkeley
UC Berkeley Previously Published Works

Title
Reducing Communication in Graph Neural Network Training

Permalink
https://escholarship.org/uc/item/1sx4j5dv

ISBN
978-1-7281-9998-6

Authors
Tripathy, Alok
Yelick, Katherine
Buluç, Aydın

Publication Date
2020-11-19

DOI
10.1109/sc41405.2020.00074

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Reducing Communication in Graph Neural Network Training

Alok Tripathy, Katherine Yelick, Aydın Buluç

* Electrical Engineering and Computer Sciences, University of California, Berkeley
† Computational Research Division, Lawrence Berkeley National Laboratory

arXiv:2005.03300v3 [cs.LG] 2 Sep 2020

***Abstract*—Graph Neural Networks (GNNs) are powerful and flexible neural networks that use the naturally sparse connectivity information of the data. GNNs represent this connectivity as sparse matrices, which have lower arithmetic intensity and thus higher communication costs compared to dense matrices, making GNNs harder to scale to high concurrencies than convolutional or fully-connected neural networks.**

We introduce a family of parallel algorithms for training GNNs and show that they can asymptotically reduce communication compared to previous parallel GNN training methods. We implement these algorithms, which are based on 1D, 1.5D, 2D, and 3D sparse-dense matrix multiplication, using torch.distributed on GPU-equipped clusters. Our algorithms optimize communication across the full GNN training pipeline. We train GNNs on over a hundred GPUs on multiple datasets, including a protein network with over a billion edges.

***Index Terms*—Graph neural networks, distributed training, communication-avoiding algorithms**

I. Introduction

Graph Neural Networks (GNNs) [25] are types of neural networks that use the connectivity information that is natural in datasets that can be represented as graphs, such as molecules, transportation and social networks, the power grid, and proteins. The neighborhood connectivity information in GNNs is unrestricted and potentially irregular giving them greater applicability than convolutional neural networks (CNNs), which impose a fixed regular neighborhood structure. GNNs have been successful in many application domains and often take advantage of specialized variations such as recurrent GNNs, spatial-temporal GNNs, graph convolutional networks, and graph autoencoders. High-quality surveys of GNNs describe these variations and their applications in more detail [30], [33]. GNNs are also provably quite powerful and, for example, are known to be equivalent to the powerful Weisfeiler-Lehman algorithm for graph isomorphism when the GNNs' underlying aggregation and combination operators are sufficiently flexible [31].

Two dominant uses of GNNs are *node embedding*, which predicts certain properties of individual vertices in a large graph, and *graph embedding*, which predicts certain properties of whole graphs. The presentation of this work follows the node embedding case but the general techniques we introduce are applicable to the graph embedding case as well.

As with CNNs, the training algorithms for GNNs are based on variations of gradient descent, and typically use the idea of *mini-batching* in which a small set of training samples are evaluated and then used to update the model in a single step. Mini-batching has two purposes. First, it allows the neural neural to train on a smaller memory footprint. Second, it strikes a good balance between achieving high-performance through higher arithmetic intensity and achieving good convergence. Unlike images in a database, vertices of a graph are dependent on each other, which is one reason behind GNNs expressiveness. However in the case of GNNs, this dependency makes it hard to process a mini-batch of vertices. After only a few layers, the chosen mini-batch ends up being dependent on the whole graph. This phenomenon, known as the *neighborhood explosion*, completely nullifies the memory reduction goals.

To overcome *neighborhood explosion*, researchers resort to sophisticated sampling-based algorithms that can help GNN training have a smaller memory footprint by reducing the number of k-hop neighbors considered. Sampling algorithms, however, come with approximation errors. Here, we use the aggregate memory of a cluster or supercomputer to train GNNs without mini-batching, similar to other work that use distributed memory to train GNNs [34], [18]. In particular, ROC [18] showed that (1) full gradient descent can be competitive with mini-batching in terms of performance, and (2) sampling based methods can lead to lower accuracy. We build on this work by presenting distributed algorithms with reduced communication. Our distributed algorithms are general and while presented for full gradient descent, they can be easily modified to operate on a mini-batch setting.

Training of GNNs can be memory limited on single node machines, and we support training on distributed-memory architectures where two primary challenges are communication costs and load balance. This paper primarily focuses on minimizing communication costs for GNN training. Some of the best algorithms presented in this paper, namely the 2D and 3D algorithms, also automatically address load balance through a combination of random vertex permutations and the implicit partitioning of the adjacencies of high-degree vertices.

The primary contribution of our paper is the presentation of parallel GNN training algorithms that reduce communication, which are fundamentally different than existing approaches for GNN training. On P processes, our 2D algorithm, which consumes optimal memory, communicate a factor of $O(\sqrt{P})$ fewer words than commonly utilized vertex-partitioning based approaches. The 3D algorithm we describe reduces the number

of words communicated by another factor of $O(P^{1/6})$ at the expense of higher memory consumption. Finally, our 1.5D algorithm optimizes communication for a given memory footprint, reducing communication volume by a factor of $O(c)$ and latency cost by a factor of $O(c^2)$ at the expense of asymptotically increasing memory footprint by $O(c)$. Our work presents algorithmic recipes to get the fastest GNN implementations at large scale.

Our distributed algorithms can be implemented in any system that allows arbitrary divisions of tensors to processes, such as Mesh-Tensorflow [26]. We opted to use PyTorch for our demonstration due to its ubiquity and excellent support for existing GNN models through PyTorch Geometric. All of our experiments are run on the Summit supercomputer at the Oak Ridge Leadership Computing Facility (OLCF). We used the existing single-node kernel implementations in cuSPARSE that are easily called from PyTorch. Faster implementations of key kernels such as sparse matrix times tall-skinny dense matrix (SpMM) exist [32], [2] and would decrease our overall runtime. Faster single node kernels are equivalent from a relative cost perspective to running on clusters with slower networks; both would make our reduced-communication algorithms more beneficial.

Our current implementations operate on the standard real field, but they can be trivially extended to support arbitrary `aggregate` operations to increase the expressive power of GNNs [31]. For example, many distributed libraries such as Cyclops Tensor Framework [27] and Combinatorial BLAS [8] allow the user to overload scalar addition operations through their semiring interface, which is exactly the neighborhood `aggregate` function when applied to graphs. Finally, we note that while our focus is on GNN training, all of our algorithms are applicable to GNN inference. All of our code is available publicly as the CAGNET (Communication-Avoiding Graph Neural nETwork) package at https://github.com/PASSIONLab/CAGNET.

II. Related Work

Parallelism opportunities in the training of deep neural networks (DNNs) have been studied intensively in the recent years [6]. For DNNs generally, the two broad cases of parallelism are model and data parallelism. Data parallelism replicates the DNN model in the memory of each process and only partitions the data. Data parallelism can be sub-classified into sample (also called batch) and domain parallelism. In the particular case of convolutional neural networks (CNNs), domain parallelism is often referred to as spatial parallelism [12]. Model parallelism, on the other hand, partitions the model explicitly. In the common case, each DNN layer can be partitioned into all processes and layers can be computed in their original order. Alternatively, inter-layer pipeline parallelism can be exploited for certain parts of the computation, but not all. For CNNs, further dimensions of parallelism in the form of filter and channel are exploitable as special cases of model parallelism [13].

This might seem like a daunting list of parallelism opportunities to consider for GNN training. Fortunately, as shown in the next section, GNN training is simply a series of algebraic transformations on sparse and dense matrices. Consequently, one can achieve highly-parallel algorithms without even considering the semantic meaning of the dimensions that are partitioned by the algorithm. Ours is similar to an approach taken in earlier work in parallelizing the training of fully-connected and convolutional neural networks [15]. Differently for GNNs, the issues of sparsity and load balance play a prominent role in performance and scalability. Our paper provides several different classes of algorithms that take advantage of many different parallelism opportunities available. We asymptotically analyze the communication costs of the algorithms we present, as well as potential alternatives.

Existing work in parallel GNN training implement their algorithms in specialized frameworks [18], [22], [34]. This requires practitioners to port their models and code to that framework, which might be impossible given the lack of an ecosystem to rapidly implement different GNN algorithms. We implement our algorithms using Pytorch [24], utilizing torch.distributed and PyTorch Geometric libraries. Given the wide availability and popularity of PyTorch, not to mention the vast set of GNN variants implemented in PyTorch Geometric [14], any practitioner with access to a distributed cluster can easily utilize our algorithms to scale their models.

The other PyTorch based distributed graph embedding libraries we are aware of are PyTorch-BigGraph (PBG) [21] and Deep Graph Library (DGL) [29]. PBG's website explicitly says that it is not for use with models such as graph convolutional networks and deep networks. Consequently, while it presents some interesting ideas, PBG does not seem to have the expressiveness required to implement GNNs. By contrast, our distributed algorithms can be used to implement anything that is supported by PyTorch Geometric, which already implements a vast majority of top GNN models in the literature. On the other hand, DGL is an active large-scale project that provides a convenient graph-based interface instead of exposing sparse matrices to the user, and it automatically fuses operations to avoid unnecessary data movement and computation. Our algorithmic work is complementary and can incorporated into DGL in the future.

The details of data partitioning in various GNN training systems are light. ROC [18] advocates a specialized graph partitioning method, and shows that it scales better than random vertex and edge partitioning. AliGraph [34] mentions that it implements both a graph partitioning based approach and a 2D partitioning approach, but does not give any details or provide communication cost analyses.

