REDUCING COMMUNICATION IN GRAPH NEURAL NETWORK TRAINING

Alok Tripathy¹ Katherine Yelick¹² Aydın Buluç¹²

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, based on 1D, 1.5D, 2D, and 3D distributed sparse-dense matrix multiplication, and show that they can asymptotically reduce communication compared to previous parallel GNN training methods. We train GNNs on over a hundred GPUs on multiple datasets, including a protein network with over a billion edges.

1 INTRODUCTION

Graph Neural Networks (GNNs) (Scarselli et al., 2008) 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. High-quality surveys of GNNs describe them, their variations, and their applications in more detail (Wu et al., 2020; Zhou et al., 2018).

While our techniques are generally applicable, we focus on *node classification*, which predicts labels of individual vertices.

Mini-batching (i.e. iteratively training on a small sets of vertices) for GNNs has encountered a so-called *neighborhood explosion* problem, as a mini-batch of vertices is dependent on other vertices in the graph. Training a minibatch of vertices in a k-layer network requires all k-hop neighbors of the batch of vertices, resulting in a huge memory footprint even for small k. 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 minibatching, similar to other work that use distributed memory to train GNNs (Zhu et al., 2019; Jia et al., 2020). In par-

ticular, ROC (Jia et al., 2020) 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.

The primary contribution of our paper¹ is the presentation of parallel GNN training algorithms that reduce communication, which are fundamentally different than existing approaches for GNN training. In particular, this paper focuses on graph convolutional networks. Each algorithm provably reduces communication volume by factors such as $O(\sqrt{P})$, where *P* is the number of processes. Our work presents algorithmic recipes to get the fastest GNN implementations at large scale. In this shortened version of our paper, we primarily focus on two algorithms (1D and 1.5D)

Our code is available publicly as the CAGNET (Communication-Avoiding Graph Neural nETwork) package at https://github.com/PASSIONLab/CAGNET.

2 RELATED WORK

Our approach treats GNN training strictly as a series of matrix multiplication operations. In this way, we can achieve highly-parallel algorithms without even considering the semantic meaning of the dimensions that are partitioned by the algorithm, similar to an approach taken in earlier work

¹Electrical Engineering and Computer Sciences, University of California, Berkeley ²Computational Research Division, Lawrence Berkeley National Laboratory. Correspondence to: Alok Tripathy <alokt@berkeley.edu>.

Proceedings of the First MLSys Workshop on Graph Neural Networks and Systems (GNNSys'21), San Jose, CA, USA, 2021. Copyright 2021 by the author(s).

¹This is a condensed and slightly updated version of the paper that appeared in SC'20, full version is available as: "Tripathy, Alok, Katherine Yelick, and Aydin Buluc. Reducing Communication in Graph Neural Network Training. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 987-1000"

in parallelizing the training of fully-connected and convolutional neural networks (Gholami et al., 2018).

Existing work in parallel GNN training implement their algorithms in specialized frameworks (Jia et al., 2020; Ma et al., 2019; Zhu et al., 2019). This requires practitioners to port their models and code to that framework. We implement our algorithms using Pytorch (Paszke et al., 2019), 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 Py-Torch Geometric (Fey & Lenssen, 2019), 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) (Lerer et al., 2019) and Deep Graph Library (DGL) (Wang et al., 2019). Our algorithmic work is complementary and can incorporated into DGL in the future.

3 BACKGROUND

3.1 Notation

Table 1 summarizes the notation used in our paper. There is a unique sparse matrix **A** that represents the graph structure but there are L distinct **H** and **G** matrices, indexed l = 0...L - 1, which are embedding matrices and their derivatives, respectively. Finally there are L - 1 weight matrices **W** and **Y**, indexed l = 0...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. $nnz(\mathbf{A}) = dn$.

3.2 Graph Neural Networks

Consider a dataset that is represented as a graph G(V, E). 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.

