Communication Optimization for Distributed Execution of Graph Neural Networks

Süreyya Emre Kurt*, Jinghua Yan*, Aravind Sukumaran-Rajam†*, Prashant Pandey*, P. Sadayappan*
*Kahlert School of Computing, University of Utah, Salt Lake City, Utah
†Meta Platforms , Menlo Park, California

***Abstract*—Graph Neural Networks (GNNs) have emerged as a very powerful and popular machine learning model for numerous application domains. Each stage of a GNN requires an aggregation (sparse matrix-matrix multiplication) and a linear operation (dense matrix-matrix multiplication). Numerous efforts have addressed the development of distributed implementations for GNNs. Although efficient algorithms for distributed matrix multiplication are well known, the challenge here is the collective optimization of sequences of distributed matrix-matrix multiplications required for GNN, where many degrees of freedom also exist in the ordering of the component matrix-multiplication operations.**

This paper develops a new approach to distributed GNN, R*e*D*istribution of* M*atrices* (RDM), centered around communication-free distributed matrix-multiplication enabled by matrix redistribution between GNN stages. While the approach is applicable to the numerous algorithmic variants of GNN, the experimental evaluation focuses on GCN (Graph Convolutional Network), including both full-batch training as well as sampling-based training using GraphSAINT. Experimental evaluation with 2-layer and 3-layer GCN, using 128 or 256 hidden features, across eight sparse datasets, on a multi-GPU system with 8 GPUs shows that RDM attains a geometric mean speedup between $2\times$ and $3.7\times$ over two state-of-the-art multi-GPU GCN implementations, CAGNET and DGCL.

***Index Terms*—Graph Neural Networks, Distributed Algorithms, Multi-GPU GNN, Performance Modeling**

I. Introduction

Graphs contain structural information that relate entities (nodes) through relationships (edges). These relationships often provide transitive information for a node by examining the information at neighbors; e.g., nodes are likely to exhibit similar characteristics to their neighbors in the graph. Graph Neural Networks [1] have emerged as a prominent machine learning methodology to exploit the information contained in a graph. There has been an explosion of GNN research over the last decade. Due to the increasing size of graphs processed by GNNs, there is a considerable interest in developing distributed implementations for GNNs [2]–[10].

GNN algorithms like Graph Convolution Networks (GCN) [11], GraphSAGE [12], Graph Attention Networks (GAT) [13], and many others, have a computational structure where each layer of the GNN uses the graph structure to aggregate neighbor information, followed by application of a learnable neural network layer to the results of the aggregation. The aggregation operation can be efficiently implemented as an SpMM (Sparse-Dense Matrix Multiplication, where one matrix is sparse and the other is dense) operation and neural operation can be implemented as a GEMM (dense matrix-matrix multiplication).

The execution time of the aggregation step via an SpMM with a very large sparse matrix is generally much higher than the dense matrix-matrix multiplication for the neural operations [14]. This is largely because of the significantly lower performance (GFLOPs) achievable on CPUs and GPUs for SpMM with a very large sparse matrix ($N \times N$, where N is the number of graph vertices) versus dense matrix multiplication (GEMM). Further (as elaborated later), with existing distributed GNN implementations [2], [4], [8], inter-node communication is needed for the graph-aggregation step via the SpMM operation, while the subsequent GEMM operations for the neural operation are performed locally without any inter-node communication. Therefore the optimization of distributed SpMM has been a focus of several recent studies [4], [8].

Current implementations of distributed GNN generally use a *vertex-partitioned* approach to distributing the work of the aggregation step, as well as the neural processing among the nodes of a multi-node system. The vertex partitioning approach requires communication of the input features and intermediate activations across processor nodes, for graph edges whose vertices are mapped to different processors.

In this paper, we develop a different approach to distributed GNN — GNN-RDM, based on **R*e*D*istribution of* M*atrices***. We seek to perform *communication-free* matrix multiplication for both the sparse (aggregation) and dense (neural) steps, with appropriate *redistribution* of dense matrices among the memories of the distributed system between matrix operations. A significant advantage of this approach is improved scalability with increase in the number of GPUs. While existing distributed GNN schemes generally incur increasing communication volumes as the number of GPUs is increased, the total volume of data movement with RDM is independent of the number of GPUs.

Exploiting algebraic properties that allow the reordering of the dense and sparse matrix multiplications at each GNN layer to reduce the number of data redistribution steps, we develop a systematic taxonomy for a design space of configurations for the redistribution-based GNN. We further develop a performance model that can be used to identify pareto-optimal configurations with respect to data movement cost and the operation counts for SpMM operations, allowing a model-driven approach to selection of the best configuration

TABLE I: Notations used in the paper.

Notation	Definition
A	Adjacency matrix for GCN with N vertices
PX	Process with id X
P	Total number of processes
H_{in} and H_0	Input features to the GCN
H_{out}	Final embedding calculated in GCN
H_i	Output of layer i
f_i	Number of columns for matrix H_i
W_i	Weight matrix for layer i

for a given set of GNN parameters (graph structure, input embedding size, and sizes of intermediate neurons).

The space of algorithmic variants for GNN is very large, with differences along many dimensions. This includes differences in the specific combining operation used (sum, min, max, etc.) in aggregating data from neighbor vertices, the normalization operation used, the batch size used etc. Another difference among GNN variants is whether or not sampling on the graph is performed when aggregating information from neighbors, and if so how the sampling is done. The new RDM approach to distributed GNN is broadly applicable across the numerous variants of GNN used in applications today, including both sampling-based GNNs and full-batch GNNs.

In this paper, we demonstrate the impact of our work on two significant GNN models: i) full-batch GCN, the primary mode of training used for GNNs in metagenomics applications [15], [16], and ii) GraphSaint GCN [17], which has demonstrated that very high training accuracy can be achieved via graph-sampling to create subgraphs for training.

The paper makes the following contributions:

- It develops a novel multi-GPU GNN implementation based on *data redistribution* to minimize inter-node communication.
- It develops a systematic taxonomy and a design space of configurations for distributed multi-GPU GNN, along with an analytical performance model to identify pareto-optimal configurations with respect to communication overhead and sparse operation count.
- It presents an experimental study demonstrating the performance benefits of new approach to distributed GNN over existing frameworks, both for full-batch training as well as training based on cluster-sampling.

II. Background on GNN

A GNN comprises of a number of layers, each with a trainable linear neural operator, along with an aggregation operator that combines feature vectors from neighboring vertices in the graph. In this section, we first summarize the operations at a GNN layer for the forward and backward propagation during the training of the GNN, followed by a description of the 1D distributed GNN approach of CAGNET [4]. We use the same notation as Tripathy et al. [4].

We first discuss *full-batch* GCN (Graph Convolutional Network, a specific GNN algorithm) training *without sampling* as implemented in CAGNET. Later, we discuss GCN with sampling in Sec. III-F.

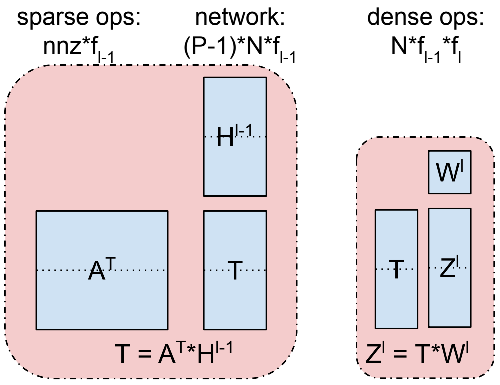


Fig. 1: 1D scheme in CAGNET.