III. Background

A. Notation

Table I summarizes the notation used in our paper. There is a unique sparse matrix $\mathbf{A}$ that represents the graph structure but there are L distinct $\mathbf{H}$ and $\mathbf{G}$ matrices, indexed $l = 0 \ldots L-1$,

TABLE I
LIST OF SYMBOLS AND NOTATIONS USED BY OUR ALGORITHM

Symbols and Notations	
Symbol	Description
$\mathbf{A}$	Modified adjacency matrix of graph $(n \times n)$
$\mathbf{H}^l$	Embedding matrix in layer l $(n \times f)$
$\mathbf{W}^l$	Weight matrix in layer l $(f \times f)$
$\mathbf{Y}^l$	Matrix form of $\frac{\partial \mathcal{L}}{\partial W^l_{ij}}$ $(f \times f)$
$\mathbf{Z}^l$	Input matrix to activation function $(n \times f)$
$\mathbf{G}^l$	Matrix form of $\frac{\partial \mathcal{L}}{\partial Z^l_{ij}}$ $(n \times f)$
σ	Activation function
f	Length of feature vector per vertex
f_u	Feature vector for vertex u
L	Total layers in GNN
P	Total number of processes
α	Latency
β	Reciprocal bandwidth

which are embedding matrices and their derivatives, respectively. Finally there are $L-1$ weight matrices $\mathbf{W}$ and $\mathbf{Y}$, indexed $l = 0 \ldots L-2$, because the number of transitions between feature vectors are one less than the number of embedding matrices.

When analyzing communication costs we use the $\alpha - \beta$ model where each message takes a constant α time units latency regardless of its size plus an inverse bandwidth term that takes β time units per word in the message, to reach its destination. Thus, sending a message of k words takes $\alpha + \beta k$ time. In addition, we use $nnz(\mathbf{A})$ when referring to the number of nonzeros in the sparse adjacency matrix $\mathbf{A}$, which is equal to the number of edges in the graph with self loops added. We also use d for the average degree of a vertex in $\mathbf{A}$, i.e. $nnz(\mathbf{A}) = dn$.

B. Graph Neural Networks

Consider a dataset that is represented as a graph $G(V, E)$, such as a protein network, social network, grid transmission network, or the union of tangled high-energy particle tracks. Here, V is the set of vertices (nodes) and E is the set of edges. We can consider the classification of the nodes or the edges. Without loss of generality, we will describe a GNN for node classification. The goal of the so-called *node embedding* problem is to map the nodes of a graph into a low-dimensional embedding space such that the similarity of two nodes $u, v \in V$ is approximated by their similarity in the low-dimensional space $z_u^{\mathsf{T}} z_v$. Here, z_v, which is typically a k-dimensional vector of floating-point values, is the embedding of vertex v. In addition to node and edge classification, GNNs can also be used to classify graphs or perform regression on graphs. In this case, the input would be a set of graphs such as the atomistic structures of a set of molecules.

Let $\mathbf{A}$ be the $n \times n$ sparse adjacency matrix of the graph with added self-connections. The addition of self-connections ensures that each node does not forget its embedding when going from layer i to layer $i+1$. The rows and columns of $\mathbf{A}$ are also often normalized [19], so for an undirected graph one actually uses $\mathbf{D}^{-1/2}(\mathbf{A}+\mathbf{I})\mathbf{D}^{-1/2}$ due to its favorable spectral properties. Here, $\mathbf{I}$ is the identity matrix and $\mathbf{D}$ is a diagonal matrix of modified vertex degrees. To avoid notational burden, we will still refer to this modified adjacency matrix with $\mathbf{A}$. We also distinguish between $\mathbf{A}$ and its transpose $\mathbf{A}^{\mathsf{T}}$ explicitly in order to present a general training algorithm that works for both directed and undirected graphs. $\mathbf{H}^0$ is a dense $n \times d$ matrix of input node features. These features are application dependent attributes on graph nodes. A high-quality embedding can be achieved by using a neural network that uses the topology of the graph. In particular, the GNN forward propagation processes t he input features matrix $\mathbf{H}^{(l)}$ at level l using following simple equation: $\mathbf{H}^{(l)} = \sigma(\mathbf{A}^{\mathsf{T}}\mathbf{H}^{(l-1)}\mathbf{W}^l)$.

Here, $\mathbf{W}^{(l)}$ is the *trainable matrix* that holds the model parameters at the lth level of the neural network, and σ is the activation function such as ReLU. Consequently, the most time consuming operations are the multiplication of a sparse matrix with a dense matrix (SpMM) and dense matrix multiply. Backpropagation also relies on the same computational primitives. We provide backpropagation derivations in Section III-D.

C. Forward Propagation

At each node, the product $\mathbf{A}^{\mathsf{T}}\mathbf{H}^{(l-1)}$ combines the $(i-1)$th feature vectors of its neighbors while the subsequent multiplication with $\mathbf{W}^l$ mixes the features and maps them into the new feature space at the ith level. Finally, nonlinearity is achieved via the $\sigma()$ function on the output.

$$\mathbf{Z}^l = \mathbf{A}^{\mathsf{T}}\mathbf{H}^{(l-1)}\mathbf{W}^l$$
$$\mathbf{H}^l = \sigma(\mathbf{Z}^l)$$

D. Backpropagation Derivations

Equation 1: Here we derive the gradient of the loss with respect to $\mathbf{Z}^L$, leveraging the chain rule and that $\mathbf{Z}^l = \sigma(H^l)$.

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial Z^L_{ij}} &= \sum_{u \in V} \sum_{v \in f_u} \frac{\partial \mathcal{L}}{\partial H^L_{uv}} \frac{\partial H^L_{uv}}{\partial Z^L_{ij}} \\
&= \frac{\partial \mathcal{L}}{\partial H^L_{ij}} \frac{\partial H^L_{ij}}{\partial Z^L_{ij}} \qquad \left(\frac{\partial H^L_{uv}}{\partial Z^L_{ij}} = 0 \text{ iff } u \neq i \text{ and } v \neq j\right) \\
&= \frac{\partial \mathcal{L}}{\partial H^L_{ij}} \sigma'(Z^L_{ij})
\end{aligned}$$

$$\boxed{G^L = \frac{\partial \mathcal{L}}{\partial Z^L_{ij}} = \nabla_{H^L}\mathcal{L} \odot \sigma'(Z^L)}$$

Equation 2: Here we derive a recurrence to propagate the gradient backwards through the neural network, leveraging the

chain rule and the lemma stated below.

$$G_{ij}^{l-1} = \frac{\partial \mathcal{L}}{\partial Z_{ij}^{l-1}} = \sum_{u \in V} \sum_{v \in f_u} \frac{\partial \mathcal{L}}{\partial Z_{uv}^{l}} \frac{\partial Z_{uv}^{l}}{\partial Z_{ij}^{l-1}}$$
$$= \sum_{u \in V} \sum_{v \in f_u} \frac{\partial Z_{uv}^{l}}{\partial Z_{ij}^{l-1}} G_{uv}^{l}$$
$$= \sum_{u \in V} \sum_{v \in f_u} W_{jv}^{l} G_{uv}^{l} \sigma'(Z_{ij}^{l-1})$$

(see lemma below)

$$\boxed{\mathbf{G}^{l-1} = \mathbf{A}\mathbf{G}^{l}(\mathbf{W}^{l})^{\mathsf{T}} \odot \sigma'(\mathbf{Z}^{l-1})}$$

Lemma for Equation 2:

$$Z_{uv}^{l} = \sum_{i \in N(u)} \sum_{j \in f_i} H_{ij}^{l-1} W_{jv}^{l}$$

(See forward prop equations)

$$= \sum_{i \in N(u)} \sum_{j \in f_i} \sigma(Z_{ij}^{l-1}) W_{jv}^{l}$$

$$\frac{\partial Z_{uv}^{l}}{\partial Z_{ij}^{l-1}} = W_{jv}^{l} \sigma'(Z_{ij}^{l-1})$$

Equation 3: This final equation represents the gradient $\mathbf{Y}^l$ of the loss with respect to the weights in the network, and this gradient is used in gradient descent.

$$\mathbf{Y}^{l-1} = \left(\frac{\partial \mathcal{L}}{\partial W^{l}}\right)_{ij} = (\mathbf{H}^{l-1})^{\mathsf{T}} \mathbf{A}\mathbf{G}^{l}$$
$$\mathbf{W}^{l-1} = \mathbf{W}^{l-1} - \mathbf{Y}^{l-1}$$

The second step in Equation 3 is simply the gradient descent step. This step does not require communication, so it is not discussed in our analysis in the following section.

IV. Communication Schemes and their Analyses

In this section, we present 1D, 1.5D, 2D, and 3D parallel algorithms for GNN training and analyze their communication costs. Table IV summarizes the matrix partitioning used by these algorithms. The presented communication costs are for one epoch, which is a single pass over the whole dataset.

Ideally, a distributed-memory parallel GNN training algorithm consumes $O(nfL + nnz(\mathbf{A}))$ total memory across all processes. Our 1D and 2D algorithms achieve this bound up to constant factors, while 1.5D and 3D algorithms, whose exact memory consumption is provided in their respective subsections, do not. To avoid expensive transposing, our 1D and 1.5D algorithms stores both $\mathbf{A}$ and $\mathbf{A}^{\mathsf{T}}$ when the input graph is directed. This decision does not increase storage costs asymptotically and is currently immaterial due to almost all GNN-based learning being performed on undirected graphs.

All pseudocodes (Algorithms 1,2,3) take the inputs

1) $\mathbf{A} \in \mathbb{R}^{n \times n}$: sparse adjacency matrix,
2) $\mathbf{H}^{l-1} \in \mathbb{R}^{n \times f^{l-1}}$: dense input activations matrix,
3) $\mathbf{W} \in \mathbb{R}^{f^{l-1} \times f^{l}}$: dense training matrix,

and output $\mathbf{H}^l : \mathbb{R}^{n \times f^l}$: dense output activations matrix.