Let A be the $n \times n$ sparse adjacency matrix of the graph with added self-connections. In this work, we focus on the Graph Convolution Network (GCN) update rule (Kipf & Welling, 2017), but our techniques can be applied to other types of layers as well. The rows and columns of A are normalized in GCNs, so for an undirected graph one actu-

Table 1. List of symbols and notations used by our algorithm

Symbols and Notations					
Symbol	Description				
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 $\frac{\partial \mathcal{L}}{\partial W_{ij}^l}$ $(f \times f)$				
\mathbf{Z}^l	Input matrix to activation function $(n \times f)$				
\mathbf{G}^{l}	Matrix form of $\frac{\partial \mathcal{L}}{\partial Z_{ij}^l}$ $(n \times f)$				
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				

ally 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} . \mathbf{H}^0 is a dense $n \times f$ 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 the 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 *l*th 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.

3.3 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 *i*th level. Finally, nonlinearity is achieved via the $\sigma()$ function on the output.

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

3.4 Backpropagation

We provide the backpropagation equations for the above forward propagation equations, with derivations presented in our full paper (Tripathy et al., 2020). The objective of these equations is to compute gradients of the inputs for each layer $l(\mathbf{G}^{l})$ and for the weights in each layer (\mathbf{Y}^{l}) .

$$\mathbf{G}^{L} = \frac{\partial \mathcal{L}}{\partial \mathbf{Z}_{ij}^{L}} = \nabla_{\mathbf{H}^{L}} \mathcal{L} \odot \sigma'(\mathbf{Z}^{L})$$
$$\mathbf{G}^{l-1} = \mathbf{A}\mathbf{G}^{l}(\mathbf{W}^{l})^{\mathsf{T}} \odot \sigma'(\mathbf{Z}^{l-1})$$
$$\mathbf{Y}^{l-1} = \left(\frac{\partial \mathcal{L}}{\partial W^{l}}\right)_{ij} = (\mathbf{H}^{l-1})^{\mathsf{T}} \mathbf{A}\mathbf{G}^{l}$$

4 PARALLEL ALGORITHMS

In this section, we present 1D, 1.5D parallel algorithms for GNN training and analyze their communication costs. For space, we only present the final communication complexity for our 2D and 3D algorithms. The complete algorithms and communication analyses are in our full paper (Tripathy et al., 2020). The presented communication costs are for one epoch, which is a single pass over the whole dataset.

4.1 A One-Dimensional (1D) Algorithm

In this regime, matrices \mathbf{A}^{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}^{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 *i*th 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 *j*th block column of $\mathbf{A}_i^{\mathsf{T}}$, although the whole block row is owned by a single process.

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

Let **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 **H** to form a single row of **T**. However, only a portion of **H** is locally available at any time in parallel algorithms. The algorithm, thus, performs multiple iterations to fully form one row of **T**. Algorithm 1 shows the pseudocode of the algorithm.

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

Communication is 1D Block Row \mathbf{A}^{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 communica-

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: function BLOCKROWFW($\mathbf{A}^{\mathsf{T}}, \mathbf{H}^{l-1}, \mathbf{W}, \mathbf{H}^{l}$) 2: for all processes P(i) in parallel do 3: for j = 1 to p do 4: BROADCAST(\mathbf{H}_{j}^{l-1}) 5: $\mathbf{T}_{i} \leftarrow \mathbf{T}_{i} + \mathbf{A}_{ij}^{\mathsf{T}}\mathbf{H}_{j}^{l-1}$ 6: $\mathbf{Z}_{i} \leftarrow \mathbf{T}_{i}\mathbf{W}$ 7: $\mathbf{H}_{i}^{l} \leftarrow \mathbf{H}_{i}^{l} + \sigma(\mathbf{Z}_{i})$

tion. The first multiplication is essentially a sparse matrix times (tall-skinny) dense matrix, also known as sparse matrix times multiple dense vectors (SpMM).

The per-process communication cost is thus

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

4.1.2 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*.

4.1.3 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 **A** in block rows, the communication pattern and the cost is identical to the forward propagation. The intermediate product \mathbf{AG}^{l} is naturally block row partitioned. The last step of multiplying the block row distributed \mathbf{AG}^{l} with replicated \mathbf{W}^{l} to yield a block row distributed \mathbf{G}^{l-1} does not require any communication.

4.1.4 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{AG}^{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{AG}^{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 $T_{comm} = \alpha \lg P + \beta f^2$.

