

# Sparsity-aware communication for distributed graph neural network training

*Ujjaini Mukhopadhyay  
Katherine A. Yelick, Ed.  
Aydin Buluç, Ed.*



Electrical Engineering and Computer Sciences  
University of California, Berkeley

Technical Report No. UCB/EECS-2023-253

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-253.html>

December 1, 2023

Copyright © 2023, by the author(s).  
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

### Acknowledgement

I would like to thank my advisors, Katherine Yelick and Aydn Buluç, research mentor, Alok Tripathy, and collaborator, Oguz Selvitopi. I am forever grateful for their guidance and inspiration. Furthermore, I'd like to thank my friends and family for their dedicated commitment to supporting and encouraging me.

---

**Sparsity-aware communication for distributed graph neural network training**

by Ujjaini Mukhopadhyay

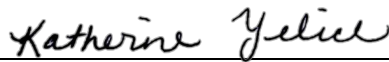
---

**Research Project**

Submitted to the Department of Electrical Engineering and Computer Sciences,  
University of California at Berkeley, in partial satisfaction of the requirements for the  
degree of **Master of Science, Plan II.**

Approval for the Report and Comprehensive Examination:

**Committee:**



---

Professor Katherine Yelick  
Research Advisor

5/11/2023

---

(Date)

\* \* \* \* \*



---

Professor Aydın Buluç  
Second Reader

05/11/2023

---

(Date)

Sparsity-aware communication for distributed graph neural network training

by

Ujjaini Mukhopadhyay

A thesis submitted in partial satisfaction of the

requirements for the degree of

Masters of Sciences

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Katherine Yelick, Chair

Professor Aydın Buluç

Spring 2023

## Abstract

Sparsity-aware communication for distributed graph neural network training

by

Ujjaini Mukhopadhyay

Masters of Sciences in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Katherine Yelick, Chair

Graph neural network (GNN) training has low computational intensity and thus communication costs can limit scalability. Sparse-matrix dense-matrix multiplication (SpMM), where the dense matrix is tall and skinny, is the bottleneck in full-graph training of GNNs. Previous work on distributing SpMM focused on sparsity-oblivious algorithms, where blocks of the matrices are communicated regardless of the sparsity pattern. This provides predictable communication patterns that can be overlapped with computation and maximizes the use of optimized collective communication functions, but it wastes significant bandwidth by communicating unnecessary data.

We present sparsity-aware algorithms that communicate only the parts of matrix blocks that are needed based on the sparsity pattern of the input graph. We couple our sparsity-aware SpMM algorithm with a communication-avoiding (1.5D) approach and a specialized graph partitioning algorithm that minimizes the maximum data volume communicated per process in addition to the total data volume. This addresses communication load imbalance and therefore total cost better than average volume alone. We explore the trade-offs from different graph problems and machine sizes, with the combined optimizations showing up to 14x improvement on 256 GPUs relative to a popular GNN framework based on communication-oblivious SpMM.

# Contents

<b>Contents</b>	<b>i</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>4</b>
2.1 Graph Neural Networks . . . . .	4
2.2 Related Work . . . . .	5
<b>3 Sparsity-Aware SpMM Algorithms</b>	<b>7</b>
3.1 Sparsity-Aware 1D Algorithm . . . . .	7
3.2 Sparsity-Aware 1.5D Algorithm using Point-to-Point Communication . . . . .	10
3.3 Sparsity-Aware 1.5D Algorithm using All-to-All Communication . . . . .	13
<b>4 Graph Partitioning</b>	<b>15</b>
<b>5 Experimental Setup</b>	<b>17</b>
5.1 System Details . . . . .	17
5.2 Implementation Details . . . . .	17
5.3 Datasets . . . . .	18
<b>6 Results</b>	<b>20</b>
6.1 Performance of 1D Sparsity-Aware Algorithm . . . . .	20
6.2 Performance of 1.5D Sparsity-Aware Algorithm . . . . .	23
<b>7 Conclusion</b>	<b>25</b>
<b>Bibliography</b>	<b>26</b>

To Maa, Baba, and Nimama.

# Chapter 1

## Introduction

Graph neural networks (GNNs) have recently demonstrated success for many scientific and engineering problems, such as protein structure prediction in structural biology, track reconstruction in particle physics, and traffic prediction in autonomous driving.

While GNNs typically have fewer layers than CNNs, the graphs can be enormous, making GNN training expensive in both time to solution and memory footprint. Because GNNs represent graphs which encode dependencies, even mini-batch training (only training on a small subset of vertices) can lead to neighborhood-explosion, a phenomenon where the entire graph is processed for one step of training. While there are sampling algorithms that can mitigate the effects of neighborhood explosion, they often introduce approximation error [12, 9, 24]. Thus, in this work we focus on full-graph training. Sparse-matrix tall-skinny-dense-matrix multiplication (SpMM) has been identified as the bottleneck of GNN training [23], especially for the full-graph training case we consider in this paper. Prior work addressed time and memory costs by distributing GNN training across multiple compute nodes [23, 13, 18].

One way to parallelize full-graph GNN training on distributed-memory architectures is to utilize a sparsity-oblivious approach where parts of the matrices that are communicated throughout the algorithm execution is fixed regardless of the pattern of the sparse matrix. A sparsity-oblivious approach has several benefits, such as straightforward generalization from dense matrix algorithms, ability to overlap communication with computation due to predictable communication phases, and the ability to utilize well-optimized collective communication functions that often use less bandwidth than performing point-to-point communication with the same amount of data among the same set of senders and receivers.

In this work, we take the alternative *sparsity-aware* approach that takes advantage of the sparsity of the input graph to avoid transferring parts of the dense matrices. For example in 1D block-row SpMM, if the local sparse matrix stored in process  $P(i)$  has an empty column  $j$ , then the  $j$ th row of the dense matrix need not be communication to process  $P(i)$ . We extend this idea to all GNN training steps. Unfortunately, input graphs do not often come in an order that maximizes this kind of structured sparsity that allows the sparsity-aware algorithm to minimize communication.



Graph and hypergraph partitioning has a long and celebrated history in scientific computing, which is documented in recent surveys [7, 8]. Perhaps one of the most canonical uses of graph/hypergraph partitioning is for sparse matrix-vector multiplication (SpMV) typically within an iterative sparse solver such as conjugate gradient. However, the overhead of partitioning often takes many SpMV iterations to amortize, limiting the widespread use of partitioners in production codes. Partitioning is also utilized to parallelize more heavy-weight kernels such as SpMM and sparse-matrix sparse-matrix multiplication [6, 3], where it is often easier to amortize the overhead of the partitioning as these operations incur much more computation and communication than parallel SpMV.

In contrast to sparse iterative solvers, GNN training has significantly more work to do, easily making up for the cost of partitioning with the reduction in runtime. This is because (1) the main workhorse is SpMM, as opposed to SpMV, (2) each epoch of training performs  $2(L - 1)$  SpMM operations where  $L$  is the number of neural network layers, and (3) it takes hundreds of epochs for GNN training to converge to the desired accuracy, and (4) the sparsity pattern of the matrix that represents the input graph does not change throughout training, so partitioning only needs to be done once.

Graph and hypergraph partitioners by default optimize the total communication volume, while attempting to balance the computational load (e.g., by assigning the same number of nonzeros per processor in the case of sparse matrix partitioning). However, most popular partitioners such as Metis [15] do not provide a mechanism to balance the communication. When each process pair sends and receives different amounts of data with point-to-point messages, this causes a load imbalance in communication where the overall time spent in communication is dominated by the process that sends (or receives) the largest amount of data. As the communication cost of the parallel SpMM is more bandwidth-bound than it is latency-bound (i.e., the sizes of the transferred messages are large, often multiple megabytes) due to large lengths of the feature vectors that need to be communicated, disregarding the load imbalance in communication can be severe for performance and hurt scalability as we demonstrate in this paper.

To alleviate this issue, we rely on a recently-proposed partitioning method [2] that aims to minimize the maximum amount of communicated data besides total amount.

