Rethinking Graph Data Placement for Graph Neural Network Training on Multiple GPUs

Shihui Song
The University of Iowa
Iowa City, IA, USA
shihui-song@uiowa.edu

Peng Jiang
The University of Iowa
Iowa City, IA, USA
peng-jiang@uiowa.edu

Abstract

Graph partitioning is commonly used for dividing graph data for parallel processing. While they achieve good performance for the traditional graph processing algorithms, the existing graph partitioning methods are unsatisfactory for data-parallel GNN training on GPUs. In this work, we rethink the graph data placement problem for large-scale GNN training on multiple GPUs. We find that loading input features is a performance bottleneck for GNN training on large graphs that cannot be stored on GPU. To reduce the data loading overhead, we first propose a performance model of data movement among CPU and GPUs in GNN training. Then, based on the performance model, we provide an efficient algorithm to divide and distribute the graph data onto multiple GPUs so that the data loading time is minimized. For cases where data placement alone cannot achieve good performance, we propose a locality-aware neighbor sampling technique to further reduce the data movement overhead without losing accuracy. Our experiments with graphs of different sizes on different numbers of GPUs show that our techniques not only achieve smaller data loading time but also incur much less preprocessing overhead than the existing graph partitioning methods.

CCS Concepts: • **Computing methodologies** → **Parallel algorithms**; • **Software and its engineering** → **Distributed memory**.

Keywords: graph neural network, data loading

ACM Reference Format:
Shihui Song and Peng Jiang. 2022. Rethinking Graph Data Placement for Graph Neural Network Training on Multiple GPUs. In *2022 International Conference on Supercomputing (ICS '22), June 28–30, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3524059.3532384

1 Introduction

Graph Neural Networks (GNNs) have emerged as the state-of-the-art models for machine learning tasks on graphs [4, 6, 16, 31, 35, 36]. Due to their superior accuracy, GNNs play an important and increasing role in many application domains, including content recommendation [30], traffic prediction [34], drug discovery [19], and molecular property prediction [6].

Different from the traditional graph processing algorithms, GNNs make predictions on graphs with node features. The basic idea is to learn a vector representation (also called embedding) of each node by recursively aggregating the features of neighboring nodes. The embeddings are used for downstream tasks such as node classification [4, 16] or link prediction [35, 36]. As the feature of each node contains hundreds or thousands of attributes, the data processed by GNNs are much larger than the graph structure itself. For instance, a graph with 10M nodes and feature vector length of 1K needs at least 40GB memory to store node features. This exceeds the memory capacity of most GPUs, making it challenging to train GNNs efficiently on large graphs.

To support large graphs, a straightforward approach is to partition the graph and distribute the node features onto multiple GPUs. The current GNN systems either take an off-the-shelf graph partitioning method [21, 37] or propose heuristic partitioning methods [14, 20] for this task. For example, DGL [37] adopts METIS [15] graph partitioning. It assumes that the graph can be entirely stored on multiple GPUs. PaGraph [20] uses a heuristic partitioning method based on training nodes. For large graphs that exceed the memory capacity of multiple GPUs, it stores the graph on CPU and buffers the most frequently accessed nodes of each partition on GPU.

We find that the existing graph partitioning methods are unsatisfactory for GNN training in terms of both data loading efficiency and preprocessing overhead. Figure 1 shows the breakdown epoch time of GNN training with the graph partitioning methods in PaGraph [20] and DGL [37]. We assume that the GPU memory is small and we can only store 20% of nodes that are most frequently accessed on each GPU. The figure shows that loading the input features is a performance bottleneck in GNN training when the graph data cannot be entirely put on GPU. (The data loading time of DGL reported in PaGraph paper is even longer because they do not use GPU buffer for DGL.) With two GPUs, the data loading time accounts for more than 40% of the total execution time. If we add two more GPUs, we have 80% of nodes stored on four GPUs; however, the data loading still takes about 40% of the total execution time. The results suggest that the existing graph partitioning methods do not utilize the aggregate GPU memory efficiently for GNN training. The graph partitioning procedure is also expensive. PaGraph partitioning has $O(N^2)$ time complexity where N is the number of nodes in graph, and it takes a long time for large graphs. DGL (METIS) partitioning is faster, but it runs out of memory for the large graphs used in our experiments.