Forward Pass: Let A represent the adjacency matrix for the connectivity graph. For layer l, let H_{l-1} be the input, H_l the output, and W_l the weight matrix. In the forward pass, there are two main operations: aggregation and the linear layer. In the aggregation operation, the transpose of adjacency matrix A is multiplied with H_{l-1} and the linear layer is applied to the result. This is described in eq. (1). A non-linear function is then applied to this output, as shown in eq. (2).

$$Z^l = A^\top H^{l-1} W^l \tag{1}$$

$$H^l = \sigma(Z^l) \tag{2}$$

Backward Pass: The backward pass of a layer computes the gradients. Let G^l be the gradient for H^l. Based on the loss value, the gradient is first computed for the last layer. The backward pass at stage l receives G^l from stage $l+1$, and computes the gradient Y^l for the weights W^l and the gradient G^{l-1}, as described in eqs. (3) and (4).

$$G^{l-1} = AG^l(W^l)^\top \odot \sigma'(Z^{l-1}) \tag{3}$$

$$Y^l = (H^{l-1})^\top AG^l \tag{4}$$

CAGNET Distributed GNN: Prior schemes [4], [8] have implemented distributed GNN by dividing both the sparse adjacency matrix and the dense activations across the nodes, with inter-node communication of needed dense activations at each stage of the GNN. The 1D scheme in CAGNET [4] divides both A and activations into horizontal bands and each process is responsible for storing its own tile as shown in Figure 1. For the SpMM operation $T = A^\top * H^{l-1}$, each process broadcasts its owned local data slice of H^{l-1} to other nodes so that each node has access to all elements of H^{l-1} to perform the computations for their local data slice of the output T. The next operation, dense matrix multiplication (GEMM) $Z^l = T * W^l$, does not require any communication since W^l is replicated among all the nodes. In the backward pass of GNN, a similar order of SpMM ($T = A * G^l$) and GEMM ($U = (H^{l-1})^\top T$) and Hadamard product $G^{l-1} = U \odot \sigma'(Z^{l-1})$ are applied. If a hardware multicast is not available, the total volume of data to be moved for a forward layer l broadcast is $(P-1) * N * f_{l-1}$ where H^{l-1} contains $N * f_{l-1}$ elements. For a 2 layer GNN with f_{in} input features, hidden layers with f_h neurons and f_{out} output classes, the total volume of data movement from the broadcast operations during the SpMM operations will be $(P-1)(f_{in}+2f_h+f_{out})N$, which is linear in terms of the number of processes P.

III. Communication Optimization for Distributed GNN

In this section we describe GNN-RDM, a new approach to distributed GNN based on **R**e**D**istribution *of* **M**atrices. In contrast to previously developed schemes for distributed GNN [3], [4], [7], [10], [18], which utilize a uniform strategy at each stage in the GNN, we perform a more comprehensive analysis of the data access patterns and inter-dependencies between stages and devise a distributed GNN approach that is demonstrated to achieve higher performance and be more scalable with increase in number of nodes than existing distributed GNN schemes.

As further elaborated below, the overall optimization problem for distributed GNN involves the following decisions:

[**Operator level**] For a given SpMM or GEMM operation at a GNN stage, there are many possible distributed execution options, corresponding to different partition/distribution strategies for the input and output dense matrices.

[**Block level**] For each GNN layer, there is flexibility in the order of the SpMM and GEMM operations, because of the associativity of chain matrix multiplication: the matrix chain product ABC can be computed either as $(AB)C$ or $A(BC)$. This choice affects both the total number of arithmetic operations, as well as the amount of inter-processor communication required for distributed execution.

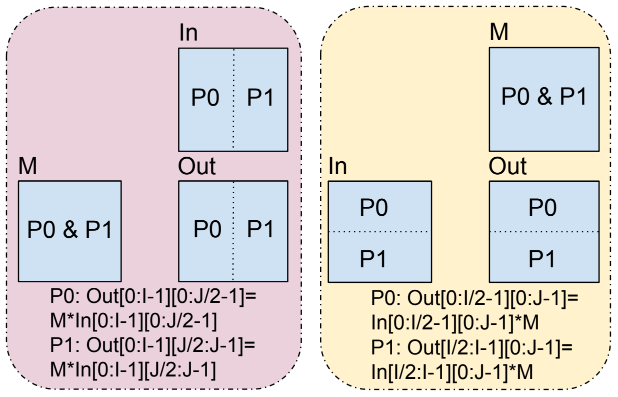
[**Network level, intra-pass**] The choice of orders of the SpMM and GEMM operations at adjacent layers affects the computational cost (number of floating-point operations) as well as the communication cost.

[**Network level, across forward/backward pass**] There is potential for reducing the total number of SpMM operations in the backward pass by saving and reusing a post-SpMM matrix in the forward pass, but this constrains the relative order of GEMM and SpMM in the backward pass.

A. Communication-Free Matrix Multiplication

A key idea behind our approach is that distributed matrix multiplication can be communication-free if one of the two input matrices is replicated on all nodes of a distributed system and the other input matrix is appropriately partitioned across processors. This property is used to devise a distributed GNN scheme where communication for the SpMM and GEMM operations is avoided by suitably redistributing the input dense matrix operand to a form that enables communication-free matrix multiplication for both operations.

Fig. 2 shows how matrix multiplication can be performed without communication when one of the input matrices is fully replicated among nodes and the other input matrix is suitably partitioned/distributed. Consider $Out = M * In$, where M is fully replicated among nodes, as shown in Figure 2a, which is representative of the SpMM operation in GNN. Vertically slicing In and Out in a compatible manner will enable distributed matrix-multiplication without any inter-processor communication. If either one of In or Out were horizontally tiled, there would be a need to broadcast In or a reduction for Out, requiring a greater volume of data movement. Similarly, if M is the right (second) operand in the matrix multiplication $Out = In * M$, with M being fully replicated across processors, and In and Out being horizontally sliced across processors, communication-free matrix multiplication is feasible, as shown in Figure 2b.



(a) GNN-SpMM: left input matrix replicated (Out = M*In) (b) GNN-GEMM: right input matrix replicated (Out = In*M)

Fig. 2: Communication free distributed matrix multiplication where one of the input matrices is replicated among nodes.

B. Operation Order

Section II described the operations needed for the forward and backward layers for GCN. Consider Equation (1). There are two choices for computation of $A^\top H^{l-1} W^l$: either first perform a sparse-dense matrix multiplication (SpMM) to form a temporary $T = A^\top H^{l-1}$, followed by the dense matrix product TW^l; or first perform the dense matrix multiplication $T = H^{l-1}W^l$, followed by the SpMM $A^T T$. The associativity of matrix-chain multiplication ensures that the two orders of computing the triple matrix-chain multiplication are equivalent. Similarly, in the backward pass (Equation (3)), there are two options: either first perform SpMM AG^l, or dense matrix multiplication $G(W^l)^\top$. As discussed later, the total number of arithmetic operations depends on the chosen execution order.

C. SpMM Reuse across Forward and Backward Pass

Another factor that we consider is the fact that some computational reuse is feasible across the collection of computations for the forward and backward passes of a layer. Fig. 3 shows that if AG^l is first computed for the computation of G^{l-1}, it can also be reused by choosing it as the first operation in computing W^l (shown shaded yellow). An alternative is to reuse a saved intermediate $A^T H^{l-1}$ from the forward pass (shaded blue). From the above observation of potential reuses, we can conclude that if the dense matrix multiplication is executed first in both forward and backward passes, an additional sparse operation will be required for computing gradient of W^l.

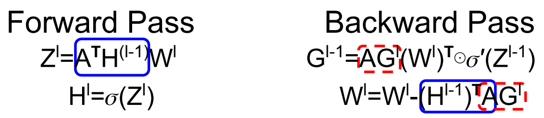


Fig. 3: Reuse between different GNN operations. $A^\top H^{l-1}$ (or $(H^{l-1})^\top A$) and AG^l needs to computed twice. We can avoid recomputation for one of them.

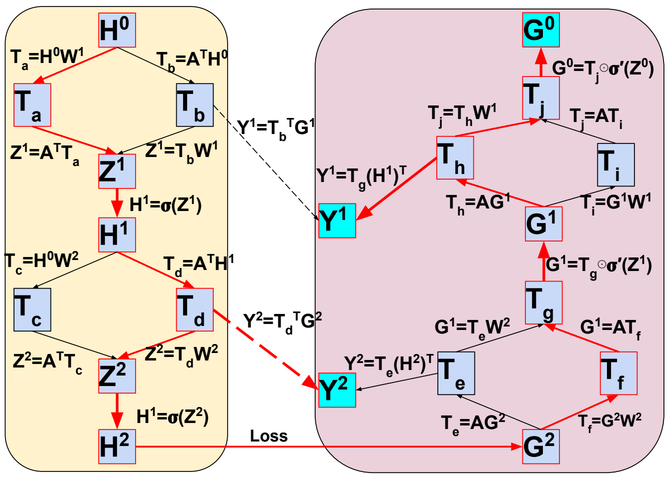


Fig. 4: Computation order and reuses. Path chosen for case 10 in Table IV is highlighted in red.