4.1.5 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.

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

To reduce clutter, we can consider the "average" feature

vector length f, resulting in the simplified formula.

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

4.2 1.5D Block Row Algorithm

For 1.5D algorithms (Koanantakool et al., 2016), 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 *i*th process row P(i, :) collectively store the *i*th block row of \mathbf{A}^{T} . Because of this difference, while matrices are partitioned into block rows and columns, there are only P/c such blocks.

$$\mathbf{A}^{\mathsf{T}} = \begin{pmatrix} \mathbf{A}_{1}^{\mathsf{I}} \\ \vdots \\ \mathbf{A}_{p/c}^{\mathsf{T}} \end{pmatrix} \mathbf{H} = \begin{pmatrix} \mathbf{H}_{1} \\ \vdots \\ \mathbf{H}_{p/c} \end{pmatrix}$$
(2)

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

Let **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

$$\mathbf{T}_{i} = \mathbf{T}_{i} + \mathbf{A}_{i}^{\mathsf{T}} \mathbf{H} = \mathbf{T}_{i} + \sum_{k=jq}^{(j+1)q} \mathbf{A}_{ik}^{\mathsf{T}} \mathbf{H}_{k} \qquad (3)$$

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. A and H are distributed on a $p/c \times c$ process grid, W is replicated.

1: function Block1.5DFW($\mathbf{A}^{\mathsf{T}}, \mathbf{H}^{l-1}, \mathbf{W}, \mathbf{H}^{l}$) for all processes P(i, j) in parallel do 2: $s = p/c^2$ 3: ▷ number of stages for k = 0 to s - 1 do 4:
$$\begin{split} q &= j \, s + k \\ \hat{\mathbf{H}}^{l-1} &\leftarrow \mathsf{BCAST}(\mathbf{H}_{qj}^{l-1}, P(:, j)) \\ \mathbf{Z}^l &\leftarrow \mathbf{Z}^l + \mathsf{SpMM}(\mathbf{A}_{iq}^\mathsf{T}, \hat{\mathbf{H}}^{l-1}) \end{split}$$
5: 6: 7: $\hat{\mathbf{Z}}^{l} \leftarrow \text{AllReduce}(\mathbf{Z}^{l}, +, P(i, :))$ 8: $\hat{\mathbf{H}}^{l} \leftarrow \text{GEMM}(\mathbf{Z}^{l}, \mathbf{W}^{l-1})$ 9:

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

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 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 *i*th 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

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

4.2.2 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*.

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

Communication: 1.5D Block Row Recall that **A** is partitioned by rows and stored separately from \mathbf{A}^{T} if graph is directed. **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

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

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

Communication: (small) **1.5D Outer Product** We reuse the intermediate product \mathbf{AG}^{l} that was computed in the previous step. Multiplying $(\mathbf{H}^{l-1})^{\mathsf{T}}$ with \mathbf{AG}^{l} is a dense 1.5D Outer Product on two matrices with nf elements, resulting in a small $f \times f$ output. The resulting communication cost is:

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

4.2.5 Total Communication

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

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

Because of replication, we do incur an extra c times memory cost for our 1.5D algorithm compared to our 1D algorithm. However, while our 1D algorithm had no scaling factors, our 1.5D algorithm scales with the harmonic mean of P/c and c.

4.3 Block Two-Dimensional (2D) Algorithms

Thorough communication analysis for the 2D algorithm can be found in our full paper (Tripathy et al., 2020). Total communication volume is:

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

Overall, the communication volume scales with \sqrt{p} , much more than our 1D and 1.5D algorithms. However, the constants in the 2D algorithm are significantly larger than the constants in 1D and 1.5D algorithms.

4.4 Block 3D algorithms

Thorough communication analysis for the 3D algorithm can be found in our full paper (Tripathy et al., 2020). Total communication volume, ignoring $\lg P$ latency terms that are strictly dominated by the $P^{1/3}$ terms, is:

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

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.

5 EXPERIMENTAL SETUP

5.1 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. Details of our datasets are in Table 2.

Table 2. 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 (Kipf & Welling, 2017) though deeper and wider networks are certainly possible. We verified that our parallel implementation achieves the same embeddings up to floating point accumulation errors as serial implementations. Consequently, we only provide performance numbers.