A. A One-Dimensional (1D) Algorithm

In this regime, matrices $\mathbf{A}^{\mathsf{T}}$ and $\mathbf{H}$ are distributed to processes in block rows, where each process receives n/P consecutive rows. For example, given a matrix $\mathbf{A}^{\mathsf{T}}$, we write $\mathbf{A}_i^{\mathsf{T}} = \mathbf{A}^{\mathsf{T}}(i(n/P) : (i+1)(n/P) - 1, :)$ to denote the block row owned by the ith process, assuming n/P is an integer. To simplify the algorithm description, we use $\mathbf{A}_{ij}^{\mathsf{T}}$ to denote $\mathbf{A}_i^{\mathsf{T}}(:, j(n/P) : (j+1)(n/P)-1)$, the jth block column of $\mathbf{A}_i^{\mathsf{T}}$, although the whole block row is owned by a single process.

$$\mathbf{A}^{\mathsf{T}} = \begin{pmatrix} \mathbf{A}_1^{\mathsf{T}} \\ \vdots \\ \mathbf{A}_p^{\mathsf{T}} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{11}^{\mathsf{T}} & \dots & \mathbf{A}_{1p}^{\mathsf{T}} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{p1}^{\mathsf{T}} & \dots & \mathbf{A}_{pp}^{\mathsf{T}} \end{pmatrix}, \mathbf{H} = \begin{pmatrix} \mathbf{H}_1 \\ \vdots \\ \mathbf{H}_p \end{pmatrix} \tag{1}$$

Let $\mathbf{T}$ be the intermediate product $\mathbf{A}^{\mathsf{T}}\mathbf{H}^{l-1}$. For each process $P(i)$, the computation is:

$$\mathbf{T}_i = \mathbf{T}_i + \mathbf{A}_i^{\mathsf{T}}\mathbf{H} = \mathbf{T}_i + \sum_{j=1}^{p} \mathbf{A}_{ij}^{\mathsf{T}}\mathbf{H}_j$$

The row-wise algorithm forms one row of output at a time, and each process may potentially need to access all of $\mathbf{H}$ to form a single row of $\mathbf{T}$. However, only a portion of $\mathbf{H}$ is locally available at any time in parallel algorithms. The algorithm, thus, performs multiple iterations to fully form one row of $\mathbf{T}$. Algorithm 1 shows the pseudocode of the algorithm.

Algorithm 1 Parallel algorithm for GNN forward propagation, which computes $\mathbf{H}^l \leftarrow \sigma(\mathbf{A}^{\mathsf{T}}\mathbf{H}^{l-1}\mathbf{W}^l)$, using the 1D block row decomposition

```
1: procedure BlockRowFW(A^T, H^{l-1}, W, H^l)
2:     for all processes P(i) in parallel do
3:         for j = 1 to p do
4:             Broadcast(H_j^{l-1})
5:             T_ij ← A_ij^T H_j^{l-1}
6:             Z_i ← T_ij W
7:             H_i^l ← H_i^l + σ(Z_i)
```

1) Equation $\mathbf{Z}^l = \mathbf{A}^{\mathsf{T}}\mathbf{H}^{l-1}\mathbf{W}^l$:

Communication is 1D Block Row: $\mathbf{A}^{\mathsf{T}}$ is partitioned by rows, and $\mathbf{H}^{l-1}$ is partitioned by rows. This yields a 1D Block Row multiplication. $\mathbf{W}^l$ is fully-replicated on each process, and is multiplied with $\mathbf{A}^{\mathsf{T}}\mathbf{H}^{l-1}$ after communication. The first multiplication is essentially a sparse matrix times (tall-skinny) dense matrix, also known as sparse matrix times multiple dense vectors (SpMM).

Our 1D algorithm moves the dense matrix in this SpMM operation using a broadcast. The alternative approach of moving the sparse matrix would yield a similar communication cost in practice because the dense feature matrices in GNNs have approximately the same size (in terms of bytes) as the graphs they are run on. As input and model trends change in the future, a simple heuristic can determine the matrix to be broadcasted in practice, without increasing code complexity.

TABLE II
DATA DISTRIBUTION FOR ALL THE ALGORITHMS CONSIDERED IN OUR PAPER

Matrix	1D Algorithm	1.5D Algorithm	2D Algorithm	3D Algorithm
$\mathbf{A}$	Block row	Block row, replicated c times	Block 2D	Block Split 3D
$\mathbf{A}^\mathsf{T}$	Block row	Block row, replicated c times	Not stored	Not stored
$\mathbf{H}^l$	Block row	Block row, replicated c times	Block 2D	Block Split 3D
$\mathbf{G}^l$	Block row	Block row, replicated c times	Block 2D	Block Split 3D
$\mathbf{W}^l$	Fully-replicated	Fully-replicated	Fully-replicated	Fully-replicated

Alternatively, one could shift matrices as opposed to broadcasting them. Point-to-point communication is still in beta in NCCL, the library we use for communication. The cost of a single broadcast of an m word message to P processes has a lower bound of $O(\alpha \lg P + \beta m)$ [9], but high-level algorithms such as SUMMA [28] can avoid the $\lg P$ factor in the latency term through pipelining. In fact, NCCL uses a pipelined ring algorithm for its broadcasts, which in fact achieves the same link utilization as matrix shifting when the message sizes are large enough to fill the pipeline. Consequently, we will also not spell out the additional $\lg P$ factor for our broadcasts.

The per-process communication cost is thus

$$T_{comm} = \alpha(P-1) + \frac{P-1}{P}\beta\, nf \approx \alpha P + \beta\, nf$$

2) Equation $\mathbf{H}^l = \sigma(\mathbf{Z}^l)$:

No Communication: $\mathbf{H}^l$, the result of activation, is partitioned by rows as is $\mathbf{H}^{l-1}$. No further communication is necessary here to use $\mathbf{H}^l$ in Eq. 1 for layer l.

3) Equation $\mathbf{G}^{l-1} = \mathbf{A}\mathbf{G}^l(\mathbf{W}^l)^\mathsf{T} \odot \sigma'(\mathbf{Z}^{l-1})$:

Communication is 1D Block Row: Because we also partition $\mathbf{A}$ in block rows, the communication pattern and the cost is identical to the forward propagation. The intermediate product $\mathbf{A}\mathbf{G}^l$ is naturally block row partitioned. The last step of multiplying the block row distributed $\mathbf{A}\mathbf{G}^l$ with replicated $\mathbf{W}^l$ to yield a block row distributed $\mathbf{G}^{l-1}$ does not require any communication.

4) Equation $\mathbf{Y}^{l-1} = (\mathbf{H}^{l-1})^\mathsf{T}\mathbf{A}\mathbf{G}^l$:

Communication is (small) 1D Outer Product: Algebraically, there are two matrix multiplications in this step of the backpropagation. However, we can reuse the intermediate product $\mathbf{A}\mathbf{G}^l$ that we computed in the previous equation at the expense of increasing the memory footprint slightly. Then the only task is to multiply $(\mathbf{H}^{l-1})^\mathsf{T}$ and $\mathbf{A}\mathbf{G}^l$, which is a small 1D outer product that requires an all-reduce on low-rank matrices of size $f \times f$. This multiplication has communication cost $T_{comm} = \alpha \lg P + \beta\, f^2$.

5) Total Communication of our 1D Algorithm: Given that the embedding (i.e., feature vector) lengths are different for each layer of the GNN, we use the superscript to denote the length of the feature vector f^l in layer l. This results in the following communication bound.

$$T_{comm} = \sum_{l=1}^{L}\Big(\alpha(\lg P + 2P) + \beta\big(nf^{l-1} + nf^l + f^{l-1}f^l\big)\Big)$$

To reduce clutter, we can consider the "average" feature vector length f, resulting in the simplified formula.

$$T_{comm} = L\left(\alpha(\lg P + 2P) + \beta(2nf + f^2)\right)$$

6) Potential for graph partitioning: Since we are moving dense matrices and keeping the sparse matrix stationary, graph and hypergraph partitioning tools can be applied as pre-processing to heuristically minimize communication. We experimented with graph partitioning to evaluate its potential for us. We ran Metis on the Reddit data, which is described in Section V-A. For 64 processes, Metis' partitions suggested a 72% total communication reduction over random block row distribution. However, the total runtime of our bulk-synchronous algorithm would be dictated by the maximum communication per process, which was only a 29% percent reduction over random 1D block row partitioning. These numbers are actually optimistic and do not take into account the need to perform individualized "request and send" operations for exploiting the graph partitioning results.

For example, instead of relying on more efficient broadcast operations as done in Algorithm 1, each process that owns a part of $\mathbf{A}$ would (depending on its sparsity structure) individually request a subset of rows of $\mathbf{H}$ from its owner during forward propagation. This increases latency as well as the bandwidth costs due to the communication of request indices, and makes it impossible to benefit from collective communication libraries such as NCCL and gloo. Further, given the scale free nature of most graph datasets, graph partitioning is unlikely to produce an asymptotic improvement despite its added computational costs.