Our contributions in this paper are as follows:

- For all steps of GCN (graph convolutional network) training, we present sparsity-aware algorithms that only communicate parts of dense matrices that will result in nonzero output when multiplied with the sparse matrix.
- We use graph partitioning to amplify the communication-reducing effect of sparsity-aware algorithms. We employ a specialized partitioner that is designed to minimize the maximum amount of communication between pairs of communicating processors, which avoid load balance in communication.
- We demonstrate the generality of my approach by integrating the sparsity-awareness to both 1D and communication-avoiding 1.5D algorithms.

- We demonstrate significant performance improvements for full-graph GNN training, compared to both the sparsity-oblivious approach as well as a sparsity-aware implementation that uses off-the-shelf partitioners that only minimize the total communication volume.

# Chapter 2

## Background

### 2.1 Graph Neural Networks

Graph Neural Networks take as input a graph  $G = (V, E)$ , where  $V$  is the set of vertices and  $E$  is the set of edges. This graph can represent any network structure found in the real world, such as protein-protein interaction, particle tracks, metagenomic read overlaps, social networks, and transportation networks. While GNNs can solve a wide variety of machine learning problems, we focus on *node classification* without loss of generality. In this problem, each vertex takes an associated *feature vector* as input, and a subset of vertices have an associated *label*. The objective of the network is to classify unlabeled vertices in the graph using input features, graph connectivity, and vertex labels. To that end, the neural network maps vertices to low-dimensional embedding vectors such that similar vertices have similar embedding vectors. More concretely, the similarity of two vertices  $u, v \in V$  with embedding vectors  $z_u$  and  $z_v$ , respectively, is simply the dot-product  $z_u^\top z_v$ . The feature vectors for each vertex are represented as a tall-skinny dense matrix  $\mathbf{H} \in \mathbb{R}^{n \times f}$ .

GNNs follow the *message-passing* model, consisting of a **message** step and an **aggregate** step per iteration of training [12]. The **message** step creates a message per edge in the graph. The **aggregate** step takes a vertex  $v$  and combines the messages across all of that  $v$ 's incoming neighbors. The output is multiplied with a parameter weight matrix, and the result is an embedding vector  $z_v$ . Message-passing can be expressed in terms of sparse matrix multiplication as  $\mathbf{Z}^l \leftarrow \mathbf{A}^\top \mathbf{H}^{l-1} \mathbf{W}^l$ . This formulation represents forward propagation in Graph Convolution Networks (GCNs), introduced by Kipf and Welling [16]. In addition, forward propagation includes an activation function  $\mathbf{H}^l \leftarrow \sigma(\mathbf{Z}^l)$ . After several layers of both steps, the network outputs an embedding vector per vertex, after which the network inputs vectors and labels into a loss function for backpropagation. Prior work has shown that the

Table 2.1: 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 $l$ ( $f \times f$ )
$\mathbf{G}^l$	Matrix form of $\frac{\partial \mathcal{L}}{\partial \mathbf{Z}_{ij}^l}$ ( $n \times f$ )
$\mathbf{Z}^l$	input matrix to activation function ( $n \times f$ )
$\sigma$	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
$\alpha$	Latency
$\beta$	Reciprocal bandwidth

operations for GCN training, for both forward and backward propagation, are [23]:

$$\begin{aligned}
 \mathbf{Z}^l &\leftarrow \mathbf{A}^\top \mathbf{H}^{l-1} \mathbf{W}^l \\
 \mathbf{H}^l &\leftarrow \sigma(\mathbf{Z}^l) \\
 \mathbf{G}^{l-1} &\leftarrow \mathbf{A} \mathbf{G}^l (\mathbf{W}^l)^\top \odot \sigma'(\mathbf{Z}^{l-1}) \\
 \mathbf{W}^{l-1} &\leftarrow \mathbf{W}^{l-1} - \mathbf{Y}^{l-1}
 \end{aligned}$$

Here, the first two operations represent forward propagation in GCN training, while the last two compute the input and weight gradients respectively.

## 2.2 Related Work

Literature in parallel training of GNNs is extended, with many contributions from academia, government, and industry research labs. A recent ACM computings survey article focuses on the computational aspects of GNN training, and covers many systems and frameworks [1]. In this paper, we focus on full-graph training (as opposed to mini-batch training) and hence focus on works that support full-graph training. Dorylus [22] and DistGNN [19] are two noteworthy examples of distributed full-graph training on CPUs. ROC [14], CAGNET [23], and BNS-GCN are similarly noteworthy examples of distributed full-graph training on GPUs.

Graph partitioning has been previously employed by DistDGL for reducing communication in GNN training. DistDGL uses a key-value store for storing vertex embedding and focuses on mini-batch training. Their partitioning algorithm also does not minimize the maximum communication volume, potentially resulting in load imbalances in communication.

SpMM, which is the workhorse of full-batch GNN training, has been the target of recent parallelization efforts. Selvitopi et al. [20] presented and investigated 1.5D and 2D

sparsity-oblivious algorithms under bulk-synchronous and asynchronous communication scenarios specifically focusing on a setting with distributed-memory nodes with GPUs. Koanantakool et al. [17] investigated communication costs of a number of 1.5D and 2D distributed SpMM algorithms and gave a recipe which one to utilize according to different factors such as replication factor, relative sparsity, etc. Graph/hypergraph partitioning in the context of SpMM has extensively been studied by Acer et al. [2]. Pointing out that optimizing the single partitioning objective of total volume may often result in poor scalability for this operation, they propose a general framework that relies on multiple constraints to encapsulate various communication cost metrics related to volume throughout the partitioning. Among the partitioners that address multiple communication cost metrics and can show benefits in the parallelization of SpMM are shown by Deveci et al. [11] and Slota et al. [21].

## Chapter 3

# Sparsity-Aware SpMM Algorithms

In this section, we present our 1D and 1.5D sparsity-aware distributed SpMM implementations for full-batch GNN training. These algorithms are adapted from their sparsity-oblivious versions, presented in prior literature [17]. The memory costs for a GNN are dominated by the input graph ( $\mathbf{A}$ ) and node activations ( $\mathbf{H}^0 \dots \mathbf{H}^{L-1}$ ), with a total cost of  $O(\text{nnz}(\mathbf{A}) + nfL)$ . Both 2D and 3D SpMM algorithms exist in the literature as well. However, others show that 2D and 3D algorithms are less performant for full-batch GNN training, so we choose to focus on 1D and 1.5D algorithms.

Sparsity-aware algorithms improve on their sparsity-oblivious versions by communicating less data, and consequently improving runtime [5]. In the sparsity-oblivious SpMM algorithms communication occurs in units of entire block rows of the dense  $\mathbf{H}$  matrix regardless of the matrix structure of  $\mathbf{A}$ . This approach is simple and has predictable communication patterns, and can take advantage of highly optimized logarithmic collective operations. However, these algorithms also unnecessarily communicate rows of  $\mathbf{H}$  that will not be read in the local SpMM computation because they would be multiplied by zeros that are not explicitly stored. We introduce sparsity-aware algorithms that communicate specific rows of  $\mathbf{H}$ , at the cost of linear in process count communication and storing row indices that must be sent.

Both algorithms take as input

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.