Figure 4 shows all possible computation orders and the data reuses. Each node in the graph represents a matrix and each edge represents a different computation order. Doted edges represent the possible reuse between forward and backward operators. The group of nodes on the left (with yellow background) represents the forward computations and the group of nodes on the right represents the backward computations. The operators with turquoise background (Y^1, Y^2, and G^0) represents the final output. For example, the edges T_a and T_b represent the dense first and sparse first orderings (sec. III-D, fig. 5) respectively, for the first forward layer. Z^1 can then computed from T_a or T_b as A^1T_a or T_bW^1 respectively. The remaining computations can be done similarly. The red arrows shows the dense-sparse-dense-sparse ordering (corresponds to ID 10 in Table IV). Note that T_d is required to compute Y^2 in the backward pass. It can be either saved during the forward pass or recomputed during the backward pass. The dotted red arrow indicates that in the chosen path (represented by red arrow), we saved T_d during the forward pass so that it can be reused during the backward pass.

D. Data Movement Cost

We can observe that in both the forward and backward passes, the SpMM operations are of the form AG, where the sparse matrix is the left operand. From Fig. 2, vertical slicing of the input/output dense matrices would result in communication-free execution. In contrast, the dense matrix multiplication operations are always of the form HW, for which horizontal slicing of the input/output dense matrices is needed for communication-free distributed execution.

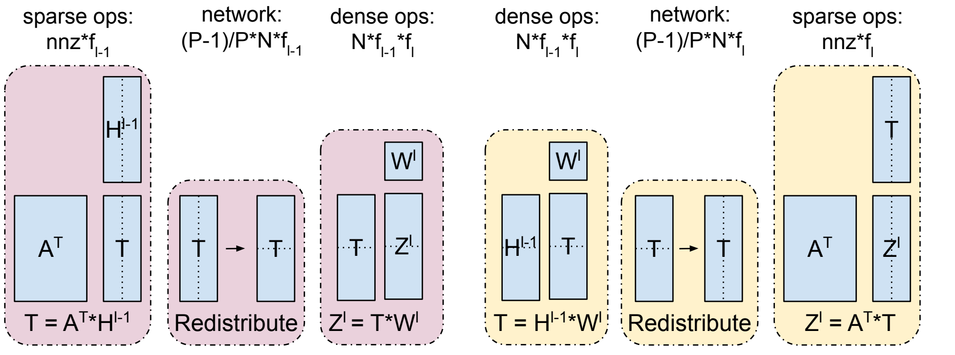
Thus, regardless of which one of $A^\top H^{l-1}$ or $H^{l-1}W^l$ is executed first in the forward pass in Equation (1), one of SpMM or dense mat-mult uses the output of the other operation and they require their inputs to be distributed differently. Thus, there will always be a need for redistribution of the intermediate dense matrix. Since redistribution can be done by an all-to-all personalized communication of the intermediate result, the total data movement volume will be constant in terms of the number of nodes.

Figures 5a and 5b illustrate the data partitioning and redistribution on a two node distributed-system (each dense matrix is sliced into two parts) for both choices: perform SpMM first or dense matrix-multiplication first. The number of arithmetic operations, as well as the aggregate volume of data movement, in a distributed system are shown above each operation. For the SpMM operation in Fig. 5a, the total number of FMA (Fused Multiply Add) operations is $nnz * f_{l-1}$, where nnz is the number of nonzero elements in the sparse matrix and f_{l-1} is the embedding size for the input vectors at this stage, or the width (number of columns) in the dense matrices. For the GEMM, the number of operations is $N * f_{l-1} * f_l$, where N is the number of vertices and f_l is the embedding size (the number of output neurons) at each graph vertex for that layer. The redistribution of a vertically sliced partition to a horizontally sliced partition (or vice-versa) will require a total volume of inter-node data movement volume of $(P-1)/P * N * f$ elements, where P is the number of processors.

E. Handling Very Large Sparse Matrices

If the sparse matrix A is too large to fit in a single GPU's memory, it is partitioned and distributed among multiple GPUs. Consider P GPUs to be viewed as a logical 2D grid of size $P_i \times P_j$. If there is sufficient aggregate GPU memory to make R_A replicas of A, they are organized in the following manner: the sparse matrix A is divided into $P_i = P/P_j$ row panels and each GPU holds $N_i R_A/P \times T_k$ tile of A. A group of GPUs holding the entire A will also store a $N_i \times N_j/P$ tile of B and can communicate among themselves to compute C tile of size $N_i \times N_j/P$ using the 1D algorithm described in [4]. Since A matrix is replicated R_A times, the data-movement of this algorithm is $(P/R_A - 1) * |B|$ since B is broadcast $P/R_A - 1$ times. When $R_A = 1$ this scheme will be identical to 1D scheme presented in CAGNET [4]; however, when $R_A > 1$ the data movement will be less than half of the $R_A = 1$ case.

Since the data movement for SpMM and R_A are inversely related, choosing largest possible R_A replication factor for A would give the least data movement for this scheme. If we assume the size of the memory per node is M, total size of input features and activation is $H_{all} = \sum H_i$, and the size of the sparse adjacency matrix is G, then each process can use $M - H_{all}$ of it's memory to store the adjacency matrix. Therefore adjacency matrix can be replicated $R_A = P\frac{M-H_{all}}{G}$ times. Therefore, the largest possible value of R_A is $P\frac{M-H_{all}}{G}$. Note that since there are P processes, adjacency matrix can be replicated at most P times; so, when this number is larger



(a) SpMM First (b) GEMM First

Fig. 5: Comparison between SpMM or GEMM being the first operation in the first layer of the GNN computation for a 2 node system.

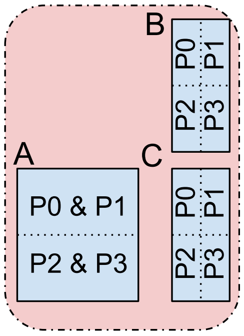
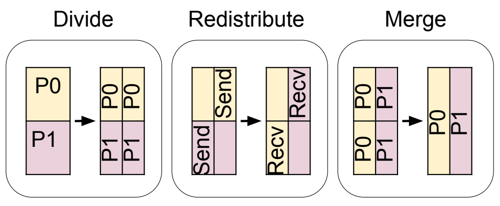
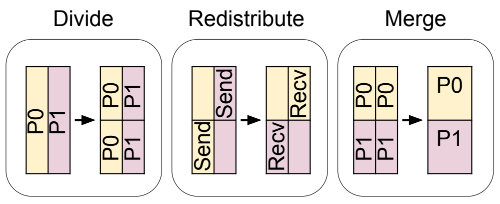


Fig. 6: SpMM on a 4-node system with 2-way row-panel partitioning of the sparse adjacency matrix A.



(a) Horizontal to vertical



(b) Vertical to horizontal

Fig. 7: Redistribution logic.

than P we are going to use $R_A = P$ or in other words we will fully replicate adjacency matrix.

Fig. 6 shows an example where $P_j = R_A = 2$ and $P = 4$. Matrix A is divided into 2 row-panels on a 4 node system. Each of the row panels of A is replicated among 4/2=2 nodes (top row-panel is stored in P0 and P1; bottom row-panel in P2 and P3). Each node computes a distinct slice of the output dense matrix of size $N/2 \times f/2$. In order to compute the owned output slice, each node will need all partitions of the dense input array in the same vertical slice. For example, P1 will need the input partitions owned by itself and P3. The needed data is communicated by broadcasting each input dense matrix partition to all processors owning partitions in the same vertical slice, i.e., between P0 and P2; and between P1 and P3.

F. Sampling in GNN

So far we have discussed *full-batch, full-graph* training, the mode of GNN primarily used in some application domains such as metagenomics [15], [16], and also demonstrated to be very effective in other applications. However, the use of sampling is very common, in order to address the *neighborhood explosion* problem. In sampling based GNN methods, the halo of the batch or k-hop neighborhood information uses only a sampled subset of the connected vertices, instead of all of them. A very effective approach to sampling is graph-sampling to create independently processed subgraphs, e.g., GraphSaint GCN [17], which has demonstrated that very high training accuracy can be achieved. We demonstrate the use of our RDM-based distributed GNN with such a sampling approach. For other kinds of sampling approaches that do not create independent sub-graphs, a masked SpMM kernel can be performed on the sampled neighbors of each vertex. This method would be similar to our proposed GCN-RDM scheme by passing along the sampled neighbor information to all processes and replacing SpMM routine with masked SpMM. Furthermore, a random generated seed can be passed to all processes and each process can generate it's sparse mask individually, reducing the communication overhead for the sampling mask.