B. 1.5D Block Row Algorithm

For 1.5D algorithms [20], processes are organized in a rectangular $P = P/c \times c$ grid. Matrices are, however, partitioned into block rows and columns as done in 1D. The difference between 1D and 1.5D algorithms is that these partitions are now replicated across process rows. For instance, processes across the ith process row $P(i,:)$ collectively store the ith block row of $\mathbf{A}^\mathsf{T}$. Because of this difference, while matrices are partitioned into block rows and columns, there are only P/c such blocks.

$$\mathbf{A}^\mathsf{T} = \begin{pmatrix} \mathbf{A}_1^\mathsf{T} \\ \vdots \\ \mathbf{A}_{p/c}^\mathsf{T} \end{pmatrix} \mathbf{H} = \begin{pmatrix} \mathbf{H}_1 \\ \vdots \\ \mathbf{H}_{p/c} \end{pmatrix} \quad (2)$$

Similar to 1D, each submatrix $\mathbf{A}_i^\mathsf{T}$ is further partitioned in p/c block columns.

Let $\mathbf{T}$ be the intermediate product of $\mathbf{A}^\mathsf{T}\mathbf{H}^{l-1}$. Each process row $P(i,:)$ computes the following:

$$\mathbf{T}_i = \mathbf{T}_i + \mathbf{A}_i^\mathsf{T}\,\mathbf{H} = \mathbf{T}_i + \sum_{j=1}^{p/c} \mathbf{A}_{ij}^\mathsf{T}\,\mathbf{H}_j$$

However, each process computes a subset of the terms in the above summation. These partial sums are then added within process rows with a reduction on $P(i,:)$. If $q = p/c^2$, then the computation done by process $P(i,j)$ is

$$\mathbf{T}_i = \mathbf{T}_i + \mathbf{A}_i^\mathsf{T}\,\mathbf{H} = \mathbf{T}_i + \sum_{k=jq}^{(j+1)q} \mathbf{A}_{ik}^\mathsf{T}\,\mathbf{H}_k \tag{3}$$

These steps are outlined in detail in Algorithm 2. While our pseudocode only outlines the special case where c^2 perfectly divides p, our implementation is more general, and assigns more stages to the last process column if necessary.

Algorithm 2 Block 1.5D algorithm for GNN forward propagation, which computes $\mathbf{H}^l \leftarrow \sigma(\mathbf{A}^\mathsf{T}\mathbf{H}^{l-1}\mathbf{W}^l)$ in parallel. $\mathbf{A}$ and $\mathbf{H}$ are distributed on a $p/c \times c$ process grid, $\mathbf{W}$ is replicated.

1: **procedure** BLOCK1.5DFW($\mathbf{A}^\mathsf{T}, \mathbf{H}^{l-1}, \mathbf{W}, \mathbf{H}^l$)
2: **for** all processes $P(i,j)$ **in parallel do**
3: $s = p/c^2$ ▷ number of stages
4: **for** $k = 0$ to $s-1$ **do**
5: $q = j\,s + k$
6: $\hat{\mathbf{H}}^{l-1} \leftarrow$ BCAST($\mathbf{H}_{qj}^{l-1}$, $P(:,j)$)
7: $\mathbf{Z}^l \leftarrow \mathbf{Z}^l +$ SPMM($\mathbf{A}_{iq}^\mathsf{T}, \hat{\mathbf{H}}^{l-1}$)
8: $\hat{\mathbf{Z}}^l \leftarrow$ ALLREDUCE($\mathbf{Z}^l$, +, $P(i,:)$)
9: $\hat{\mathbf{H}}^l \leftarrow$ GEMM($\mathbf{Z}^l$, $\mathbf{W}^{l-1}$)

1) Equation $\mathbf{Z}^l = \mathbf{A}^\mathsf{T}\mathbf{H}^{l-1}\mathbf{W}^l$:

a) Communication: 1.5D Block Row.: Both $\mathbf{A}^T$ and $\mathbf{H}$ are partitioned by rows in a $P/c \times c$ process grid. We group process rows into c "chunks", with p/c^2 process rows per chunk. These chunks represent the block rows of $\mathbf{H}$ that a particular process column accesses, as per Equation 3. To compute a submatrix of $\mathbf{A}^\mathsf{T}\mathbf{H}$, we broadcast each block row to a process column based on its chunk. If a block row is in chunk i, we broadcast it to ith process column $P(:,i)$. Since there are p/c^2 chunks with c blocks row each, each process participates in only p/c^2 broadcasts. After these iterations of broadcasts complete, each process within a process row has a partial sum for its submatrix. We run an all-reduction to compute the final block row. Note that $\mathbf{W}^l$ is fully-replicated, so we do not need to communicate data to multiply with $\mathbf{W}^l$. The overall communication cost for this equation is

$$T_{comm} = \alpha\Big(\frac{P}{c^2}\lg\frac{P}{c^2}\Big) + \beta\Big(\frac{nf}{c} + \frac{nfc}{P}\Big)$$

2) Equation $\mathbf{H}^l = \sigma(\mathbf{Z}^l)$:

No Communication: $\mathbf{H}^l$, the result of activation, is partitioned by rows as is $\mathbf{H}^{l-1}$. No further communication is necessary here to use $\mathbf{H}^l$ in Eq. 1 for layer l.

3) Equation $\mathbf{G}^{l-1} = \mathbf{A}\mathbf{G}^l(\mathbf{W}^l)^\mathsf{T} \odot \sigma'(\mathbf{Z}^{l-1})$:

Communication: 1.5 Block Row: Recall that $\mathbf{A}$ is partitioned by rows and stored separately from $\mathbf{A}^\mathsf{T}$ if graph is directed. $\mathbf{G}$ is also partitioned by rows. Hence, we can apply the same 1.5D algorithm used in Equation 1. We also need to account for $\sigma'(\mathbf{Z}^{l-1})$. Recall that, as in Equation 2, this step requires no communication as $\mathbf{Z}^{l-1}$ is partitioned by rows. The communication cost for this equation is

$$T_{comm} = \alpha\Big(\frac{P}{c^2}\lg\frac{P}{c^2}\Big) + \beta\Big(\frac{nf}{c} + \frac{nfc}{P}\Big)$$

4) Equation $\mathbf{Y} = (\mathbf{H}^{l-1})^\mathsf{T}\mathbf{A}\mathbf{G}^l$:

Communication: (small) 1.5D Outer Product: We store the intermediate product $\mathbf{A}\mathbf{G}^l$ that was computed in the previous step and reuse it here. Multiplying $(\mathbf{H}^{l-1})^\mathsf{T}$ with $\mathbf{A}\mathbf{G}^l$ is a dense 1.5D Outer Product on two matrices with nf elements, resulting in a small $f \times f$ output. Because of the small output, this outer product is neither compute nor memory-intensive. The resulting communication cost is:

$$T_{comm} = \alpha\Big(\lg\frac{P}{c}\Big) + \beta(f^2)$$

5) Total Communication: Ignoring $\lg P$ latency terms and f^2 bandwidth terms, we have a total communication cost of

$$T_{comm} = \sum_{l=1}^{L}\Big(\alpha\Big(2\frac{P}{c^2}\log\frac{P}{c^2}\Big) + \beta\Big(\frac{2nf}{c} + \frac{2nfc}{P}\Big)\Big)$$

While our 1D algorithm did not scale with increasing process counts, we see here that our 1.5D algorithm scales in proportion to the harmonic mean of P/c and c. This performance benefit, however, comes at a memory cost. The input for the 1.5D algorithm is replicated, evidenced by multiple processes storing the same data across process rows. Formally, the per-process memory and the total memory are

$$M_{proc}^{1.5D} = L\Big(\frac{nnz(\mathbf{A})}{p/c} + \frac{nf}{p/c}\Big)$$
$$M_{total}^{1.5D} = Lc(nnz(\mathbf{A}) + nf)$$

The total input memory used by the 1.5D algorithm is c times more than the input for 1D. This is not negligible, as GNN models tend to be big. In addition, GPUs tend to have less memory, making input replication prohibitive.

C. Block Two-Dimensional (2D) Algorithms

Processes are logically organized on a square $P = P_r \times P_c$ mesh, indexed by their row and column indices so that the (i,j)th process is denoted by $P(i,j)$. Matrices are assigned to process according to a 2D block decomposition. For a given $n \times m$ matrix, each process gets a submatrix of dimensions $n/P_r \times m/P_c$ in its local memory. For example, $\mathbf{A}^\mathsf{T}$ is

partitioned as shown below and $\mathbf{A}_{ij}^{\mathsf{T}}$ is assigned to process $P(i, j)$.

$$\mathbf{A}^{\mathsf{T}} = \begin{pmatrix} \mathbf{A}_{11}^{\mathsf{T}} & \dots & \mathbf{A}_{1p_c}^{\mathsf{T}} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{p_r1}^{\mathsf{T}} & \dots & \mathbf{A}_{p_rp_c}^{\mathsf{T}} \end{pmatrix} \quad (4)$$

The pseudocode for the general rectangular case is listed in Algorithm 3. The first phase of the algorithm is based on a variant of the Scalable Universal Matrix Multiplication (SUMMA) algorithm [28]. When $\text{BCAST}(\mathbf{A}_{ic}, P(i, :))$ is called, the owner of $\mathbf{A}_{ic}$ becomes the root and broadcasts its submatrix to all the processes along the ith process row. Symmetrically, $\text{BCAST}(\mathbf{H}_{rj}, P(:, j))$ means that whomever is the owner of $\mathbf{H}_{rj}$ broadcasts its submatrix along the jth process column.

Variables *lcols* and *lrows*, which are significant only at the broadcasting processes, are the local column and row ranges for matrices that are to be broadcast. Here we used the colon notation where the range $i, i+1, \dots, i+k$ is denoted by $(i : i+k)$ and empty colon $(:)$ specifies the full possible range. For example, $M(:, j)$ denotes the jth column, and $M(1 : k, :)$ denotes the first k rows of any two-dimensional object M. The SPMM and GEMM are local sparse-dense and dense-dense matrix multiplications, respectively.