### 3.1 Sparsity-Aware 1D Algorithm

Our 1D algorithm assumes both  $\mathbf{A}^\top$  and  $\mathbf{H}$  are distributed in block rows across processes, the same distribution as the sparsity-oblivious algorithms. Each process receives  $n/P$  contiguous

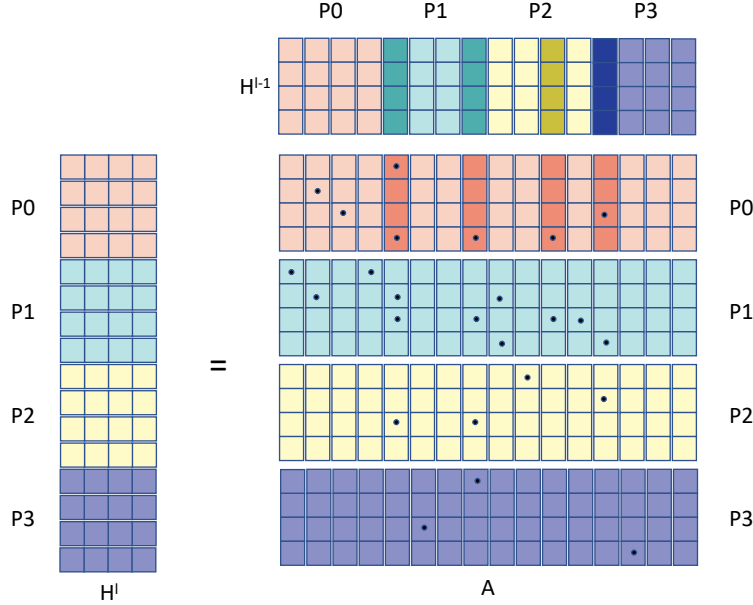


Figure 3.1: The above image shows the specific rows of  $H$  Process 0 requests. The dots represent nonzeros and note that only the subcolumns which contain at least one nonzero are requested. Even if a subcolumn contains more than one nonzero, the corresponding row on  $H$  is only requested once by that process. Here, P0 requests 2 rows from P1, 1 row from P2, and 1 row from P3. The rows that are requested are highlighted on  $H^{l-1}$ . Also note that no diagonal blocks are communicated because processors already own the corresponding block row of  $H^{l-1}$ .

rows of both  $\mathbf{A}^\top$  and  $\mathbf{H}$ . We use  $\mathbf{A}_i^\top$  and  $\mathbf{H}_i$  to refer to the block rows of  $\mathbf{A}^\top$  and  $\mathbf{H}$  that exist on process  $P(i)$ .

$$\mathbf{A}^\top = \begin{pmatrix} \mathbf{A}_1^\top \\ \vdots \\ \mathbf{A}_p^\top \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{11}^\top & \dots & \mathbf{A}_{1p}^\top \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{p1}^\top & \dots & \mathbf{A}_{pp}^\top \end{pmatrix}, \mathbf{H} = \begin{pmatrix} \mathbf{H}_1 \\ \vdots \\ \mathbf{H}_p \end{pmatrix} \quad (3.1)$$

Our sparsity-aware algorithms first has each process  $P(i)$  locally compute  $NnzCols(i, j)$  for  $j = 1 \dots n$ . For a given  $i, j$  pair,  $NnzCols(i, j)$  returns a vector of the nonzero column ids in  $\mathbf{A}_{ij}^\top$ . These nonzero column ids of  $\mathbf{A}_{ij}^\top$  specify the rows of  $\mathbf{H}$  needed to compute  $\mathbf{A}_{ij}^\top \mathbf{H}_j$ .

Let  $\mathbf{Z}^l$  be an intermediate product  $\mathbf{A}^\top \mathbf{H}^{l-1}$ . Like its sparsity-oblivious counterpart, our sparsity-aware 1D algorithm, the computation for  $\mathbf{Z}_i^l$  for process  $P(i)$  is

$$\mathbf{Z}_i^l = \mathbf{Z}_i^l + \mathbf{A}_i^\top \mathbf{H} = \mathbf{Z}_i^l + \sum_{j=1}^p \mathbf{A}_{ij}^\top \mathbf{H}_j$$

---

**Algorithm 1** Sparsity-Aware 1D algorithm for GNN forward propagation.  $\mathbf{A}$  and  $\mathbf{H}^l$  are distributed in block rows across  $p$  processes.  $NnzCols(i, j)$  returns the nonzero column indices in  $\mathbf{A}_{ij}^\top$ , and is computed as a preprocessing step.

---

```

1: procedure BLOCK1DSAFW( $\mathbf{A}, \mathbf{H}^{l-1}, \mathbf{W}, \mathbf{H}^l$ )
2:   for all processes  $P(i)$  in parallel do
3:      $\mathbf{T} \leftarrow [\mathbf{H}[NnzCols(i, 0), :]; \dots; \mathbf{H}[NnzCols(i, P-1), :]]$ 
4:     ALLTOALLV( $\mathbf{T}, P(\cdot)$ )
5:     for  $k = js$  to  $js + s - 1$  do
6:        $\hat{\mathbf{H}}^{l-1}[NNZCOLS(i, k)] = \mathbf{T}[k]$ 
7:        $\mathbf{Z}^l \leftarrow \mathbf{Z}^l + \text{SPMM}(\mathbf{A}^\top, \hat{\mathbf{H}}^{l-1})$ 
8:      $\mathbf{H}^l \leftarrow \text{GEMM}(\mathbf{Z}^l, \mathbf{W})$ 

```

---

Note that process  $P(i)$  stores  $\mathbf{A}_i^\top$  locally, but not  $\mathbf{H}_j$  for  $i \neq j$ . In the sparsity-oblivious algorithm, each process  $P(j)$  would broadcast its entire block row  $\mathbf{H}_j$  to all other processes. Our sparsity-aware 1D algorithm ensures that each process  $P(j)$  only sends the necessary rows of  $\mathbf{H}$  to each other process. Algorithm 1 describes how to compute  $\mathbf{Z}^l$  in more detail.

**Equation**  $\mathbf{Z}^l = \mathbf{A}^\top \mathbf{H}^{l-1} \mathbf{W}^l$

In Algorithm 1, the only communication is an all-to-allv call that exchanges rows of  $\mathbf{H}$  (recall that  $\mathbf{W}^l$  is fully-replicated, so no communication is necessary). A single process will receive data from  $P-1$  other processes, and the time taken by this operation can be upper-bounded by the maximum value of  $NnzCols(i, j)$ , across all pairs of processes  $i, j$ , times  $f$ . To reduce clutter, we use  $cut_P(A)$  to denote this number. This results in the following per-process communication cost with the  $\alpha - \beta$  model.

$$T_{comm} = \alpha(P-1) + (P-1)cut_P(A)f\beta$$

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

No communication is necessary as  $\mathbf{H}^l$  is partitioned by rows. This step is identical to the sparsity-oblivious algorithm.

**Equation**  $\mathbf{G}^{l-1} = \mathbf{A}\mathbf{G}^l(\mathbf{W}^l)^\top \odot \sigma'(\mathbf{Z}^{l-1})$

The communication in this step is identical to the communication in forward propagation (Section 3.1). Recall that, for undirected graphs  $\mathbf{A} = \mathbf{A}^\top$ , and no communication is needed for transposition. For directed graphs, we store both  $\mathbf{A}$  and  $\mathbf{A}^\top$ . In addition,  $\mathbf{A}$ ,  $\mathbf{G}^l$ , and  $\mathbf{Z}^{l-1}$  are all partitioned into block rows. Since  $\mathbf{A}$  is sparse and  $\mathbf{G}^l$  is dense, we use our sparsity-aware SpMM implementation to compute  $\mathbf{A}\mathbf{G}^l$ . The communication pattern is identical to that outlined in Algorithm 1.