5.2 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 (Papatheodore, 2019), but each socket has access to half (12.5 GB/s) of this node injection bandwidth.

5.3 Implementation details

We implement our 3-layer GNN architecture mostly in Py-Torch Geometric (PyG) 1.3 (Fey & Lenssen, 2019). Within PyG, we use torch.distributed with a NCCL backend for our communication primitives.

For our SpMM calls, we separately call cuSPARSE's csrmm2 function in a C++ extension. We compile our C++ backend with CUDA 10.1.

For Reddit, we use the input feature vectors and training split used by Hamilton et al. (Hamilton et al., 2017) 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.

6 **RESULTS**

6.1 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 **Reducing Communication in Graph Neural Network Training**



Figure 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. Amazon and Protein also ran out of memory on 4 and 16 GPUs, respectively.



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

node. Diving into the specifics of Summit architecture (Papatheodore, 2019), 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 *i*th 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, leading to no 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.

6.2 Performance of the 2D and 3D Implementations

The performance of 2D and 3D algorithms can be found in our full paper (Tripathy et al., 2020).

7 CONCLUSIONS AND FUTURE WORK

We presented distributed GNN training algorithms that asymptotically reduce communication costs by dividing two or three dimensions of the iteration space across the training pipeline. Overall, we show that the 1.5D algorithm is the most effective on the Summit machine. Complete results can be found in our full paper (Tripathy et al., 2020).

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

- Fey, M. and Lenssen, J. E. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.
- Gholami, A., Azad, A., Jin, P., Keutzer, K., and Buluç, A. Integrated model, batch, and domain parallelism in training neural networks. In SPAA'18: 30th ACM Symposium on Parallelism in Algorithms and Architectures, 2018.
- Hamilton, W., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), Advances in Neural Information Processing Systems 30, pp. 1024–1034. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/ 6703-inductive-representation-learning-on-large-graphs. pdf.
- Jia, Z., Lin, S., Gao, M., Zaharia, M., and Aiken, A. Improving the accuracy, scalability, and performance of graph neural networks with ROC. In *Proceedings of Machine Learning and Systems (MLSys)*, pp. 187–198. 2020.
- Kipf, T. N. and Welling, M. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.
- Koanantakool, P., Azad, A., Buluç, A., Morozov, D., Oh, S.-Y., Oliker, L., and Yelick, K. Communicationavoiding parallel sparse-dense matrix-matrix multiplication. In *Proceedings of the IPDPS*, 2016.
- Lerer, A., Wu, L., Shen, J., Lacroix, T., Wehrstedt, L., Bose, A., and Peysakhovich, A. PyTorch-BigGraph: A large-scale graph embedding system. In *Proceedings of the 2nd SysML Conference*, 2019.
- Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., and Dai, Y. NeuGraph: Parallel deep neural network computation on large graphs. In USENIX Annual Technical Conference (USENIX ATC 19), pp. 443–458, Renton, WA, 2019. USENIX Association. ISBN 978-1-939133-03-8.
- Papatheodore, T. Summit architecture overview. Introduction to Summit, 2019. URL https: //www.olcf.ornl.gov/wp-content/uploads/2019/05/ Summit_System_Overview_20190520.pdf.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N.,

Antiga, L., et al. PyTorch: An imperative style, highperformance deep learning library. In *Advances in Neural Information Processing Systems*, pp. 8024–8035, 2019.

- Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.
- Tripathy, A., Yelick, K., and Buluç, A. Reducing communication in graph neural network training. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.
- Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., Li, M., Zhou, J., Huang, Q., Ma, C., Huang, Z., Guo, Q., Zhang, H., Lin, H., Zhao, J., Li, J., Smola, A. J., and Zhang, Z. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR*, abs/1909.01315, 2019. URL *http://arxiv.org/abs/1909.* 01315.
- Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Philip, S. Y. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.
- Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., and Sun, M. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.
- Zhu, R., Zhao, K., Yang, H., Lin, W., Zhou, C., Ai, B., Li, Y., and Zhou, J. AliGraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment*, 12(12):2094–2105, 2019.