Algorithm 3 Block 2D algorithm for GNN forward propagation, which computes $\mathbf{H}^l \leftarrow \sigma(\mathbf{A}^{\mathsf{T}}\mathbf{H}^{l-1}\mathbf{W}^l)$ in parallel. $\mathbf{A}$ and $\mathbf{H}$ are distributed on a $p_r \times p_c$ process grid, $\mathbf{W}$ is replicated. Blocking parameter b is required to evenly divide f^{l-1}/p_r and f^{l-1}/p_c.

1: **procedure** BLOCK2DFW($\mathbf{A}^{\mathsf{T}}, \mathbf{H}^{l-1}, \mathbf{W}, \mathbf{H}^l$)
2: **for** all processes $P(i, j)$ **in parallel do**
3: **for** $q = 1$ to f^{l-1}/b **do** ▷ 1st SUMMA phase
4: $c = (qb)/p_c$ ▷ broadcasting process column
5: $r = (qb)/p_r$ ▷ broadcasting process row
6: $actv = (qb : (q+1)b)$ ▷ active columns/rows
7: $lcols = actv \pmod{(n/p_c)}$
8: $lrows = actv \pmod{(n/p_r)}$
9: $\hat{\mathbf{A}}^{\mathsf{T}} \leftarrow \text{BCAST}(\mathbf{A}_{ic}^{\mathsf{T}}(:, lcols), P(i, :))$
10: $\hat{\mathbf{H}}^{l-1} \leftarrow \text{BCAST}(\mathbf{H}_{rj}^{l-1}(lrows, :), P(:, j))$
11: $\mathbf{T}_{ij} \leftarrow \mathbf{T}_{ij} + \text{SPMM}(\hat{\mathbf{A}}^{\mathsf{T}}, \hat{\mathbf{H}}^{l-1})$
12: **for** $q = 1$ to f^{l-1}/b **do** ▷ 2nd phase
13: $c = (qb)/p_c$
14: $actv = (qb : (q+1)b)$
15: $lcols = actv \pmod{(n/p_c)}$
16: $\hat{\mathbf{T}} \leftarrow \text{BCAST}(\mathbf{T}_{ic}(:, lcols), P(i, :))$
17: $colrange = (jf^l/p_c + 1 : (j+1)f^l/p_c)$
18: $\mathbf{H}_{ij}^l \leftarrow \mathbf{H}_{ij}^l + \text{GEMM}(\hat{\mathbf{T}}, \mathbf{W}(actv, colrange))$

We start by analyzing the special $p_r = p_c = \sqrt{p}$ case to give the intuition. For each process $P(i, j)$, the computation of the intermediate product $\mathbf{T} = \mathbf{A}^{\mathsf{T}}\mathbf{H}^{l-1}$ is:

$$\mathbf{T}_{ij} = \sum_{k=1}^{\sqrt{p}} \mathbf{A}_{ik}^{\mathsf{T}} \mathbf{H}_{kj}$$

1) Equation $\mathbf{Z}^l = \mathbf{A}^{\mathsf{T}}\mathbf{H}^{l-1}\mathbf{W}^l$*:*

Communication: 2D SUMMA SpMM + partial SUMMA: Both $\mathbf{A}^T$ and $\mathbf{H}^{l-1}$ are partitioned into a $\sqrt{P} \times \sqrt{P}$ process grid. This yields a 2D multiplication which we can do with an optimized SUMMA algorithm. To compute a submatrix of $\mathbf{A}^T\mathbf{H}^{l-1}$, each process in the submatrix's row must broadcast their $\mathbf{A}$ and each submatrix's column must broadcast their $\mathbf{H}^{l-1}$. $\mathbf{W}^l$ is fully-replicated on each process, and is multiplied with $\mathbf{T} = \mathbf{A}^T\mathbf{H}^{l-1}$ after communication. However, this also requires communicating $n \times f$ sized $\mathbf{T}$ along the process row, something we label as "partial SUMMA".

$$T_{comm} = \alpha 2\sqrt{P} + \beta\Big(\frac{nnz(\mathbf{A})}{\sqrt{P}} + \frac{2nf}{\sqrt{P}}\Big)$$

2) Equation $\mathbf{H}^l = \sigma(\mathbf{Z}^l)$*:*

Communication: All-Gather: $\mathbf{H}^l$ is partitioned in a 2D process grid. When σ is element-wise, no communication is needed. However, when σ is not element-wise, in particular for `log_softmax`, each process needs to broadcast its $\mathbf{H}^{l-1}$ with the entire process row.

$$T_{comm} = \alpha \lg P + \beta \frac{nf}{\sqrt{P}}$$

3) Equation $\mathbf{G}^{l-1} = \mathbf{A}\mathbf{G}^l(\mathbf{W}^l)^{\mathsf{T}} \odot \sigma'(\mathbf{Z}^{l-1})$*:*

Communication: 2D SUMMA SpMM + partial SUMMA + All-Gather: $\mathbf{A}$ and $\mathbf{G}^l$ are partitioned in a 2D process grid, and results in the same communication pattern as Equation 1. We also need to account for $\sigma'(\mathbf{Z}^{l-1})$. Recall that this needs communication when σ is not elementwise. This has the same communication pattern as Equation 2. The communication cost is

$$T_{comm} = \alpha(2\sqrt{P} + \lg P) + \beta\Big(\frac{nnz(\mathbf{A})}{\sqrt{P}} + \frac{3nf}{\sqrt{P}}\Big).$$

4) Equation $\mathbf{Y} = (\mathbf{H}^{l-1})^{\mathsf{T}}\mathbf{A}\mathbf{G}^l$*:*

Communication: 2D dense SUMMA + All-Gather: Strictly speaking, there are two matrix multiplications here. The first is multiplying $\mathbf{A}$ and $\mathbf{G}^l$, which would have been a 2D SUMMA SpMM. However, we can reuse the same intermediate product from the previous equation, as we have done in the case of 1D Block Row algorithm. This increases storage by an additive nf/P term on each process. The intermediate product $\mathbf{A}\mathbf{G}^l$ is already partitioned on a $\sqrt{P} \times \sqrt{P}$ process grid. The second multiplication, which is the only multiplication we have to pay for computing this equation, is between $(\mathbf{H}^{l-1})^{\mathsf{T}}$ and the previously saved intermediate product $\mathbf{A}\mathbf{G}^l$. This is a 2D dense SUMMA on two matrices with nf elements, resulting in a small $f \times f$ output. The final allgather is to keep $\mathbf{Y}$ replicated. Overall, the communication cost due to this equation is

$$T_{comm} = \alpha(\sqrt{P} + \lg P) + \beta\Big(\frac{2nf}{\sqrt{P}} + f^2\Big).$$

5) Total Communication:

$$= \sum_{l=1}^{L} \Big(\alpha(5\sqrt{P} + 3\lg P) + \beta\big(\frac{8nf^l}{\sqrt{P}} + \frac{2nnz(\mathbf{A})}{\sqrt{P}} + (f^l)^2\big)\Big)$$

$$\approx L\Big(\alpha(5\sqrt{P} + 3\lg P) + \beta\big(\frac{8nf}{\sqrt{P}} + \frac{2nnz(\mathbf{A})}{\sqrt{P}} + f^2\big)\Big)$$

The communication volume scales with $\sqrt{p}$. However, the constants in the 2D algorithm are significantly larger than the constants in 1D and 1.5D algorithms. The latency cost of the 2D algorithm is asymptotically better than the 1D algorithm but worse than the 1.5D algorithm when $c{=}\sqrt{p}$. Finally, the 2D algorithm also moves the sparse matrix, creating additional costs. This is not a problem asymptotically as long as $nnz(A) = O(nf)$ but can be a bottleneck if the graph is larger than the aggregate embedding size across all vertices.

6) The Rectangular Grid Case: When the process grid is non-square, the 2D SUMMA algorithm is still well-defined and relatively easy to implement. Consider the forward propagation equation where $\mathbf{A}^\mathsf{T}$ and $\mathbf{H}$ are 2D block partitioned on a $P_r \times P_c$ grid. Each process needs to access $(1/P_r)$th of $\mathbf{A}^\mathsf{T}$ and $(1/P_c)$th of $\mathbf{H}$ to form its own $n/P_r \times f/P_c$ piece of the intermediate output $\mathbf{T} = \mathbf{A}^\mathsf{T}\mathbf{H}$. However, we now need to consider the communication due to the 2nd phase of Algorithm 3, which communicates this $\mathbf{T}$ matrix along the process row. The forward propagation communication cost becomes:

$$T_{comm} = \alpha\,\text{gcf}(P_r, P_c) + \beta\Big(\frac{nnz(\mathbf{A})}{P_r} + \frac{nf}{P_c} + \frac{nf}{P_r}\Big),$$

where gcf denotes greatest common factor. This suggests that if the average vertex degree of the graph is significantly larger than the feature vector length, then there are potential savings is terms of sparse matrix communication by increasing the P_r/P_c ratio. However, this comes at the expense of increasing the sum of other two terms, which correspond to dense matrix communication. This is because the sum of two numbers whose product is fixed is minimized when those numbers are equal, or put differently, square has the smallest perimeter of all rectangles of a given area. Consequently, given the unclear benefit to cost ratio of using non-square grids, our implementations focus on square grids in this work.

D. Block 3D algorithms

For ease of presentation, let us assume that processes are logically organized on a 3D $\sqrt[3]{P} \times \sqrt[3]{P} \times \sqrt[3]{P}$ mesh, indexed by three indices, though in general each of the three process dimensions can be different. Our matrix to process mesh assignment follows the Split-3D-SpGEMM approach of Azad et al. [3]. We call our variation Split-3D-SpMM, because the primary GNN operation is SpMM as opposed to SpGEMM. Each 2D plane in this 3D process mesh is called a "layer".