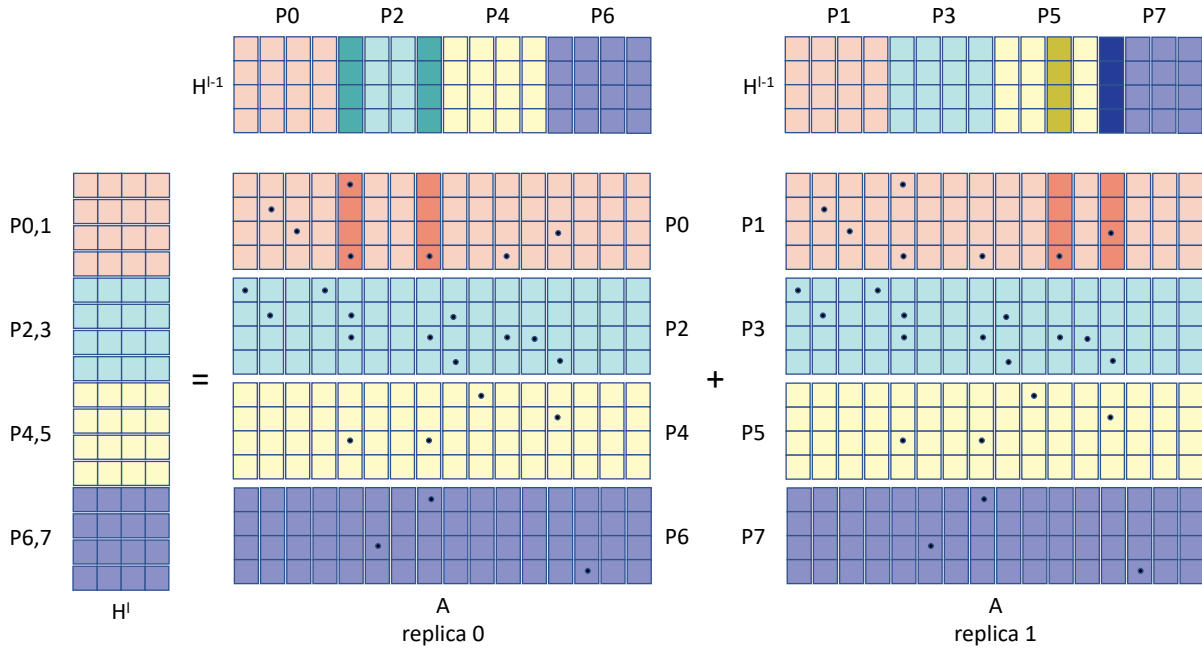


Figure 3.2: The above diagram shows the rows of  $H$  that are requested by  $P_0$  and  $P_1$  in the case of using  $p = 8$  processors and  $c = 2$  replication factor. Note that every block row of  $A$  and  $H$  is replicated  $c$  number of times. As with the 1D algorithm, only the rows corresponding to nonzero subcolumns are requested from  $H$ . Here,  $P_0$  requests 2 rows from  $P_2$  while  $P_1$  requests one row from  $P_5$  and one row from  $P_7$ .

$$\text{Equation } \mathbf{Y}^{l-1} = (\mathbf{H}^{l-1})^\top \mathbf{A} \mathbf{G}^l$$

Communication in this step is a small 1D outer product. Locally multiplying  $\mathbf{H}^{l-1} \mathbf{A} \mathbf{G}^l$  yields a single matrix per process of size  $f \times f$  that must be reduced across processes. We treat this communication cost as a lower-order term.

## 3.2 Sparsity-Aware 1.5D Algorithm using Point-to-Point Communication

In 1.5D algorithms, processes are organized in a  $P/c \times c$  process grid [17, 23]. In this regime, both  $\mathbf{A}^\top$  and  $\mathbf{H}$  are partitioned into  $P/c$  block rows, and each block row is replicated on  $c$  processes. Alternatively, instead of replicating a block row across  $c$  processes, one can split the block row of  $A$  into  $c$  chunks. Thus, the communication cost to replicate could be less. However, asymptotically, the communication of the SpMM algorithm does not change.

Specifically, each process in process row  $P(i, :)$  stores  $\mathbf{A}_i^\top$  and  $\mathbf{H}_i$ .

$$\mathbf{A}^\top = \begin{pmatrix} \mathbf{A}_1^\top \\ \vdots \\ \mathbf{A}_{p/c}^\top \end{pmatrix} \mathbf{H} = \begin{pmatrix} \mathbf{H}_1 \\ \vdots \\ \mathbf{H}_{p/c} \end{pmatrix} \quad (3.2)$$

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

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

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

However, each process only computes a partial sum above this summation. Of the  $p/c$  terms, each process in  $P(i, :)$  sums  $q = p/c^2$  distinct terms in parallel. These partial sums are then summed across all  $c$  processes in  $P(i, :)$  with an all-reduce call. The result is the final  $\mathbf{T}_i$  matrix replicated on each process in  $P(i, :)$ . The computation done on process  $P(i, j)$  is

$$\mathbf{T}_i = \mathbf{T}_i + \mathbf{A}_i^\top \mathbf{H} = \mathbf{T}_i + \sum_{k=jq}^{(j+1)q} \mathbf{A}_{ik}^\top \mathbf{H}_k \quad (3.3)$$

Like the 1D algorithm, a process  $P(i, j)$  would store  $\mathbf{A}_{ik}^\top$  locally, but accessing  $\mathbf{H}_k$  for all values of  $k$  requires communication. The sparsity-oblivious counterpart for 1.5D communicates entire block rows of  $\mathbf{H}$ . Our sparsity-aware version communicates the rows of  $\mathbf{H}$  needed for local SpMM computation, with an added space and latency cost of communicating the necessary row indices.

---

**Algorithm 2** Sparsity-Aware 1.5D algorithm using Point-to-Point communication for GNN forward propagation.  $\mathbf{A}$  and  $\mathbf{H}^l$  are distributed on a  $p/c \times c$  process grid.  $NnzCols(i, j)$  returns the nonzero column indices in  $\mathbf{A}_{ij}^\top$ , and is computed as a preprocessing step.

---

```

1: procedure BLOCK1.5DSAFW( $\mathbf{A}, \mathbf{H}^{l-1}, \mathbf{W}, \mathbf{H}^l$ )
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:       if  $P(i, j) = P(q, j)$  then
7:         for  $l = 0$  to  $p/c$  do
8:            $srows = NnzCols(l, j)$ 
9:            $ISEND(\mathbf{H}^{l-1}[srows, :], P(l, j))$ 
10:           $rrows = NnzCols(i, q)$ 
11:           $RECV(\mathbf{H}^{l-1}[rrows, :], P(q, j))$ 
12:           $\hat{\mathbf{Z}}^l \leftarrow \mathbf{Z}^l + \text{SPMM}(\mathbf{A}_{iq}^\top, \mathbf{H}^{l-1})$ 
13:           $\mathbf{Z}^l \leftarrow \text{ALLREDUCE}(\hat{\mathbf{Z}}, +, P(i, :))$ 
14:           $\mathbf{H}^l \leftarrow \text{GEMM}(\mathbf{Z}^l, \mathbf{W})$ 

```

---

**Equation**  $\mathbf{Z}^l = \mathbf{A}^\top \mathbf{H}^{l-1} \mathbf{W}^l$

We describe our sparsity-aware 1.5D algorithm in Algorithm 3. Each iteration  $q$  of the outer loop for process  $P(i, j)$  receives the rows of  $\mathbf{H}$  at row indices  $NnzCols(i, q)$ , followed by a local SpMM operation. The number of rows received is upper bounded by  $cut_P(G)$ . Finally, the all-reduce has each process row reduce matrices of size  $n/(p/c) \times f$ . This will be a lower-order term. This yields the following overall communication cost:

$$T_{comm} = \alpha \left( \frac{P^2}{c^3} \right) + \frac{P^2}{c^3} cut_{P(A)/c} f \beta$$

In practice,  $cut_P(A)$  will, up until a transition point, scale down roughly by  $P/c$  since the graph partitioner partitions  $\mathbf{A}$  into  $P/c$  partitions. Having  $P/c$  in the denominator of  $cut_P(G)$  yields a bandwidth term that scales down by  $c$ .

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

No communication is necessary as  $\mathbf{H}^l$  is partitioned by rows. This step is identical to the sparsity-oblivious algorithm.

**Equation**  $\mathbf{G}^{l-1} = \mathbf{A} \mathbf{G}^l (\mathbf{W}^l)^\top \odot \sigma'(\mathbf{Z}^{l-1})$

Like the 1D algorithm, no communication is required to transpose  $\mathbf{A}^\top$  to  $\mathbf{A}$ , since we either assume a symmetric matrix or explicitly store the transpose. Matrices  $\mathbf{A}$ ,  $\mathbf{G}^l$ , and  $\mathbf{Z}^{l-1}$  are

all partitioned in block rows, and  $\mathbf{A}$  is sparse while  $\mathbf{G}^l$  is dense. Thus, multiplying  $\mathbf{A}\mathbf{G}^{l-1}$  follows the same communication pattern as Algorithm 3. The communication cost is the same as Section 3.2.