IV. Performance Modeling

A. Cost Model of a Layer

In Section III-B we described the impact on performance of the relative order of SpMM and GEMM operations in the forward and backward passes. This section discusses the modeling of communication and computation cost. Figure 5 (a) and (b) show the SpMM-first and the GEMM-first ordering in the forward pass; analysis of the backward layer is similar.

1) Forward Pass: In Equation (1), let W^l represent a matrix with dimensions $f_{l-1} * f_l$, N represent the number of vertices in the network and nnz represent the number of edges. The extent of A is $N * N$, H^{l-1} is $N * f_{l-1}$ and Z^l is $N * f_l$. The number of operations in the forward pass are described in Table II. The number of operations in the GEMM does not depend on the order, but the number of operations in the SpMM depends on calculation order. If $f_l > f_{l-1}$, an SpMM-first strategy yields lower communication volume and computation for SpMM; if $f_l < f_{l-1}$ the GEMM-first strategy leads to fewer operations in SpMM and lower communication

TABLE II: Number of operations in the forward pass.

	SpMM first	**GEMM first**
SpMM Ops	$nnz * f_{l-1}$	$nnz * f_l$
GEMM Ops	$N * f_{l-1} * f_l$	$N * f_{l-1} * f_l$
Comm.	$(P-1)/P * N * f_{l-1}$	$(P-1)/P * N * f_l$
Comm. $(R_A < P)$	$(R_A-1)/R_A * N * f_{l-1}$ $+(P/R_A-1) * N * f_{l-1}$	$(R_A-1)/R_A * N * f_l$ $+(P/R_A-1) * N * f_l$

TABLE III: #Ops. in backward pass (non-memoized).

	SpMM first	**GEMM first**
SpMM Ops	$nnz * f_l$	$nnz * f_{l-1}$
SpMM Ops (N.M.)	$nnz * f_l$	$nnz * f_{l-1}$ + min($nnz * f_l$, $nnz * f_{l-1}$)
GEMM Ops	$2 * N * f_{l-1} * f_l$	$2 * N * f_{l-1} * f_l$
Comm.	$(P-1)/P * N * f_l$	$(P-1)/P * N * f_{l-1}$
Comm. (N.M.)	$(P-1)/P * N * f_l$	$(P-1)/P * N * f_{l-1}$ +2*min($nnz * f_l$, $nnz * f_{l-1}$)
Comm. $(R_A < P)$	$(R_A-1)/R_A * N * f_{l-1}$ $+(P/R_A-1) * N * f_{l-1}$	$(R_A-1)/R_A * N * f_l$
Comm. $(R_A < P)$ **(N.M.)**	$(R_A-1)/R_A * N * f_{l-1}$ $+(P/R_A-1) * N * f_{l-1}$	$(R_A-1)/R_A * N * f_l$ $+(P/R_A-1) * N * f_l$ +2*min($nnz * f_l$, $nnz * f_{l-1}$)

volume. The number of operations and communication volume of the nonlinear function in Equation (2) does not dependent on the execution order.

After the last forward layer, the loss function is computed. The loss function typically needs all the embeddings for a single vertex to be in the same process node. Thus, an extra redistribution is needed if the final embedding H^L is vertically tiled. Therefore, the cost of the last redistribution is $(P-1)/P * N$.

2) Backward Pass: Ignoring the Hadamard product in Equation (3), the analysis of SpMM-first and GEMM-first schemes for the backward pass are almost identical to the forward pass. Let the dimensions of G^l be $N * f_l$ and G^{l-1} be $N * f_{l-1}$. The GEMM cost will be same for both orders; however, if $A^\top H^{l-1}$ is not memoized in the forward pass and the GEMM is executed first in the backward pass, an additional SpMM of $A^\top H^{l-1}$ or AG^l is required. Table III summarizes the total cost of SpMM, GEMM and communication.

3) Inter Layer Communication:: This is affected by the order of multiplication. Figures 5a and 5b show how changing the multiplication order changes the data distribution of H^{l-1} and H^l among nodes. If the consecutive layers l and $l+1$ perform SpMM-first (or GEMM-first), they will require an additional redistribution of $(P-1)/P * N * f_l$ elements.

Figure 7a shows how data is redistributed from horizontal to vertical partitioning. Initially, $P0$ and $P1$ contain a horizontal slice of the data. Each processor then divides the data it owns into P vertical chunks (the divide step). In the next step, each processor redistributes the data. For example, $P0$ sends its second vertical chuck to $P1$, and $P1$ sends its first vertical chuck to $P0$. After the redistribution, each process contains the data corresponding to the entire vertical chunk it should own; however, the data is split across multiple chunks. In the final step, each processor merges its chunks (Merge step). Vertical to Horizontal redistribution also uses a similar approach and is shown in Figure 7b.

4) Row Tiling for Adjacency Matrix A: The details of an algorithm for $R_A < P$ is described in Section III-E. B and C are divided into $P/R_A \times R_A$ tiles in this scenario. If SpMM is performed first in a layer, the volume of data movement for redistribution will be $(R_A-1)/R_A * N * f_{l-1}$ because each $1 \times R_A$ group need to redistribute their C tile of size $N/R_A \times f_{l-1}$ among themselves to convert vertical slicing to horizontal slicing, required for GEMM. Similarly if GEMM is performed first, converting horizontal slicing to $(P/R_A) \times R_A$ slices will require a volume of data movement of $(R_A-1)/R_A * N * f_l$. Tables II and III show the summary for cases for $R_A < P$. Moreover, the data movement within the SpMM operation will be $(P/R_A-1)NF$, as described in Section III-E, where $F = f_{l-1}$ if SpMM is performed first and $F = f_l$ if GEMM is performed first.

B. Redistribution Model

In this section we investigate which order of SpMMs and GEMMs can yield the best performance.

If the CAGNET scheme [4] for a two layer GNN network is used for GNN-RDM, it can result in three data redistributions. As mentioned earlier, the order of operations affects the cost and a different choice can yield a scheme with lower inter-layer communication.

For a 2-layer GNN, there are 4 choices for the ordering of SpMM and GEMM for the two layers. As mentioned in Section IV-A, the cost of the GEMM operation does not change depending on multiplication order. In Table IV, the total cost of SpMM and communication are modeled. The second and third column describes whether SpMM (S) or GEMM (D) is executed first in the forward pass in layer 1 and 2, respectively. Similarly, column 4 and 5 describes whether SpMM (S) or GEMM (D) is executed first in the backward pass in layer 2 and 1, respectively. Since the nnz factor for each SpMM and the $(P-1)/P * N$ factor for each communication do not change with the choice of order of execution, they are not shown in the table. The entries in the table can be computed using the equations described in Section IV-A in $O(L \times 2^L)$ time where L is the number of layers. Since the number of layers is typically at most 4 in GNN applications, the cost of computing this table is negligible in comparison to the time for performing the training using an optimized order. Even though there are 16 different configurations for a 2-layer network, as shown in th enext section, only very few of them are Pareto optimal configurations in terms of communication and sparse operations. Therefore, after trying out each pareto-optimal configuration in the first few epochs, we can dynamically determine the fastest configuration and use that configuration for the remaining epochs.

V. Experimental Evaluation

We now present our empirical evaluation of RDM. We compare RDM against two other state-of-the-art distributed GNN systems, CAGNET [4] and DGCL [8] for full-batch training. We further evaluate RDM on GraphSaint [17], which uses a graph sampling approach that generates a number of sampled sub-graphs that are used for training. We compare RDM on GraphSAINT against the DGL implementation of

TABLE IV: Communication and computation cost: 2-layer GNN.