Considering a single SpMM such as the $\mathbf{A}\mathbf{H}^{l-1}$ in forward propagation, we note that two input matrices are split differently. After 3D distribution, each local submatrix $\mathbf{A}_{ijk}$ of $\mathbf{A}$ is of dimensions $n/\sqrt[3]{P} \times n/\sqrt[3]{P}^2$. That means the number of rows of each $\mathbf{A}_{ijk}$ is $\sqrt[3]{P}$ times its number of columns. By contrast, $\mathbf{H}$ is split along the rows across layers and each local piece $\mathbf{H}_{ijk}$ is of dimensions $n/\sqrt[3]{P}^2 \times f/\sqrt[3]{P}$. This data distribution choice makes each $\mathbf{H}^{l-1}_{ijk}$ shorter and fatter than 2D distribution makes, potentially alleviating some of the issues with local SpMM scaling we observe with our 2D implementation.

We note that each process only holds $(1/P)$th of the input matrices in Split-3D-SpMM, so there is no replication at the input stage. The replication happens in intermediate stages, as we explain in detail below.

1) Equation $\mathbf{Z}^l = \mathbf{A}^\mathsf{T}\mathbf{H}^{l-1}\mathbf{W}^l$:

Communication: One full and one partial Split-3D-SpMM: The easiest way to think about a 3D multiplication algorithm is to consider it as independent 2D algorithms executing at each layer, followed by a reduction. To compute a submatrix of $\mathbf{A}^\mathsf{T}\mathbf{H}^{l-1}$, each submatrix in $\mathbf{A}^\mathsf{T}_{:jk}$ broadcasts itself to the rest of the process row, and each submatrix $\mathbf{H}_{i:k}$ broadcasts itself to the rest of the process column. In each 2D SUMMA iteration, each process on a given layer receives a submatrix from $\mathbf{A}^T$ and a submatrix from $\mathbf{H}^{l-1}$, multiplies them, and adds them to a running sum.

Once these 2D SUMMA iterations that have been executing independently at each layer complete, processes have partial sums for $\mathbf{A}^T\mathbf{H}^{l-1}$ that need to be reduced across the third process dimension (also called a "fiber"). The partial sums after the 2D SUMMA iterations complete are $n/\sqrt[3]{P} \times f/\sqrt[3]{P}$ dense matrices, with $nf/P^{2/3}$ elements each. These partial sums are then reduce-scattered along the fiber dimension to get the product $\mathbf{A}^\mathsf{T}\mathbf{H}^{l-1}$ in a Block Split 3D format. This results in the following communication cost just to form $\mathbf{A}^\mathsf{T}\mathbf{H}^{l-1}$:

$$T_{comm} = \alpha(P^{1/3} + \lg P) + \beta\Big(\frac{nnz(\mathbf{A})}{P^{2/3}} + \frac{2nf}{P^{2/3}}\Big)$$

Note that the aggregate memory consumption over all processes prior to the fiber reduction would be $P\big(nf/P^{2/3}\big) = P^{1/3}nf$, where $P^{1/3}$ is the well-known memory replication cost factor of 3D algorithms [1], [5].

We now need to multiply this intermediate product $\mathbf{A}^\mathsf{T}\mathbf{H}^{l-1}$ with $\mathbf{W}^l$. Similar to the Block 2D case, we will perform a partial 3D dense matrix multiply where the second input matrix does not need to be communicated because it is replicated. The total communication cost, with this partial step added, is:

$$T_{comm} = 2\alpha(P^{1/3} + \lg P) + \beta\Big(\frac{nnz(\mathbf{A})}{P^{2/3}} + \frac{4nf}{P^{2/3}}\Big)$$

2) Equation $\mathbf{H}^l = \sigma(\mathbf{Z}^l)$:

Communication: All-Gather: $\mathbf{H}^l$ is partitioned in a 3D process mesh. When σ is elementwise, no communication is needed. However, when σ is not elementwise, in particular for `log_softmax`, each process needs to broadcast its $\mathbf{H}^l$ with the entire process row within a layer. This is equivalent to an all-gather per layer. No cross-layer or cross-row communication is needed as the output of `log_softmax` for a row of $\mathbf{Z}$ is only dependent on the values within that row.

$$T_{comm} = \alpha \lg P + \beta\Big(\frac{nf}{P^{2/3}}\Big)$$

3) Equation $\mathbf{G}^{l-1} = \mathbf{A}\mathbf{G}^l(\mathbf{W}^l)^\mathsf{T} \odot \sigma'(\mathbf{Z}^{l-1})$:

a) Communication: One full and one partial Split-3D-SpMM, and All-Gather: $\mathbf{A}$ and $\mathbf{G}^l$ are partitioned in a 3D process grid, and results in the same communication pattern as Equation 1. We also need to account for $\sigma'(\mathbf{Z}^l)$. Recall that this needs communication when σ is not elementwise. This has the same communication pattern as Equation 2, hence totalling:

$$T_{comm} = \alpha(2P^{1/3} + 3\lg P) + \beta\Big(\frac{nnz(\mathbf{A})}{P^{2/3}} + \frac{5nf}{P^{2/3}}\Big)$$

4) Equation $\mathbf{Y} = (\mathbf{H}^{l-1})^\mathsf{T}\mathbf{A}\mathbf{G}^l$:

Communication: 3D dense SUMMA + All-Gather: We store the intermediate product $\mathbf{A}\mathbf{G}^l$ that has been computed in the previous step, and reuse it here as we have done in other algorithms. Multiplying $(\mathbf{H}^{l-1})^\mathsf{T}$ with $\mathbf{A}\mathbf{G}^l$ is a dense 3D SUMMA on two matrices with nf elements, resulting in a small $f \times f$ output. Note that each of these inputs are Block Split 3D. As before, the final allgather is to keep $\mathbf{Y}$ replicated. The bandwidth cost of all-gather strictly dominates the bandwidth cost of fiber reduction, so we do not include the cost of fiber reduction in the bandwidth term. The resulting communication cost is:

$$T_{comm} = \alpha 2\lg P + \beta\Big(\frac{2nf}{P^{2/3}} + f^2\Big).$$

5) Total Communication: Ignoring $\lg P$ latency terms that are strictly dominated by the $P^{1/3}$ terms, we have

$$T_{comm} \approx L\Big(\alpha(4P^{1/3}) + \beta\Big(\frac{2nnz(\mathbf{A})}{P^{2/3}} + \frac{12nf}{P^{2/3}}\Big)\Big)$$

Although the 3D algorithm provides an asymptotic reduction in communication costs, it has several disadvantages compared to the 2D algorithm, which are (1) its high constants, (2) its implementation complexity, and (3) its need to do a factor of $\sqrt[3]{p}$ replication in its intermediate stages.

The exact impact of this intermediate replication to the overall memory consumption of GNN training depends on several factors such as (1) the number of layers, (2) the ratio of the maximum number of activations in a layer to the total number of activations, and (3) the ratio of the number of total activations to the number of edges (i.e. $nnz(\mathbf{A})$) in the graph. An implementation of full-batch GNN training that is optimal with respect to memory usage has the following memory footprint:

$$M^*_{total} = nnz(\mathbf{A}) + \sum_{l=1}^{L}\big(nf^l\big) \approx nnz(\mathbf{A}) + nfL$$

The memory consumption of the 2D algorithm asymptotically matches this bound and the only overheads are in the communication buffers. However, the 3D algorithm replicates the activations while processing a layer. The important point is that this replication only impacts the current layer that is being processed, either during forward or backward propagation, because the intermediate matrices are discarded after layer-wise reduction. Suppose the maximum number of activations (e.g., features) is $f^{max} = \max_{l=1}^{L} f^l$, and the layer with the highest number of activations is $k = \arg\max_{l=1}^{L} f^l$. Then the memory footprint of the 3D algorithm is

$$\begin{aligned} M^{3D}_{total} &= nnz(\mathbf{A}) + \sum_{l=1, l\neq k}^{L}\big(nf^l\big) + \sqrt[3]{p}\, nf^{max} \\ &\approx nnz(\mathbf{A}) + nf(L-1) + \sqrt[3]{p}\, nf^{max} \end{aligned}$$

When the network is deep or when the maximum number of activations in a layer is much smaller than the input graph size, this memory overhead is unlikely to be an impediment. However, many existing GNN networks are rather shallow at the moment.

V. Experimental Setup

A. Datasets and Compute Platform

We ran our experiments on two of the largest datasets used in GNN research previously, the Reddit and Amazon datasets. In addition, we use a much larger protein similarity network which pushes the boundaries of large-scale GNN traning. This 'protein' network comes from the data repository of the HipMCL algorithm [4]. It is an induced subgraph, which contains 1/8th of the original vertices, of the sequence similarity network that contained all the isolate genomes from the IMG database at the time. The characteristics of the datasets are documented in Table III. The feature (input) and label (output) embedding lengths of the protein dataset largely follows the literature on this topic [16] where the protein sequences are assumed to be initially processed independently to obtain their 128-length embeddings (via CNNs or LSTMs) before running GNNs on those embeddings. For load balancing, graph vertices are randomly permuted prior to training.

TABLE III
Datasets used in our experiments

Name	Vertices	Edges	Features	Labels
Reddit	232,965	114,848,857	602	41
Amazon	14,249,639	230,788,269	300	24
Protein	8,745,542	2,116,240,124	128	256

We use the same 3-layer GNN architecture presented by Kipf and Welling [19] though deeper and wider networks are certainly possible as done by ROC [18] given the similar performance we achieve.