**Equation**  $\mathbf{Y}^{l-1} = (\mathbf{H}^{l-1})^\top \mathbf{A}\mathbf{G}^l$

Like the 1D algorithm, communication in this step is a small outer product. Locally multiplying  $(\mathbf{H}^{l-1})^\top \mathbf{A}\mathbf{G}^l$  yields a single matrix per process of size  $f \times f$  that must be reduced across processes. We treat this communication as a lower-order term.

### 3.3 Sparsity-Aware 1.5D Algorithm using All-to-All Communication

---

**Algorithm 3** Sparsity-Aware 1.5D algorithm using All-to-All communication for GNN forward propagation.  $\mathbf{A}$  and  $\mathbf{H}^l$  are distributed on a  $p/c \times c$  process grid.  $NnzCols(i, j)$  returns the nonzero column indices in  $\mathbf{A}_{ij}^\top$ , and is computed as a preprocessing step. **rows** is simply an array of  $p/c$  pointers.

---

```

1: procedure BLOCK1.5DSAFWA2A( $\mathbf{A}, \mathbf{H}^{l-1}, \mathbf{W}, \mathbf{H}^l$ )
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:       if  $P(i, j) = P(q, j)$  then
7:         for  $p$  in  $P(:, j)$  do
8:           rows[ $p$ ] =  $H[NnzCols(p, i), :]$ 
9:       ALLTOALLV(rows,  $P(:, j)$ )
10:      for  $k = 0$  to  $p/c - 1$  do
11:        if rows( $k$ )  $\neq \emptyset$  then
12:           $\hat{\mathbf{H}}^{l-1}[NnzCols(i, k), :] = \mathbf{rows}(k)$ 
13:           $\hat{\mathbf{Z}}^l \leftarrow \hat{\mathbf{Z}}^{l-1} + \text{SPMM}(\mathbf{A}_{ik}^\top, \hat{\mathbf{H}}^{l-1})$ 
14:           $\mathbf{Z}^l \leftarrow \text{ALLREDUCE}(\hat{\mathbf{Z}}, +, P(i, :))$ 
15:           $\mathbf{H}^l \leftarrow \text{GEMM}(\mathbf{Z}^l, \mathbf{W})$ 

```

---

By switching the 1.5D Algorithm from using Point-to-point communication to All-to-All communication, we can reduce the number of communication stages. Instead of waiting for one set of sends and receives to complete before communicating more data, by sending them all at once, we can utilize more bandwidth, which might be able to decrease communication costs. However, the amount of communication (number of messages and size of each message)

still remains the same so the theoretical cost of communication computed above in section 3.2 remains the same.

Table 4.1: Average and maximum amount of data communicated in a single SpMM where the sparse matrix is distributed with Metis graph partitioner (instance: Amazon,  $f = 300$ ).

$p$	data size (MB)		load imbalance %
	average	max	
16	199.6	333.5	67.1%
32	132.9	241.6	81.8%
64	83.9	164.0	95.4%
128	52.5	117.3	123.3%
256	32.6	86.4	164.9%

## Chapter 4

# Graph Partitioning

Distribution of the sparse and dense matrices in both sparsity-oblivious and sparsity-aware GNN training can be achieved by a simple 1D block distribution where each block has roughly the same number of rows. Randomly permuting the adjacency matrix  $\mathbf{A}$  could lead to better load balance. However, it has two main shortcomings that can hinder scalability. First, it completely disregards the amount of communication during the training. This is valid even for the sparsity-aware training despite the fact that it selectively communicates only the rows of  $\mathbf{H}$  that are needed by a process. A random permutation that is applied prior to training for achieving good computational load balance may exacerbate this issue as it may cause many nonzero column segments in off-diagonal blocks of  $\mathbf{A}$ , which determine which rows of  $\mathbf{H}$  to communicate. Another shortcoming is that an even distribution of rows of  $\mathbf{A}$  may not always yield good computational load balance if the number of nonzeros in these rows are not even, which is usually the case in real-world graphs. Both of these issues can be remedied by distributing the matrices with a graph partitioner.

In sparsity-aware GNN training,  $P(i)$  needs to receive rows of  $\mathbf{H}_j$  corresponding to the nonzero column segments in its off-diagonal blocks  $\mathbf{A}_{ij}^\top$ , where  $i \neq j$ . Compared to sparsity-

oblivious training, the sparsity-aware training aims to avoid receiving the entire  $\mathbf{H}_j$  by not communicating the rows of  $\mathbf{H}_j$  corresponding to the zero column segments. However, if there are not many such zero-column segments in off-diagonal blocks, which is likely to happen if the graph is randomly permuted to get good computational load balance, the sparsity-aware training may not yield a big reduction in communication time. Partitioning the adjacency matrix with a graph partitioner prior to training among  $p$  processes helps in reducing the number of nonzero column segments in addition to achieving computational load balance. Graph partitioning is commonly used to parallelize sparse iterative solvers, usually focusing on distributing the computations related to SpMV in them. The partitioning models for SpMV can easily be extended to SpMM, the bottleneck operation in full-batch GNN training in this work. However, when distributing the adjacency matrix to processors for SpMM, it is necessary to consider the imbalance of nonzeros assigned to each process as well as the imbalance in communicated data as anything done for SpMV is amplified with a factor of at most  $f$ .

Among the two factors mentioned above, the first can easily be addressed by enforcing a stricter load balance constraint in partitioning. The second factor of communication load imbalance is more difficult to address as most partitioners usually only aim at reducing the total edgecut in partitioning, which corresponds to reducing total amount of transferred data. The problem of high load imbalance in communication can be severe as the overall communication time is determined by the bottleneck process, i.e., the process that communicates the maximum amount of data. Table 4.1 presents various statistics regarding communication in a single SpMM obtained by using the partitioner Metis [15] on Amazon data in GNN training for  $p \in \{16, 32, 64, 128, 256\}$ . The large sizes of messages in megabytes coupled with communication load imbalance which can be as high as 165% (i.e., the bottleneck process sending 2.7x the amount of data of an average process) makes it imperative to address this issue in order not to make communication a bottleneck.

To alleviate this issue, we utilize a partitioner that can handle multiple communication cost metrics related to volume [2]. This partitioner can simultaneously handle metrics such as total volume of communicated data, maximum send volume, maximum receive volume, etc. In our work we rely on this partitioner to optimize the total and maximum send volume metrics.

# Chapter 5

## Experimental Setup

### 5.1 System Details

All of our experiments are run on the Perlmutter system at NERSC, on which each node is equipped with 4 NVIDIA A100 GPUs with 40GB HBM memory. There are 2 NVLink links between each pair of GPUs within a node each with a bandwidth of 25GB/s. Each GPU is connected to an AMD EPYC 7793 CPU with a PCIe 4.0 bus. The CPU is connected 4 HPE Slingshot 11 NICs also using a PCIe 4.0 bus. Each NIC supports a 25GB/s bandwidth. Thus, within one node, our bandwidth is limited by the NVLink between each GPU, but when working with more than 4 GPUS (multiple nodes), our bandwidth is limited by the NIC bandwidth.

### 5.2 Implementation Details

We use PyTorch’s `torch.distributed` package with a backend in NCCL 2.11.4 for distributed communication. We start by dividing our adjacency matrix into block rows. These block rows are, by default, equally-sized, but if we have use a partitioner, the block rows may be variable in size depending on the returned partitions. We compute the nonzero column segments within these block rows, and communicate them to all other processors. For this, we perform a number of all-to-all calls. This step is only done once before GNN training starts, and thus, we do not include it as part of our training time because this time will be amortized as the number of epochs we train our network for increases.