ID	Forward Pass		Backward Pass		Communication	Sparse Ops
0	S	S	S	S	$f_{in}+4f_h+2f_{out}$	$f_{in}+2f_h+f_{out}$
1	S	D	S	S	$f_{in}+2f_h+4f_{out}$	$f_{in}+f_h+2f_{out}$
2	D	S	S	S	$4f_h+2f_{out}$	$3f_h+f_{out}$
3	D	D	S	S	$4f_h+4f_{out}$	$2f_h+2f_{out}$
4	S	S	S	D	$2f_{in}+2f_h+2f_{out}$	$2f_{in}+f_h+f_{out}$
5	S	D	S	D	$2f_{in}+4f_{out}$	$2f_{in}+2f_{out}$
6	D	S	S	D	$f_{in}+2f_h+2f_{out}$ $+2min(f_{in},f_h)$	$f_{in}+2f_h+f_{out}$ $+min(f_{in},f_h)$
7	D	D	S	D	$f_{in}+2f_h+4f_{out}$ $+2min(f_{in},f_h)$	$f_{in}+f_h+2f_{out}$ $+min(f_{in},f_h)$
8	S	S	D	S	$f_{in}+4f_h$	$f_{in}+3f_h$
9	S	D	D	S	$f_{in}+2f_h+2f_{out}$ $+2min(f_h,f_{out})$	$f_{in}+2f_h+f_{out}$ $+min(f_h,f_{out})$
10	D	S	D	S	$4f_h$	$4f_h$
11	D	D	D	S	$4f_h+2f_{out}$ $+2min(f_h,f_{out})$	$3f_h+f_{out}$ $+min(f_h,f_{out})$
12	S	S	D	D	$2f_{in}+4f_h$	$2f_{in}+2f_h$
13	S	D	D	D	$f_{in}+2f_h+2f_{out}$ $+2min(f_h,f_{out})$	$2f_{in}+f_h+f_{out}$ $+min(f_h,f_{out})$
14	D	S	D	D	$f_{in}+4f_h$ $+2min(f_{in},f_h)$	$f_{in}+3f_h$ $+min(f_{in},f_h)$
15	D	D	D	D	$f_{in}+4f_h+3f_{out}$ $+2min(f_h,f_{out})$ $+2min(f_{in},f_h)$	$4f_h+3f_{out}$ $+min(f_h,f_{out})$ $+min(f_{in},f_h)$

GraphSAINT. DGL could not be compared for distributed full-batch training since it is not supported. Similarly, we could not evaluate GraphSAINT sampling with CAGNET or DGCL because the only supported scheme is full-batch training.

A. Experimental Setup

We used the CAGNET [4] framework for implementing RDM. We adopted the Graph Convolution Network (GCN) model [11] and reused the graph normalization code in CAGNET.

TABLE V: Datasets used in evaluation.

Dataset	Vertices	Edges	Feature size	Labels
OGB-Arxiv	169,343	1,166,243	128	40
OGB-MAG	1,939,743	21,111,007	128	349
OGB-Products	2,449,029	61,859,140	100	47
Reddit	232,965	114,848,857	602	41
Web-Google	875,713	5,105,039	256	100
Com-Orkut	3,072,441	117,185,083	128	100
CAMI Airways	1,000,000	22,901,745	256	25
CAMI Oral	1,000,000	20,734,972	256	32

a) Datasets: We used eight publicly available graph datasets for evaluation, with characteristics listed in Table V. OGB-Arxiv, OGB-MAG and OGB-Products are datasets from the Open Graph Benchmark suite [19]. OGB-Arxiv is a citation network graph of Computer Science arXiv papers. OGB-Products represents an Amazon product co-purchasing network. OGB-MAG is a heterograph with multiple types of nodes and edges, containing a subset of the information from Microsoft Academic Graph. For this study we use the paper-cites-paper relation to generate the graph for use with GNN. Reddit is a social network graph of the Reddit forum [12]. Web-Google [20] is a web graph where each node represents a web page, and each edge is a hyperlink between web pages.

TABLE VI: Pareto-optimal configurations (IDs from Table IV) for the datasets in Table V.

	f_{in}	f_h	f_{out}	Candidates IDs
OGB-Arxiv	128	128	40	5
OGB-MAG	128	128	349	10
OGB-Products	100	128	47	5
Reddit	602	128	41	2, 3 and 10
Web-Google	256	128	100	2, 3 and 10
Com-Orkut	128	128	100	5 and 10
CAMI Airways	256	128	25	2, 3, and 10
CAMI Oral	256	128	32	2, 3, and 10

Com-Orkut [21] is an online social network where each node represents a user and edges represent the friendship between users. Since node features and labels are not provided for Web-Google and Com-Orkut datasets, we generated random values to evaluate runtime. Table V shows the details of these datasets.

We also included two metagenomic graph datasets in our evaluation. Recently, it has been shown that GNNs can be used to classify sequences in metagenomics [15], [16], [22].

GNN-based classification techniques use connectivity information between input metagenomic sequences to construct an overlap graph [22]. In the overlap graph, each sequence acts as a node and two nodes are connected by an edge if they have an overlapping region. Then node classification is performed using graph neural network in a semi-supervised setting. The tetra nucleotide content, i.e., the frequencies of all four-length sequences, is used as input node features.

We used two publicly available long read datasets from the 2nd CAMI Toy Human Microbiome Project dataset: Oral cavity and Airways.

To generate the overlap graph, we took the $fastq$ file containing all sequences in a dataset, aligned them pairwise with minimap2 [23], and excluded self-aligned sequences with the flag `-X`.

.

b) System Details: We evaluated RDM on a system with two 32-core AMD EPYC 7513 CPUs and 8 NVIDIA RTX A6000 GPUs. Each GPU has 48 GB GDDR6 memory. We used PyTorch v1.9.0 as the deep learning framework, and CUDA version 11.1. We used the built-in distributed APIs of PyTorch to perform inter-device communication and NCCL v2.7.8 as the communication backend.

c) GNN Implementation Details: We implemented 2-layer and 3-layer GCN networks for the RDM scheme using the CAGNET infrastructure [4]. We experimented with two choices for the hidden layer dimension - 128 and 256. For each RDM evaluation, the analytical model described in Sec. Sec.sec:perf-model was used to identify all pareto optimal configurations with respect to the computational and communication costs. The pareto optimal configurations were executed and the best among those reported. For the 2-layer case for 128 hidden features, the pareto optimal configurations for the datasets are shown in Table VI. The The Adam optimizer was used to train all models (RDM, CAGNET, DGL). The learning rate was set to 0.01 for all full-batch experiments. In our experiments of Section V-C, the learning rate was set to

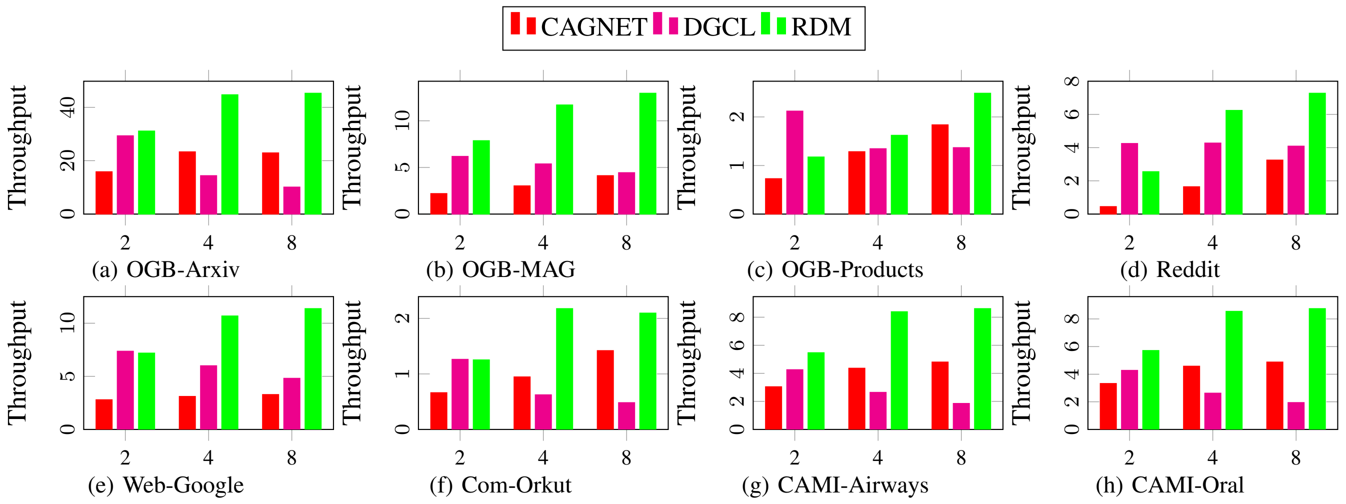


Fig. 8: Training throughput (epochs per second): 2-Layer GCN; hidden-size=128; X-axis: #GPUs.