To overcome the limitations of the existing systems, we rethink the graph data placement problem for data-parallel GNN training in this work. The question we aim to answer is, given a large graph that cannot be stored on a single device, how can we divide and place the graph data across devices so that the overall data movement overhead is minimized? We first present a performance model of data movement among CPU and GPUs in GNN training and formulate the data placement problem as an optimization problem. Then, we propose an efficient algorithm to find the optimal data

ICS '22, June 28–30, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9281-5/22/06.
https://doi.org/10.1145/3524059.3532384

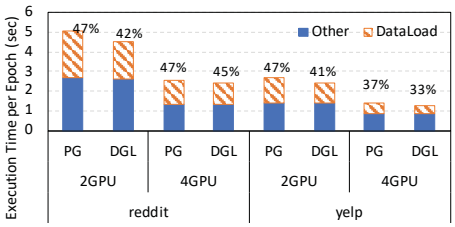


Figure 1. Ratio of data loading time to total execution time in GNN training with PaGraph (PG) and DGL.

placement strategy. Compared to the existing graph-partitioning-based methods, our data placement strategy not only achieves a lower data movement overhead but also allows global neighbor aggregation and shuffling, thus leading to better convergence of the training algorithm. Moreover, our data placement algorithm is fast, with $O(N)$ time and space complexity. The algorithm is run only once before the training process, and its overhead is negligible compared to the total training time.

In cases where the performance with the optimal data placement is still unsatisfactory (for example, when the GPU memory is small and most nodes are stored on CPU), we propose a *locality-aware neighbor sampling* technique to further reduce the data loading overhead. The idea is to increase the access frequency of nodes on GPU so that accesses to CPU are reduced. We show that our sampling technique preserves the unbiased estimation of neighbor aggregation result in each layer. By carefully adjusting the parameters of our sampling method, we can reduce the data loading time without impairing the convergence speed of the training algorithm.

We evaluate our technique with graphs of different sizes on 2~8 GPUs. The results show that our data placement strategy reduces the data loading time by 1.2x to 3.3x compared with the existing graph partitioning method. Combining our data placement strategy with locality-aware neighbor sampling, we achieve up to 4.4x speedup for data loading in large-scale GNN training.

2 Background

We now give a background on GNN and sampling-based GNN training. Figure 2b shows the computation of a two-layer Graph Convolutional Network (GCN) [16]. We use $\mathbf{x}_i^{(0)}$ to denote the feature vector of node-i. The feature vectors of each node and its neighboring nodes are first aggregated by a *Mean* function, and we use $\mathbf{x}_i^{(l)}$ to denote the aggregation result of node-i in layer l. This aggregation operation is also called graph convolution and is the key difference between different GNN models. For example, GraphSAGE [7] computes the *Max* of the neighboring nodes, while some other models use *Sum* [29]. The aggregation result is given to a linear function (*Linear*) and an activation function (*ReLU*) to obtain the intermediate embedding $\mathbf{y}_i^{(1)}$. The intermediate embeddings are further aggregated for a few layers to obtain the output embeddings.

To train a GNN, we sample a batch of training nodes and compute their output embeddings in each iteration. The output embeddings are used to make predictions, and the predicted values are compared with the ground-truth labels to obtain a loss. The loss is then back-propagated through the network to adjust the model parameters. From Figure 2b, we can see that the number of nodes involved in the computation in each training iteration is exponential w.r.t. the number of layers. In this example, we need to load and aggregate the features of all nodes in the graph to compute the output embedding of a single node $\mathbf{x}_3$. This incurs a large data movement and computation overhead when the graph is large.