Then, we create a 3-layer GNN architecture discussed in [16]. The model has a default of 16 hidden layers and 0 weight decay. For every experiment, we run the training loop for 100 epochs. We replace the distributed 1D multiplications with our sparsity-aware 1D algorithm or 1.5D algorithm. As mentioned above, the 1D algorithm uses all-to-all communication. While an all-to-all is simply a series of point-to-point sends and receives across each pair of processes, NCCL has optimized this such that the entire set of sends and receives is blocking as a whole, but each individual call is not separately blocking. This behavior is



achieved through NCCL’s use of `ncclGroupStart()` and `ncclGroupEnd()` which is used within the `torch.distributed` API [10]. Our 1.5D algorithm uses *isends* and *recvs*. By using `torch.distributed`’s `batch_isend_irecv` API, we can take advantage of the same grouping behavior as the all-to-all calls. This way, the *isend* is non-blocking.

For any underlying SpMM calls, we created a C++ extension of CuSPARSE’s `csrmm2` as described in [23]. We use CUDA 11.7.

### 5.3 Datasets

We ran experiments for the 1D and 1.5D algorithms on the Reddit, Amazon, and Protein datasets. Table 5.1 presents some properties of these graphs. Each vertex of the Reddit graph represents a post and an edge exists between two vertices if the same user commented on both posts [12]. This is our smallest and densest dataset. The vertices of the Amazon graph encode different products, and an edge exists if there exists a buyer that purchases both products [14]. This is our sparsest dataset. Finally, the vertices of the Protein graph represents proteins, and there exists an edge between two vertices if the respective proteins exhibit a certain degree of similarity. This is the largest graph in our dataset. For the Protein graph, we have used an induced subgraph consisting of 1/8 of the vertices of the larger original graph in [4]. Note that all three graphs are symmetric, so only the adjacency matrix needs to be stored because  $\mathbf{A} = \mathbf{A}^T$ .

For the Reddit dataset, we used the original features and labels as in [12]. For Amazon and Protein datasets, we chose an arbitrary number of features and labels for each dataset, and use the adjacency matrix to encode the relationship between vertices.

Table 5.1: Datasets used in our experiments

Graph	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	300	24

### Graph Partitioning

When we use a partitioner, the sparse matrix is permuted according to the partition ids provided by the partitioner. We use a symmetric permutation of the sparse matrix. For these experiments, we use the partitioner in [2] that optimizes both total communication across all processors and maximum send-communication volume for a single processor. It suffices to use the partitioner only once since the pattern of the sparse adjacency matrix does not change throughout the GNN training. The overhead of this pre-processing stage is not included in the reported runtimes in our experiments.

We have also compared the accuracy between these sparsity-aware implementations of the 1D and 1.5D algorithms and the preceding sparsity-oblivious implementations, and there has been no change in accuracy apart from any floating-point accumulation. This makes sense as we have not changed the underlying multiplication operations. Thus, in the following sections, we will only focus on the performance benefits.

# Chapter 6

## Results

### 6.1 Performance of 1D Sparsity-Aware Algorithm

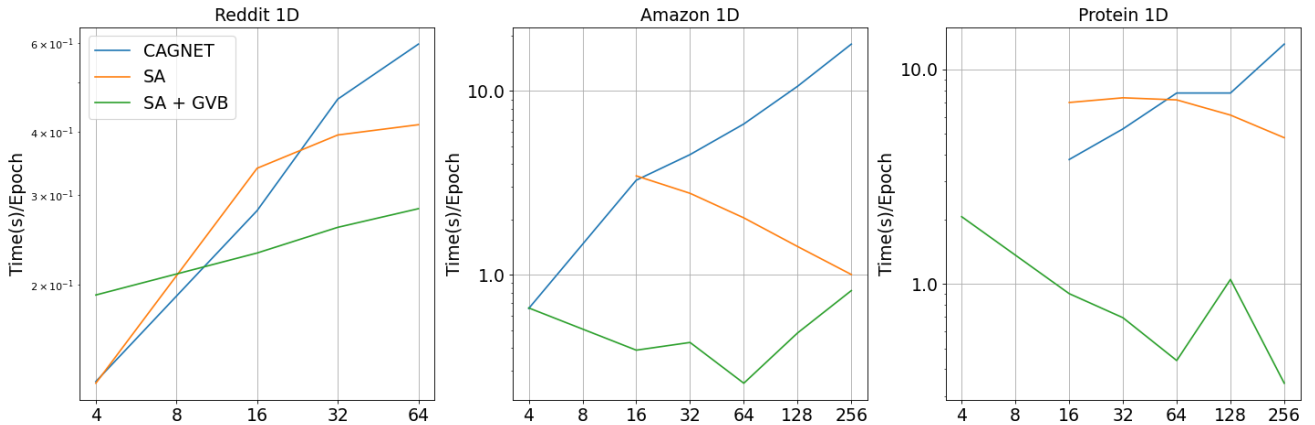


Figure 6.1: 1D performance results for sparsity-oblivious, sparsity-aware, and graph partitioning. Note that these are log-log plots of the number of GPUs versus the time for a single epoch. For Reddit, we use  $p=4, 16, 32, 64$ . For Amazon and Protein datasets, we also use  $p=128$  and  $256$ . Missing data in the line segments on Amazon for  $p=4$  and on Protein for  $p=4$  means that this trial of the experiment ran out of memory.

The performance of our 1D sparsity-aware training is compared against the 1D sparsity-oblivious training in Figure 6.1. We compare three schemes: the sparsity-oblivious training denoted with CAGNET, the sparsity-aware training described in Section 3 and denoted as SA, and its enhancement with graph partitioning as described in Section 4 and denoted as SA+GVB. We plot the average time spent at each epoch during the training with these schemes against the number of processes in  $x$  axis.

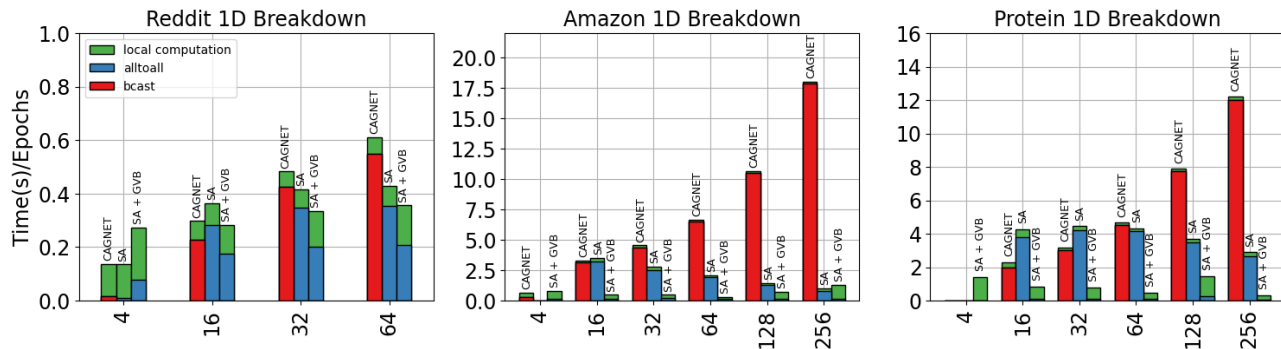


Figure 6.2: 1D performance breakdown. The x-axis of each plot refers to the number of GPUs used. This breakdown includes *local computation*, *alltoall*, and *bcst*. We compare results against CAGNET [23]. SA represents just a sparsity-aware implementation, and SA + GVB refers to our sparsity-aware implementation used in conjunction with graph partitioning. The sparsity-oblivious CAGNET implementation involves the broadcast and local computation, which in this case consists of the local SpMM computations. The sparsity-aware implementations used in the middle and right bar involves a single all-to-all call and a series of local computations which includes gathering the data to send, allocating space in GPU memory, as well as the local SpMM computation.