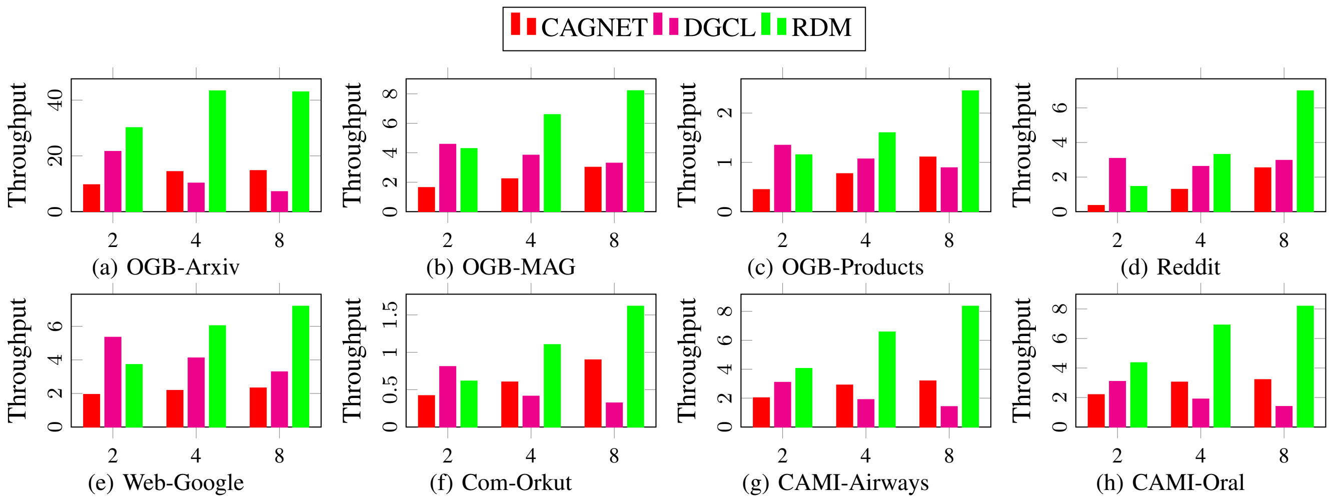


Fig. 9: Training throughput (epochs per second): 2-Layer GCN; hidden-size=256; X-axis: #GPUs.

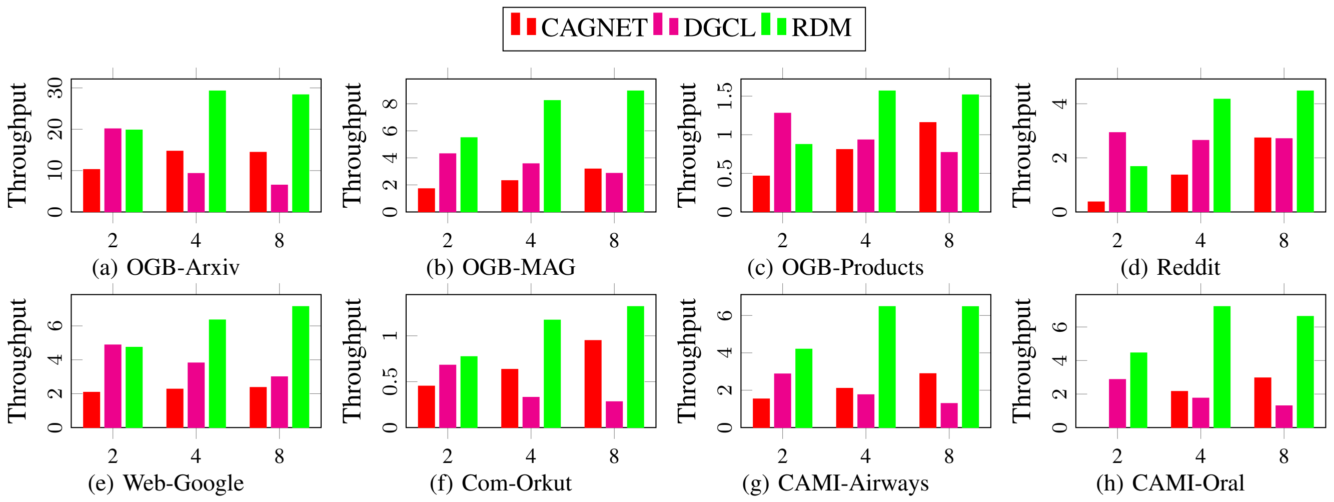


Fig. 10: Training throughput (epochs per second): 3-Layer GCN; hidden-size=128; X-axis: #GPUs.

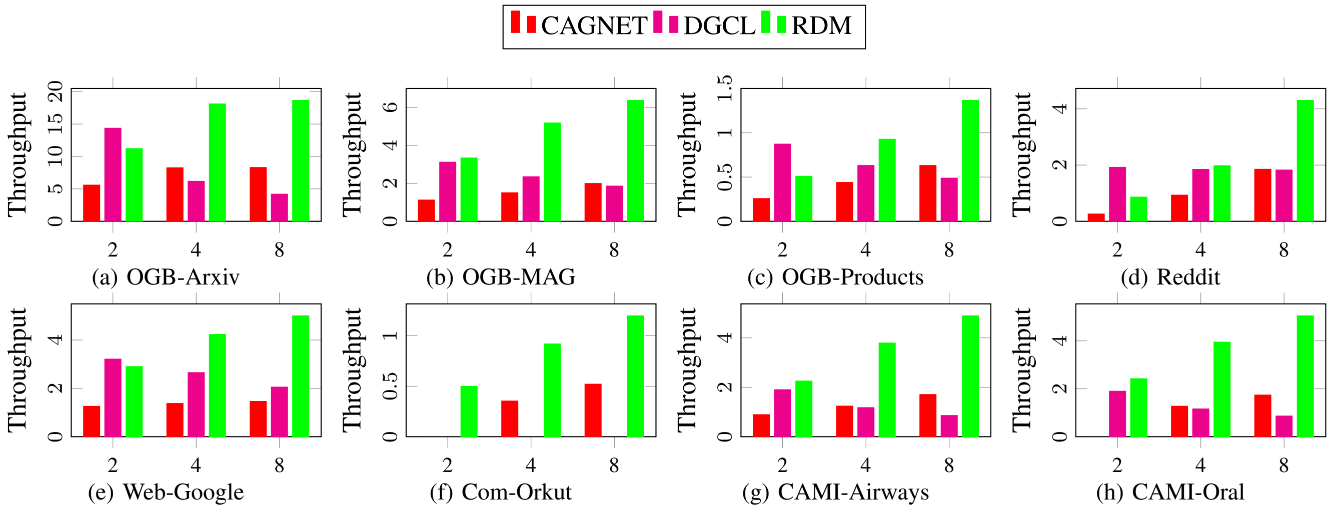


Fig. 11: Training throughput (epochs per second): 3-Layer GCN; hidden-size=256; X-axis: #GPUs.

0.001 for GraphSAINT-RDM on the Metagenomics datasets to enhance training stability. Each model was trained for 100 epochs, and the arithmetic mean for achieved throughput (epochs per second) is reported.

B. Throughput Comparison: Full-Batch GNN

We first compare the training throughput of GNN-RDM against CAGNET and DGCL. The reported throughput for GNN-RDM are based on the best of the model predicted SpMM and GEMM candidate orders. Among multiple CAGNET algorithms, we compare against the 1.5D algorithm since it was shown to be the algorithm with the best throughput by the authors. Fig. 8-11 present throughput data for the 8 datasets on 2/4/8 GPUs, for 2/3 GNN layers and 128/256 hidden features. All three implementations compute identical outputs, with small differences due to reordering of floating point operations across the schemes.

Compared to CAGNET, RDM achieves higher training throughput for all of the datasets across all GPU configurations (for a few cases CAGNET results could not be obtained since it ran out of memory). Overall, RDM's achieved throughput is significantly higher than DGCL, especially on 8 GPUs. For a couple of the benchmarks (OGB-Products and Reddit) DGCL is faster than RDM on 2 GPUs, but RDM is faster than DGCL on 4 and 8 GPUs. RDM exhibits much better scalability as the number of GPUs is increased because the total volume of inter-GPU communication remains constant, while the volume of data movement increases with the number of GPUs for CAGNET and DGCL.

Table VII summarizes the performance data in Fig. 8-11 as the geometric mean of speedup achieved by RDM over CAGNET and DGCL over all datasets. The speedup over CAGNET is at least 2× for all cases - 2, 4, or 8 GPUs, 2-layer or 3-layer network, 128 or 256 hidden features. When compared with DGCL, RDM performance is generally worse on 2 GPUs, but over 2× (between 2.1× and 2.54×) better on 4 GPUs and over 3× (between 3.13× and 3.74×) better on 8 GPUs.