We verified that our parallel implementation not only achieves the same training accuracy in the same number of epochs as the serial implementations in PyTorch, but it also outputs the same embeddings up to floating point accumulation errors. Consequently, we only provide performance numbers as the accuracy numbers are identical to the serial cases.

B. System details

All of our experiments are run on the Summit supercomputer at ORNL, which has IBM AC922 nodes with 6 NVIDIA V100 GPUs. Each GPU has 16GB of HBM2 memory. Each Summit node has two sockets, each with 1 POWER9 CPU and 3 GPUs each. Within a socket, each GPU is connected to

each other and the host CPU with NVLINK 2.0 with 50 GB/s unidirectional bandwidth. Across sockets, Summit uses the IBM X-bus interconnect with 64 GB/s. Each socket has a 16 GB/s link to the Network Interface Card (NIC). Across nodes, Summit uses dual-rail EDR Infiniband with 25 GB/s node injection bandwidth [23], but each socket has access to half (12.5 GB/s) of this node injection bandwidth.

C. Implementation details

We implement our 3-layer GNN architecture mostly in PyTorch Geometric (PyG) 1.3 [14]. Within PyG, we use torch.distributed with a NCCL backend for our communication primitives. NCCL implements broadcasts and reductions using a ring algorithm, and splits an outgoing message to smaller chunks for pipelining [10]. NCCL collectives have different completion semantics than MPI. For example, when the broadcasting process receives the delivery confirmation of its own data from its peer process, its function call returns without waiting for the completion of the broadcast on subsequent nodes. This relaxed completion combined with pipelining allows broadcasting using NCCL to be as high-performance as using manual shifting when implementing matrix multiplication, provided that the messages can be divided into enough chunks to hide the latency of filling the pipeline.

For our SpMM calls, we separately call cuSPARSE's `csrmm2` function in a C++ extension. We compile our C++ backend with CUDA 10.1. Recall each node on Summit has 6 GPUs. As such, our implementation is single-node multi-GPU when P=4, but multi-node multi-GPU in all other cases. In particular, a P process job allocates $\lceil P/6 \rceil$ nodes so that we utilize all 6 GPUs except for the last node. We only deviate from this setup for exposing the topological limitations of Summit interconnect in Figure 2

For Reddit, we use the input feature vectors and training split used by Hamilton et al. [17] as they are already provided within PyG. For the Amazon and Protein datasets, we randomly generate feature values for simplicity and use the whole graph as the training set. This does not affect performance, and in practice, users could use any values for the feature vectors. We run Reddit and Amazon for 100 epochs and Protein for 10 epochs. We do not report numbers for Amazon on 4 devices or numbers for Protein on 4 or 16 devices as the data does not fit in memory for those configurations. Jia et al. [18] observed the same behavior with PyG. We are unable to compare our results directly with Neugraph or ROC because neither code is currently publicly available.

VI. Results

A. Performance of the 1D and 1.5D Implementations

The performance of 1D (c=1) as well as 1.5D implementations (c>1) are shown in Figure 1. Since our 1D and 1.5D implementations only move the dense matrices, the communication volume is proportional to the product of the number of vertices and the number of features. Due to its small vertex count, GNN training on Reddit dataset is increasingly latency bound at large concurrencies. Consequently when P is large, increasing c directly translates into lower communication costs for Reddit, due to quadratic decrease in latency costs. On a single node, our 1.5D GNN training algorithm achieves more than 20 epochs/sec throughput, higher than all previously published results.

For Amazon and Protein, which are mostly bandwidth bound, our analysis expects communication volume in the broadcast stage to decrease linearly with increasing c. However, our results showed minimal decrease in broadcast time when fully utilizing all 6 GPUs on each Summit node. Diving into the specifics of Summit architecture [23], as explained in Section V-B, we conclude this is due to sharing network injection bandwidth. Recall that each socket on Summit has 3 GPUs and they share the same 12.5 GB/s network injection bandwidth. Also recall that broadcasts in NCCL are implemented using a pipelined ring algorithm. When only a single broadcast is active (c=1 aka 1D), the whole 12.5 GB/s injection bandwidth is used by a single GPU because the last GPU on the ith node is peered with the first GPU on the $(i+1)$th node. When $c = 2$, there are two simultaneous broadcasts happening on two virtual rings. Two GPUs on a socket now has to share the network injection bandwidth to communicate with their peers on the neighboring node. This cuts down the effective available bandwidth by half, so even though the communication volume is reduced to half we do not see any appreciable decrease in broadcast times with increasing c from 1 to 2.

Further increasing c to 4 increases the load node injection to its maximum where all 3 GPUs on the socket compete for 12.5 GB/s injection bandwidth. We confirmed that node injection bandwidth is indeed the bottleneck by running our 1D and 1.5D implementations on 1 GPU/node configuration. Figure 2 shows close-to-linear scaling for broadcast times when c is increased, for the bandwidth-bound datasets Amazon and Protein.

B. Performance of the 2D Implementation

The performance and scaling of our 2D implementation is covered by both Figure 1 (comparing with the 1D and 1.5D implementations) and Figure 3 (comparing with the 3D implementation). Figure 1 covers the process counts p=16, 36, 64, 100 and uses the default 16 middle layer dimension. Figure 3 covers the process counts p=25, 64, 121 and uses 64 as the middle layer dimension. When comparing our 2D and 3D implementations in Figure 3, we used a middle layer dimension of 64 instead of 16 because the 3D algorithm needs to partition dimension 16 across $P^{2/3}$ processes in some cases, which is impossible with P>64. Consequently, the average feature length f is $\approx 10\%$ larger in Figure 3, translating into a marginal runtime difference between Figures 1 and 3 for p= 64. We note that *dbcast* is the total time to broadcast dense matrices in any step of the 2D algorithm and is not restricted to distributed `GEMM` calls. In fact, *dbcast* time is dominated by the time it takes to communicate dense matrices (e.g., $\mathbf{H}$ and $\mathbf{G}$) during distributed SpMM calls.

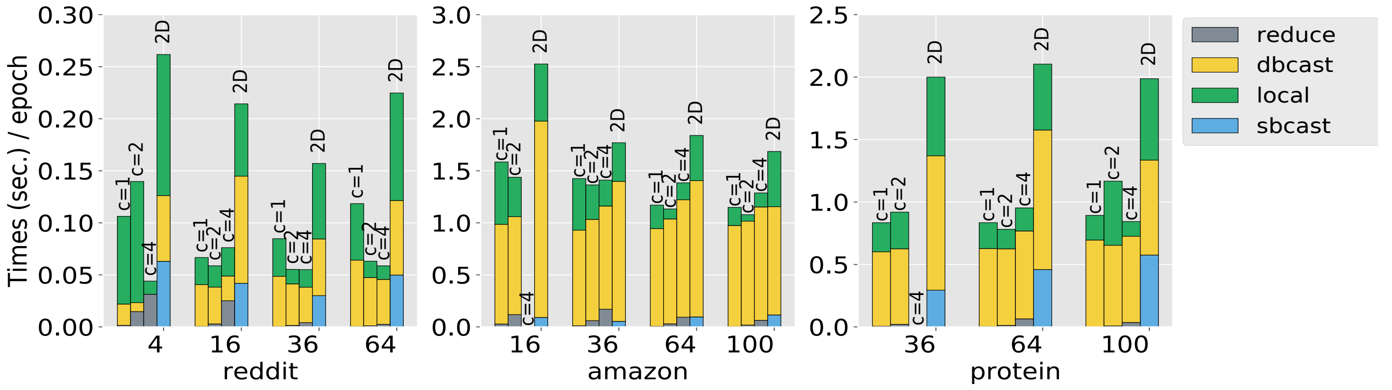

Fig. 1. 1D (c=1), 1.5D (c=2, 4), and 2D performance results when using all 6 GPUs on each node. The x-axis in each subplot is the number of GPUs used. *dbcast* refers to the broadcast of dense embedding matrices, *sbcast* refers to the broadcast of sparse adjacency matrix (only for 2D), *reduce* is the allreduce (only for 1.5D), *local* is the local computation including cuSPARSE SpMM calls, small DGEMM calls, transpose (only for 2D), and sparse matrix assembly after communication (only for 2D). Missing bars for c=4, p=16 on Amazon and c=4, p=36 on Protein means that those runs ran out of memory.

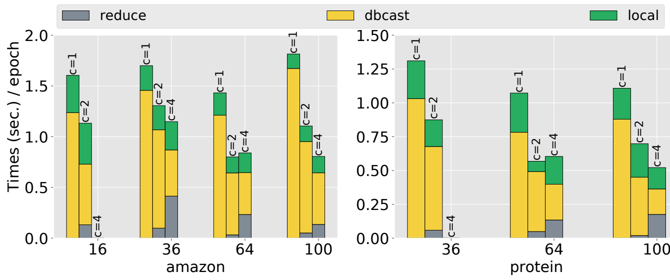

Fig. 2. 1D (c=1), 1.5D (c=2, 4) performance results when only one 1 GPU is used per node. The x-axis for each subplot is the number of GPUs used. *dbcast* refers to the broadcast of dense embedding matrices, *reduce* is the allreduce (only for 1.5D), *local* is the local computation including cuSPARSE SpMM calls, small DGEMM calls

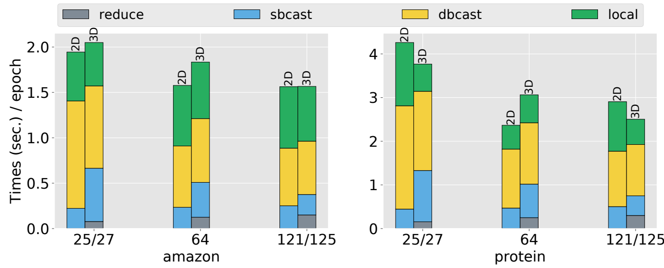

Fig. 3. Runtime of 2D Implementation vs. 3D Implementations across GPU counts. The x-axis for each subplot is the number of GPUs used. 2D was run on perfect square process counts, while 3D was run on perfect cubes. *reduce* is the allreduce (only for 3D), *dbcast* refers to the broadcast of dense embedding matrices, *sbcast* refers to the broadcast of the sparse adjacency matrix, and *local* is the local computation including cuSPARSE SpMM calls, small DGEMM calls, transpose, and sparse matrix assembly after communication.