We can see that the original sparsity-oblivious scales up with the number of processors because the amount of data sent is proportional to the number of processors. The Reddit dataset, however, seems to be latency bounded where each epoch, where regardless of the algorithm or the number of processes, training time for one epoch takes less than a second. The Amazon data shows that for a small number of processors ( $p = 16$ ), the sparsity-aware algorithm makes little to no difference to the resulting training time. This means that the block rows are wide enough to the end that the number of nonzero subcolumns is not significantly less than the total number of subcolumns in that block. Thus, the benefit of sparsity-aware algorithms is seen for higher process counts ( $p \geq 32$ ) where communication is proportional to the edgecut. The same pattern appears in the Protein results where for lower process counts ( $p < 64$ ), the sparsity-aware algorithm takes longer than the original algorithm. In these cases, the cost of using point-to-point communication which scales linearly based on the amount of data instead of broadcasts which scales logarithmically is not displaced by communicating only rows that correspond to nonzero subcolumns. Like before, this is because the edgecut is not small enough. Interestingly, the sparsity-aware timing does not seem to be increasing either for lower process count. As  $p$  increases ( $p > 64$ ), the sparsity-aware implementation starts to show benefit and the timing per epoch shows a promising decreasing trend.

A granular breakdown is presented in Figure 6.2. The original algorithm timings seem to be overwhelmingly dominated by communication. While the Reddit data seems to be

latency-bound, at a higher process count ( $p \geq 32$ ), the data shows cost of communication is decreasing. This is more prevalent in the Amazon dataset, where comparing just the original sparsity-oblivious implementation and sparsity-aware implementation, communication costs start to decrease at  $p \geq 32$ . The local computation cost stays about the same because it is mostly made of the local SpMM computation which is common across both implementations.

## Graph Partitioning Performance

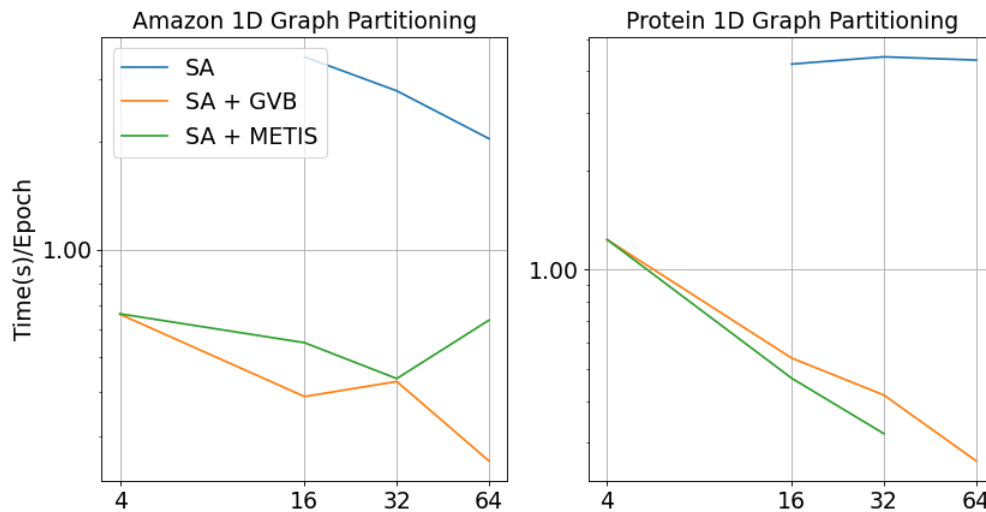


Figure 6.3: Benefits of Graph-VB (GVB) and Metis partitioners for sparsity-aware (SA) 1D training. The x-axis represents the number of GPUs used. The y-axis is the time per epoch. For this comparison, we use  $p=4,16,32$ , and 64. Note that this is also a log-log plot. The missing line segments represent that this trial of the experiment ran out of memory.

We further utilize graph partitioner Graph-VB [2] in 1D sparsity-aware GNN training to distribute the sparse adjacency matrix. As seen in Figure 6.1, utilizing a partitioner greatly benefits the training performance by reducing the overall execution time drastically. The main reason for this can be seen Figure 6.2 in which the communication bottleneck is largely overcome with the help of the partitioner where SA+GVB is further able to improve upon SA. Compared to SA, SA+GVB usually reduces the communication time. In the Protein graph, it can actually reduce the communication to such a degree that it is almost non-existent. The degree of reduction in communication is dependent on the sparsity pattern of the graphs: Reddit and Amazon graphs are more irregular than the Protein graph, which makes the job of the partitioner easy in the latter and difficult in the former. However, it may sometimes increase the local computation time (compare the blue and green bars of SA and SA+GVB in Figure 6.2). This is because we used a rather loose constraint on

computational load balance in partitioning in favor of further decrease in communication costs. Since SA is oblivious to reducing communication while distributing the sparse matrix, it can solely focus on, and obtain better computational load balance than SA+GVB.

We next compare Graph-VB (SA+GVB) against Metis (SA+METIS) to assess the effect of addressing multiple cost metrics in reducing communication and present the results on the Amazon and Protein graphs in Figure 6.3. In the Amazon graph it is clearly seen SA+GVB results in lower training time by successfully reducing the overhead of the bottleneck process, sometimes leading to more 2x performance benefit. In the Protein graph both partitioners exhibit similar behavior. In this instance both partitioners reduce the edgecut drastically (only a few thousand edges become cut out of hundreds of millions edges). Hence, the determining factor in training time between these two schemes becomes the computational load balance, in which SA+GVB performs slightly worse since it relies on variants of multi-constraint partitioning. These results show that it is possible to further improve the training performance with a more capable partitioner than a plain partitioner, especially on difficult instances whose sparsity pattern is more irregular.

## 6.2 Performance of 1.5D Sparsity-Aware Algorithm

For both the Amazon and Protein datasets, the sparsity-aware point-to-point algorithm does not outperform the original sparsity-oblivious algorithm for lower process counts as seen in Figure 6.4. We conclude that this is because the original algorithm uses broadcasts whereas the sparsity-oblivious algorithm uses point-to-point communication. At higher process counts, we see benefit of a sparsity-aware algorithm using collective communication, especially in the Amazon dataset. We expect from our analysis that as  $c$  increases, the communication time (and thus the total time) decreases. This is observed clearly in the original algorithm as well as the sparsity-aware version using point-to-point communication. The sparsity-aware algorithms on the graph partitioned dataset reveals much better runtimes both for Amazon (a larger, but less dense graph) and Protein (a smaller, but far denser graph) using both the point-to-point communication and collective communication operations. Note that when using graph partitioning, we require  $k = p/c$  partitions (rather than  $p$  partitions as in the 1D algorithm) and the edgecut only decreases up to a certain point until it starts increasing again. This point depends on the input graph. Thus, we expect that the sparsity-aware algorithm combined with graph partitioning will have decreasing runtimes until  $p = kc$  after which point, the runtime will start increasing again. We see this occur quite clearly with the Amazon dataset, where the minimum runtime for  $c = 2$  occurs at  $p = 32$  and for  $c = 4$ ,  $p = 64$ . While this pattern is not as visible in the Protein data, we can see the formation of a minimum, signaling that there is an optimal number of partitions. However, we show that when using all-to-all communication, we can reduce the latency because instead of sending many messages in different stages of communication, we send all messages in one stage of communication. Lowering latency enables better scaling to a larger number of processors. This is shown in Figure 6.4 the data by a decreasing trend