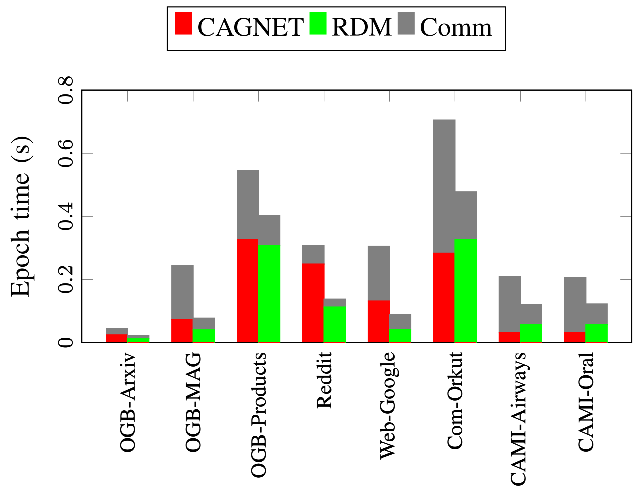


Fig. 12: Communication in CAGNET and RDM for 8 GPUs.

TABLE VII: Geometric mean of speedup of RDM over CAGNET and DGCL across the 8 datasets.

GPUs	Layers	Features	Speedup vs. CAGNET	Speedup vs. DGCL
2	2	128	2.29	0.91
		256	2.36	0.92
	3	128	2.49	1.02
		256	2.48	0.84
4	2	128	2.38	2.30
		256	2.54	2.17
	3	128	2.62	2.54
		256	2.68	2.10
8	2	128	2.04	3.13
		256	2.56	3.74
	3	128	2.00	3.27
		256	2.65	3.63

Communication Time: For the eight datasets, Figure 12 shows a breakdown of total epoch time into the time spent on computation versus communication by CAGNET and RDM, for the 2-layer GCN with 128 hidden features. The total time spent in communication (the gray region in the bars) is lower for RDM, often by a significant amount. For some cases, the compute time (red or green region) for RDM is lower than CAGNET but higher in some. This is in part due to the difference in total number of operations for different order of SpMM/GEMM, and in part due to the different width of the dense-matrix slices processed by RDM and CAGNET - the lower slice size for RDM can result in reduced data reuse and lower compute throughput. The ratio of epoch times and communication times between CAGNET and RDM for all four networks (2/3 layers and 128/256 hidden features) are presented in Table IX. RDM is consistently faster per-epoch, largely due to the significantly lower communication overhead.

TABLE VIII: Assessment of analytical model for optimal SpMM/GEMM order: 2-layer GCN with 128 hidden features.

Dataset	#GPUs	Pareto optimal	Non-pareto optimal
OGB-Arxiv	2	30	36-50
	4	22	25-38
	8	22	25-42
OGB-MAG	2	126	147-335
	4	85	103-221
	8	76	97-226
OGB-Products	2	859	756-969
	4	622	422-692
	8	402	482-662
Reddit	2	384-418	1309-1864
	4	157-168	775-985
	8	138-186	344-489
Web-Google	2	135-180	200-250
	4	95-139	145-198
	8	90-125	126-181
Com-Orkut	2	772-909	846-1045
	4	445-626	510-757
	8	473-525	546-677
CAMI-Airways	2	182-203	241-337
	4	119-138	147-241
	8	116-137	113-235
CAMI-Oral	2	174-193	226-320
	4	117-134	137-236
	8	114-143	124-224

Model Validation: We assessed the effectiveness of our analytical modeling for ordering of SpMM/GEMM operations by performing runs with all possible orderings of SpMM/GEMM. Table VIII lists the measured epoch time for the Pareto optimal configurations from the analytical model, as well as all non-Pareto-optimal configurations. With very few exceptions (e.g., OGB-Products on 2 and 4 GPUs), the model-predicted configuration(s) had better performance, sometimes with very significant differences (e.g., Reddit).

C. Evaluation with Cluster Sampling

The experiments above use RDM in full-batch GCN training (i.e., the entire set of vertices is processed together, rather than in small batches). In this section, we evaluate a redistribution-based GNN implementation for the cluster sampling approach of GraphSAINT [17], a recent GNN approach that uses sampling to generate independent subgraphs for training and has been shown to exhibit high accuracy. Each subgraph is parallelized using our RDM scheme.

We also extend a DGL implementation of GraphSAINT to use Distributed Data Parallelism (DDP) to run a batch/subgraph on each GPU (G) in parallel. Since subgraphs are run in a fully distributed manner, DDP can be expected to exhibit good scalability. However, when using DDP, the results of each of the G distributed forward and backward passes are reduced and the model weights are updated using the optimizer. If there are S subgraphs (i.e., batches) and G GPUs, there will be S/G weight updates. Therefore, DDP increases the effective batch size, which can have a negative effect on the rate of model convergence. In contrast, GraphSAINT-RDM computes the forward and backward pass for a single subgraph using distributed GPU resources. This allows the model weights to be updated after each subgraph, independent of the number of GPUs being used.

Figure 13 shows plots of test accuracy as a function of training time for the GraphSAINT-DGL implementation using DDP, the RDM based GraphSAINT, and the full batch GCN RDM implementation, for execution on 8 GPUs, using a 2-layer GNN with 128 hidden features. Trends are similar with use of 256 hidden features and 3-layer networks; details are omitted due to space limitations. Of the 8 datasets we experimented, two do not have training datasets associated with them (Com-Orkut, Web-Google) and are therefore not presented here. For the metagenomics datasets, it is not common practice to use sampling because of loss of accuracy from sampling. As expected, we observe that with the CAMI Oral and CAMI Airways dataset, full-batch training (GCN-RDM) achieves significantly higher accuracy than GraphSAINT-RDM or GraphSaint-DGL. For the other datasets all three schemes achieve comparable test accuracy, but for OGB-Products and Reddit, GraphSAINT-RDM and GraphSAINT-DGL converge much faster than GCN-RDM.

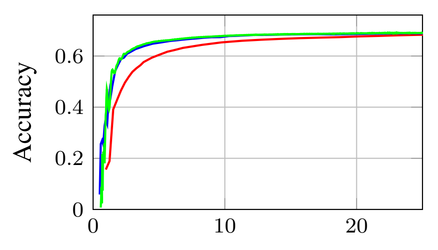
Our experimental results highlight that RDM can also be used effectively in cluster-sampling GNN schemes. However, there exist datasets where a sampling-based approach is inferior to full-batch training, where GCN-RDM is very effective. Whether full-batch training or sampling-based training is more effective for a dataset (higher test accuracy or comparable test accuracy but faster convergence), the RDM approach developed in this paper is very effective for both scenarios for distributed multi-GPU execution.

D. Space Requirement

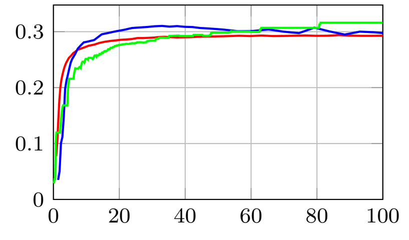
The space requirement for GNN-RDM for 8 GPUs is shown in Table X, for different values of R_A. There is an inverse relationship between the amount of memory used and the amount of inter-node data communication as described in section III-E. The case of $R_A = P$ corresponds to full replication of the graph across all GPUs, while the other extreme of $R_A = 1$ corresponds to CAGNET. The maximum possible value for R_A is chosen based on the available GPU memory.

TABLE IX: Ratio of CAGNET's Epoch time and Communication time over RDM's times.