In terms of scaling with increasing process counts, the 2D algorithm shows noteworthy speedups up until 36 GPUs on all three datasets. Beyond that, Reddit shows slowdowns whereas Amazon and Protein stagnates. For Amazon and Protein, *dbcast* time continues scaling but that is offset by the increase in local computation and/or *sbcast* times. There are two fundamental reasons why local SpMM does not scale with increasing process counts. (1) SpMM performance degrades as the matrix gets sparser. Yang et al. [32] demonstrate that when the average number of nonzeros per row (i.e., degree, $d = nnz/n$) goes down from 62 to 8, the sustained GFlops rates are cut by a factor of 3. They specifically evaluated cuSPARSE's `csrmm2` function, which is the same SpMM function we use, but the performance degradation due to increased sparsity is present for all SpMM implementations they evaluated. The average number of nonzeros per row goes down when a sparse matrix is 2D partitioned across larger device counts. This phenomenon is known as hypersparsity [7] and decreases the average degree of 2D partitioned submatrices by a factor of $\sqrt{p}$. (2) Since the dense matrices (e.g., the activation matrix) are also 2D partitioned, the number of columns in each local dense submatrix also goes down by a factor of $\sqrt{p}$, making the dense matrix "skinnier". The performance degradation at this extremely skinny regime is also well documented [2].

In terms of absolute performance, the 2D algorithm is never faster than the 1.5D algorithm. This might come surprising given the asymptotically better communication scaling per analysis in Section IV-C5. However, even when considering *dbcast* costs alone, the constants in the 2D algorithm are $4\times$ larger than the 1D and 1.5D algorithms. In addition, the 2D algorithm has to communicate the sparse matrix, which has packing and unpacking costs in addition to the costs associated with actually moving the data.

C. Performance of the 3D Implementation

We run both 2D and 3D implementations on 20 epochs for Amazon and 10 epochs for Protein. For 2D, we run on process counts 25, 64, and 121. For 3D, we run on process counts 27, 64, and 125. We specifically choose these counts to permit a fair comparison. We also do not run the 3D algorithm on Reddit as it was already latency-bound and, thus, cannot leverage the benefits of the 3D algorithm.

Overall, we see that the 3D implementation enjoys a consistent speedup on both datasets all the way until 125 GPUs, unlike the 2D algorithm. While the 3D algorithms is slower than the 2D for smaller process counts on Amazon, it matches

the 2D implementation when $P = 121/125$. For Protein, the 3D algorithm generally outperforms 2D. The difference here lies in the computation. With 2D, as discussed above, scaling computation to large process counts is difficult since the nnz per row shrinks as P increases. While in 2D, each row is partitioned across $P^{1/2}$ processes, in 3D they are partitioned across $P^{2/3}$ processes. Thus, the average nnz per row is lower in 3D than in 2D.

For both datasets, 3D communication scales roughly $5-10\%$ better than 2D, to the point where they both perform equally well at $P = 121/125$. While this is a small difference, this trend is consistent with the analysis in Sections IV-C and IV-D. While the bandwidth in 2D scales by a factor of $P^{1/2}$, the bandwidth in 3D scales by $P^{2/3}$. The 3D analysis also has larger constant factors. Coupled together, we expect the 3D implementation to scale only slightly better than the 2D implementation. We expect that, for larger process counts, the 3D implementation will leverage additional scaling to outperform 2D for Amazon.

VII. Conclusions and Future Work

We presented distributed-memory parallel GNN training algorithms that asymptotically reduce communication costs by dividing two or three dimensions (1.5D, 2D, or 3D) of the iteration space across the training pipeline when compared to the commonly used (1D) vertex partitioning methods. We evaluated these algorithms on three datasets that differ in graph size and structure, revealing a set of trade-offs in memory use, local node efficiency, and communication volume, latency (number of messages), and efficiency.

Our experimental results show that even simple variants of our algorithms can be scalable when implemented using off-the-shelf tools such as PyTorch and cuSPARSE. On a flat execution model that uses only a single GPU per node, there are clear benefits to the communication avoiding approach. On the other hand, the memory hierarchy on a six-GPU node leads to some surprising results due to issues such as injection bandwidth limitations, and the anomalies suggest opportunities to improve collective communication through hierarchical algorithms so that compute nodes' injection bandwidth to the network does not become a bottleneck. Overall, we show that the 1.5D algorithm, which can scale with available memory in the second dimension, is the most effective given the implementation environment on the Summit machine. We believe our communication-avoiding algorithms will show their true benefit on better connected architectures such as Perlmutter [11] where each GPU gets its own 25 GB/s injection bandwidth. At that point, new local SpMM implementations whose throughput degrade more gracefully with increasing sparsity will be crucial for scaling the computation part of GNN training.

The memory consumption of GNNs can be high due to the need to store activations generated during forward propagation so they can be used for backpropagation. If full-batch gradient descent is used or if graph sampling does not provide an asymptotic reduction, the memory costs become $O(nfL)$, which is prohibitive for deep networks. Consequently, we envision future work where our distributed training algorithms are carefully combined with sophisticated sampling based methods to achieve the best of worlds.

Acknowledgments

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE 1752814 and by the National Science Foundation under Award No. 1823034. This work is also supported in part by the Advanced Scientific Computing Research (ASCR) Program of the Department of Energy Office of Science under contract No. DE-AC02-05CH11231, and in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] Ramesh C Agarwal, Susanne M Balle, Fred G Gustavson, Mahesh Joshi, and Prasad Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[2] H. Metin Aktulga, Aydın Buluç, Samuel Williams, and Chao Yang. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *Proceedings of the IPDPS*. IEEE Computer Society, 2014.

[3] Ariful Azad, Grey Ballard, Aydın Buluç, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing*, 38(6):C624–C651, 2016.

[4] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydın Buluç. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic Acids Research*, 46(6):e33–e33, 01 2018.

[5] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011.

[6] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[7] Aydin Buluc and John R Gilbert. On the representation and multiplication of hypersparse matrices. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11. IEEE, 2008.

[8] Aydın Buluç and John R Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[9] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[10] NVIDIA Corporation. NCCL: Optimized primitives for collective multi-gpu communication. https://github.com/NVIDIA/nccl, 2020.

[11] Jack Deslippe. Perlmutter - a 2020 pre-exascale gpu-accelerated system for nersc. architecture and early application performance optimization results. GPU Technology Conference, 2019.

[12] Nikoli Dryden, Naoya Maruyama, Tom Benson, Tim Moon, Marc Snir, and Brian Van Essen. Improving strong-scaling of CNN training by exploiting finer-grained parallelism. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 210–220. IEEE, 2019.

[13] Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Marc Snir, and Brian Van Essen. Channel and filter parallelism for large-scale CNN training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–20, 2019.

[14] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[15] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydın Buluç. Integrated model, batch, and domain parallelism in training neural networks. In *SPAA'18: 30th ACM Symposium on Parallelism in Algorithms and Architectures*, 2018.

[16] Vladimir Gligorijevic, P Douglas Renfrew, Tomasz Kosciolek, Julia Koehler Leman, Kyunghyun Cho, Tommi Vatanen, Daniel Berenberg, Bryn C Taylor, Ian M Fisk, Ramnik J Xavier, et al. Structure-based function prediction using graph convolutional networks. *bioRxiv*, page 786236, 2019.

[17] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 1024–1034. Curran Associates, Inc., 2017.

[18] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with ROC. In *Proceedings of Machine Learning and Systems (MLSys)*, pages 187–198. 2020.

[19] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.

[20] Penporn Koanantakool, Ariful Azad, Aydın Buluç, Dmitriy Morozov, Sang-Yun Oh, Leonid Oliker, and Katherine Yelick. Communication-avoiding parallel sparse-dense matrix-matrix multiplication. In *Proceedings of the IPDPS*, 2016.

[21] Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alexander Peysakhovich. PyTorch-BigGraph: A large-scale graph embedding system. In *Proceedings of the 2nd SysML Conference*, 2019.

[22] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. NeuGraph: Parallel deep neural network computation on large graphs. In *USENIX Annual Technical Conference (USENIX ATC 19)*, pages 443–458, Renton, WA, 2019. USENIX Association.

[23] Tom Papatheodore. Summit architecture overview. Introduction to Summit, 2019.

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[25] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

[26] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pages 10414–10423, 2018.

[27] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *27th International Symposium on Parallel and Distributed Processing*, pages 813–824. IEEE, 2013.

[28] Robert A Van De Geijn and Jerrell Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[29] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR*, abs/1909.01315, 2019.

[30] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

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

[32] Carl Yang, Aydın Buluç, and John D Owens. Design principles for sparse matrix multiplication on the GPU. In *European Conference on Parallel Processing*, pages 672–687. Springer, 2018.

[33] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

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