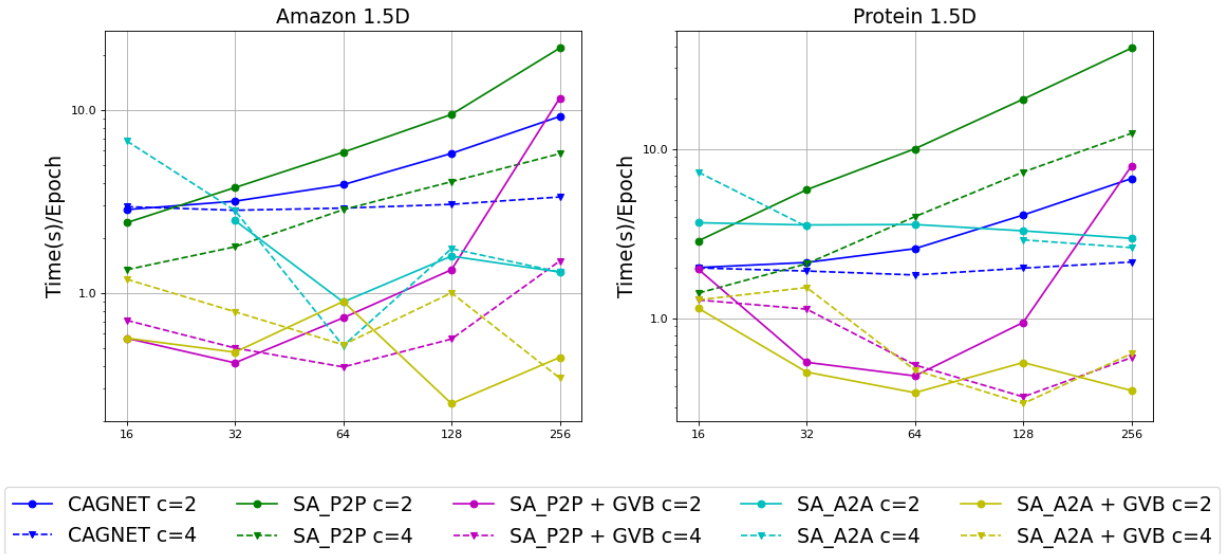


Figure 6.4: 1.5D performance results for sparsity-oblivious (CAGNET), sparsity-aware using Point-to-Point (P2P) communication and all-to-all (A2A) communication, and graph partitioning for Amazon and Protein datasets for  $c=2$  (solid line with circle markers) and  $c=4$  (dashed line with triangle markers). The x-axis of each plot refers to the number of GPUs used. We use  $p=16, 32, 64, 128, 256$ . Note that  $c$  represents the replication factor and that this is a log-log plot.

for the all-to-all version of the sparsity-aware algorithm.

# Chapter 7

## Conclusion

We have demonstrated a sparsity-aware approach to reducing communication between processors during GNN training resulting in a lower runtime per epoch. We evaluated the sparsity-aware approach against three datasets (Reddit, Amazon, and Protein) of different size and density revealing that for 1D vertex partitioning, for the cost of storing nonzero columns of the adjacency matrix and altering the communication pattern from broadcasts to point-to-point communication, there is an overwhelming benefit in communication time, thus reducing the original bottleneck in training times. We also show that using a partitioner that optimizes for both total communication volume (total number of edges crossing partitions) and maximum send communication volume (maximum number of edges from one processor to all others), we reduce load imbalance in communication, thus further reducing runtimes. Our results with respect to the 1.5D algorithm show that the same idea of sparsity-awareness combined with graph partitioning can be applied to other communication-avoiding partitioning schemes, such as 2D or 3D.



# Bibliography

- [1] Sergi Abadal et al. “Computing graph neural networks: A survey from algorithms to accelerators”. In: *ACM Computing Surveys (CSUR)* 54.9 (2021), pp. 1–38.
- [2] Seher Acer, Oguz Selvitopi, and Cevdet Aykanat. “Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems”. In: *Parallel Computing* 59 (2016), pp. 71–96.
- [3] Kadir Akbudak, Oguz Selvitopi, and Cevdet Aykanat. “Partitioning Models for Scaling Parallel Sparse Matrix-Matrix Multiplication”. In: *TOPC* 4.3 (2018), 13:1–13:34.
- [4] Ariful Azad et al. “HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks”. In: *Nucleic Acids Research* 46.6 (Jan. 2018), e33–e33. DOI: [10.1093/nar/gkx1313](https://doi.org/10.1093/nar/gkx1313). eprint: <https://academic.oup.com/nar/article-pdf/46/6/e33/24525991/gkx1313.pdf>. URL: <https://doi.org/10.1093/nar/gkx1313>.
- [5] Gray Ballard et al. “Communication optimal parallel multiplication of sparse random matrices”. In: (2013), pp. 222–231.
- [6] Grey Ballard et al. “Brief Announcement: Hypergraph Partitioning for Parallel Sparse Matrix-Matrix Multiplication”. In: *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures*. SPAA '15. Portland, Oregon, USA: ACM, 2015, pp. 86–88. ISBN: 978-1-4503-3588-1. DOI: [10.1145/2755573.2755613](https://doi.org/10.1145/2755573.2755613). URL: <http://doi.acm.org/10.1145/2755573.2755613>.
- [7] Aydın Buluç et al. “Recent Advances in Graph Partitioning”. In: *Algorithm Engineering - Selected Results and Surveys*. Vol. 9220. Lecture Notes in Computer Science, 2016. DOI: [10.1007/978-3-319-49487-6\\_4](https://doi.org/10.1007/978-3-319-49487-6_4).
- [8] Ümit V Çatalyürek et al. “More recent advances in (hyper) graph partitioning”. In: *ACM Computing Surveys* ().
- [9] Jie Chen, Tengfei Ma, and Cao Xiao. “Fastgcn: fast learning with graph convolutional networks via importance sampling”. In: *arXiv preprint arXiv:1801.10247* (2018).
- [10] NVIDIA Corporation. *NCCL: Optimized primitives for collective multi-GPU communication*. <https://github.com/NVIDIA/nvcl>. 2023.

- [11] Mehmet Deveci et al. “Hypergraph partitioning for multiple communication cost metrics: Model and methods”. In: *Journal of Parallel and Distributed Computing* 77 (2015), pp. 69–83. ISSN: 0743-7315. DOI: <https://doi.org/10.1016/j.jpdc.2014.12.002>. URL: <https://www.sciencedirect.com/science/article/pii/S0743731514002275>.
- [12] Will Hamilton, Zhitao Ying, and Jure Leskovec. “Inductive Representation Learning on Large Graphs”. In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 1024–1034. URL: <http://papers.nips.cc/paper/6703-inductive-representation-learning-on-large-graphs.pdf>.
- [13] Zhihao Jia et al. “Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with ROC”. In: *Proceedings of Machine Learning and Systems (MLSys)*. 2020, pp. 187–198.
- [14] Zhihao Jia et al. “Improving the accuracy, scalability, and performance of graph neural networks with ROC”. In: *Proceedings of Machine Learning and Systems 2* (2020), pp. 187–198.
- [15] G. Karypis and V. Kumar. “A fast and high quality multilevel scheme for partitioning irregular graphs”. In: 20.1 (1998), pp. 359–392.
- [16] 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.
- [17] Penporn Koanantakool et al. “Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication”. In: *Proceedings of the IPDPS*. 2016.
- [18] Lingxiao Ma et al. “NeuGraph: Parallel Deep Neural Network Computation on Large Graphs”. In: *USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, 2019, pp. 443–458. ISBN: 978-1-939133-03-8.
- [19] Vasimuddin Md et al. “DistGNN: Scalable distributed training for large-scale graph neural networks”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–14.
- [20] Oguz Selvitopi et al. “Distributed-Memory Parallel Algorithms for Sparse Times Tall-Skinny-Dense Matrix Multiplication”. In: *Proceedings of the ACM International Conference on Supercomputing*. ICS '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 431–442. ISBN: 9781450383356. DOI: [10.1145/3447818.3461472](https://doi.org/10.1145/3447818.3461472). URL: <https://doi.org/10.1145/3447818.3461472>.
- [21] G. M. Slota, K. Madduri, and S. Rajamanickam. “PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks”. In: *2014 IEEE International Conference on Big Data (Big Data)*. Oct. 2014, pp. 481–490. DOI: [10.1109/BigData.2014.7004265](https://doi.org/10.1109/BigData.2014.7004265).

- [22] John Thorpe et al. “Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads”. In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021, pp. 495–514. ISBN: 978-1-939133-22-9. URL: <https://www.usenix.org/conference/osdi21/presentation/thorpe>.
- [23] Alok Tripathy, Katherine Yelick, and Aydın Buluç. “Reducing communication in graph neural network training”. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2020, pp. 1–14.
- [24] Hanqing Zeng et al. “GraphSAINT: Graph Sampling based inductive learning method”. In: *Proceedings of the International Conference on Learning Representations (ICLR)*. 2020.