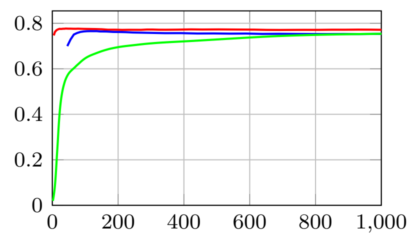
Dataset	**2-Layer Hidden:128**		**2-Layer Hidden:256**		**3-Layer Hidden:128**		**3-Layer Hidden:256**	
	Epoch	**Comm**	**Epoch**	**Comm**	**Epoch**	**Comm**	**Epoch**	**Comm**
OGB-Arxiv	1.98	1.87	2.92	2.70	1.97	1.82	1.93	1.54
OGB-MAG	3.15	4.60	2.72	3.26	2.82	3.94	3.20	3.99
OGB-Products	1.35	2.31	2.21	3.74	1.06	1.64	1.76	1.83
Reddit	2.24	2.37	2.76	2.54	1.63	2.52	2.33	2.91
Web-Google	3.47	3.71	3.09	2.77	3.02	3.02	2.65	2.21
Com-Orkut	1.48	2.79	1.80	2.14	1.35	2.16	2.00	2.40
CAMI-Airways	1.74	2.83	2.71	4.48	2.25	4.44	2.87	4.10
CAMI-Oral	1.68	2.64	2.68	4.13	2.24	4.05	2.91	4.00



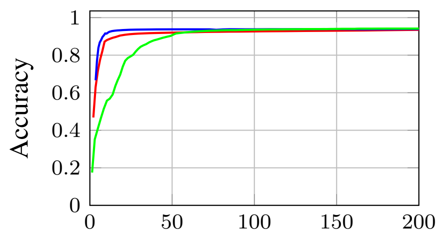
(a) OGB-Arxiv



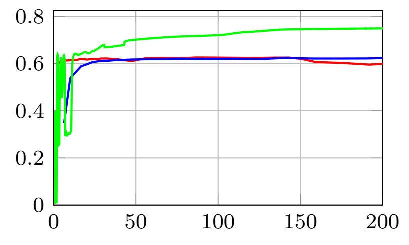
(b) OGB-MAG



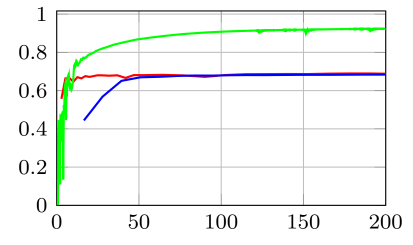
(c) OGB-Products



(d) Reddit



(e) CAMI-Airways



(f) CAMI-Oral

Fig. 13: Test accuracy versus time for GCN-RDM, GraphSAINT-RDM, and GraphSAINT-DGL: 2-layers, 128 hidden features, 8 GPUs (Time in seconds).

TABLE X: Space requirement for CAGNET and RDM. R_A defines how many replicas of the adjacency matrix A are stored. Data is for per-GPU space requirement in MBytes, for distributed GCN on 8 GPUs.

Dataset	**CAGNET**	**GNN-RDM**		
		$R_A = 2$	$R_A = 4$	$R_A = 8$
OGB-Arxiv	26MB	28MB	32MB	39MB
OGB-MAG	618MB	650MB	713MB	840MB
OGB-Products	430MB	522MB	708MB	1.1GB
Reddit	262MB	434MB	779MB	1.5 GB
Web-Google	220MB	227MB	243MB	273MB
Com-Orkut	723MB	898MB	1.3GB	2GB
CAMI Airways	239MB	273MB	342MB	479MB
CAMI Oral	239MB	270MB	332MB	457MB

VI. Related Work

Numerous efforts have addressed distributed GNNs on CPUs and/or GPUs. PyTorch Geometric (PyG) and Deep Graph Library (DGL) [24] are two widely used GNN training frameworks. PyG is compatible with PyTorch and supports training on CPU, single GPU and multiple GPUs. DGL is compatible with PyTorch, MXNet and TensorFlow. DistDGL [10] is the module integrated in DGL to support distributed model training.

We found the DGL-based implementation of GraphSAINT to be faster than PyG and so we used the former in our evaluation.

CAGNET [4] implements a number of SpMM algorithms, corresponding to 1D, 1.5D, 2D, and 3D schemes for distributed full-batch GNN training. DGCL [8] is a communication library that targets high-throughput full-batch GNN training. DGCL supports partition-based training and makes trade-offs between graph replication and communication in the distributed setting. It also leverages high-performance links such as NVLink to accelerate training. DistGNN [2] improves upon DGL for full-batch distributed training for GNNs on CPU clusters. DistGNN uses a minimum vertex-cut strategy to partition graphs, and optimizes memory management and cache reuse. The LIBXSMM library [25] is used in DistGNN to implement loop reordering and vectorization to provide high instruction-level parallelism. ROC [3] is a framework for partition-based distributed GNN training. ROC first trains a cost model to decide the optimal graph partitioning strategy to reduce communication time between devices and uses dynamic programming to optimize GPU memory usage. In this paper, we have performed comparisons with two state-of-the-art full-batch distributed GNN frameworks for GPUs: CAGNET and DGCL.

A number of efforts have sought to optimize distributed GNN using different forms of sampling. NEXTDOOR [5] is an efficient sampling algorithm that speeds up sampling time on GPUs by load balancing and caching of edges. They also provide an API for different types of sampling.

The 2PGraph [6] system uses METIS to reorder the graph to reduce the communication cost. It also proposes a fast sampling algorithm to improve the sampling time similar to Jangda et al. [5].

P3 [7] develops a distributed GNN approach that samples the sparse graph and computes aggregated results in the first layer by distributing input features so that each node has feature sets of a distinct group of vertices.

The RDM-based approach is also applicable for other forms of sampling besides GraphSAINT, but it is beyond the scope of this paper.

VII. Conclusion

This paper presents a new approach to communication-efficient implementation of distributed multi-GPU GNNs based on the redistribution of dense matrices between row-wise and column-wise partitions, along with communication-free sparse and dense matrix multiplication. Experimental evaluation demonstrates performance improvement over state-of-the-art publicly available multi-GPU GNN implementations.

Acknowledgments

We thank the reviewers for their numerous suggestions to improve the paper. The support and resources from the Center for High Performance Computing at the University of Utah are gratefully acknowledged. This work was supported in part by the U.S. National Science Foundation through award 2009007 and the Advanced Scientific Computing Research (ASCR) program within the Office of Science of the Department of Enery (DOE) under contract number DE-AC02-05CH11231, 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.

References

[1] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[2] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. Kalamkar, N. K. Ahmed, and S. Avancha, "Distgnn: Scalable distributed training for large-scale graph neural networks," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[3] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the accuracy, scalability, and performance of graph neural networks with ROC," *Machine Learning and Systems*, vol. 2, pp. 187–198, 2020.

[4] A. Tripathy, K. Yelick, and A. 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.

[5] A. Jangda, S. Polisetty, A. Guha, and M. Serafini, "Accelerating graph sampling for graph machine learning using gpus," in *Sixteenth European Conference on Computer Systems*, 2021, pp. 311–326.

[6] L. Zhang, Z. Lai, S. Li, Y. Tang, F. Liu, and D. Li, "2PGraph: Accelerating GNN training over large graphs on GPU clusters," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 103–113.

[7] S. Gandhi and A. P. Iyer, "P3: Distributed deep graph learning at scale," in *15th USENIX Symposium on Operating Systems Design and Implementation OSDI 21*, 2021, pp. 551–568.

[8] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu, "Dgcl: An efficient communication library for distributed GNN training," in *Sixteenth European Conference on Computer Systems*, 2021, pp. 130–144.

[9] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, "*NeuGraph*: Parallel deep neural network computation on large graphs," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 443–458.

[10] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, "Distdgl: distributed graph neural network training for billion-scale graphs," in *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 2020, pp. 36–44.

[11] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[12] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[13] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[14] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen, "Understanding and bridging the gaps in current GNN performance optimizations," in *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 119–132.

[15] A. Lamurias, M. Sereika, M. Albertsen, K. Hose, and T. D. Nielsen, "Metagenomic binning with assembly graph embeddings," *bioRxiv*, 2022.

[16] X.-M. Zhang, L. Liang, L. Liu, and M.-J. Tang, "Graph neural networks and their current applications in bioinformatics," *Frontiers in genetics*, vol. 12, 2021.

[17] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graphsaint: Graph sampling based inductive learning method," *arXiv preprint arXiv:1907.04931*, 2019.

[18] J. Dong, D. Zheng, L. F. Yang, and G. Karypis, "Global neighbor sampling for mixed cpu-gpu training on giant graphs," *arXiv preprint arXiv:2106.06150*, 2021.

[19] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *arXiv preprint arXiv:2005.00687*, 2020.

[20] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[21] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[22] A. Wickramarachchi and Y. Lin, "Metagenomics binning of long reads using read-overlap graphs," in *RECOMB International Workshop on Comparative Genomics*. Springer, 2022, pp. 260–278.

[23] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[24] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang, "Deep graph library: Towards efficient and scalable deep learning on graphs," *CoRR*, vol. abs/1909.01315, 2019. [Online]. Available: http://arxiv.org/abs/1909.01315

[25] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "Libxsmm: accelerating small matrix multiplications by runtime code generation," in *SC'16: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 981–991.