To reduce the computation, various neighbor sampling methods have been proposed for GNN training [2, 3, 7, 18, 30, 32, 41]. The idea is to sample a subset of neighbors and estimate the aggregation results based on the sampled nodes. As shown in Figure 2c, instead of computing the accurate value of $\mathbf{x}_1^{(1)}$ with all of $\mathbf{x}_0^{(0)}, \mathbf{x}_1^{(0)}, \mathbf{x}_2^{(0)}$ and $\mathbf{x}_3^{(0)}$, we can estimate the mean of the four feature vectors by randomly sample three of them. Suppose the sampling probabilities of the four vectors are p_0, p_1, p_2, p_3 and node-0, 2, 3 are sampled. The estimate can be computed as $\mathbf{x}_0^{(0)}/(4p_0) + \mathbf{x}_2^{(0)}/(4p_2) + \mathbf{x}_3^{(0)}/(4p_3)$. More generally, to estimate the aggregation result of node i's neighbors i.e., $\sum_{j \in Neighbor(i)} w_{ij}\mathbf{x}_j$ where $\mathbf{x}_j$ is the feature of node j in a certain layer, and w_{ij} is the weight of edge-(i, j), we can define a sequence of random variables $\xi_j \sim \text{Bernoulli}(q_j)$ where q_j is the probability that node j in the neighbor list is sampled. We can have an unbiased estimate of the aggregation result as

$$\sum_{j \in Neighbor(i)} \frac{1}{q_j} \xi_j w_{ij} \mathbf{x}_j. \tag{1}$$

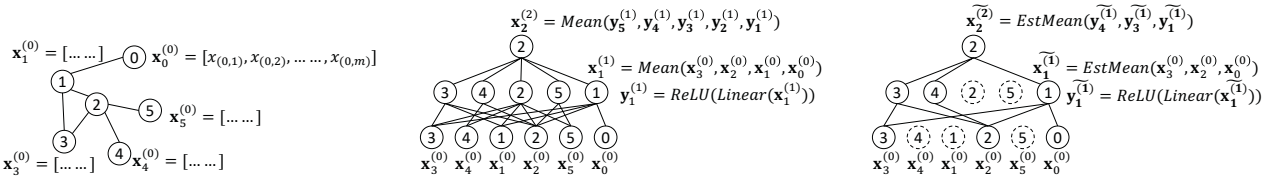
Different sampling methods have different ways of determining q_j, but they all use this formula to estimate the results.

Neighbor sampling allows us to train GNNs on very large graphs with millions to hundreds of millions of nodes. Due to the GPU memory limitation, the graph has to be stored on CPU and copied to GPU during the training process. There are mainly two types of data need to be moved. One is the graph structure (i.e., the sampled adjacency matrix); the other one is the input feature vectors. The existing sampling-based GNN training systems [30, 32, 41] generate the sampled adjacency matrices with multiple processes on CPU and copy them to GPU asynchronously in each iteration. Since the sampled adjacency matrices are small and sparse, the overhead of copying the sampled adjacency matrices can be hidden by overlapping the data transfer with the training procedure on GPU. The main data movement overhead is for copying the input features. As shown in Figure 1, loading the input features is a performance bottleneck of sampling-based GNN training. The focus of this work is to reduce this data loading overhead.

3 Minimizing Data Movement For Input Features

We consider the data movement problem in data-parallel GNN training with sampled neighbor aggregation on multiple GPUs within a single machine. That is, each GPU maintains a copy of the GNN model and computes a local gradient with sampled neighbor aggregation, and the gradients are averaged among all GPU in each iteration. This is the most common setup for training GNNs on large graphs [9]. The GPUs can be organized into groups of two connected through NVLink Bridge, as shown Figure 3a. Or they can be connected all together through NVSwitch as shown in Figure 3b. These are typical configurations of modern GPU servers/workstations for deep learning. The NVLinks among GPUs are not necessary for our algorithm to work, but they allow us to use the aggregate memory of multiple GPUs more efficiently.

We assume that each GPU-i can only store the feature vectors of a set of nodes B_i. If the memory of each GPU is large enough to

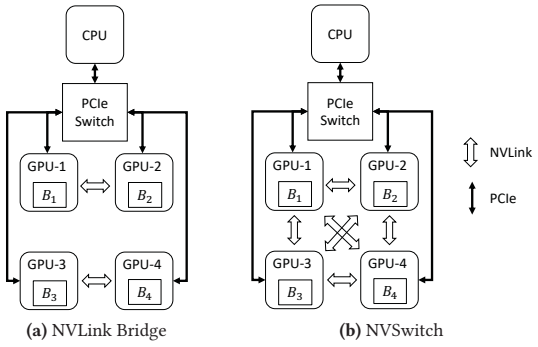


(a) An example graph: $\mathbf{x}_i^{(0)}$ represents the feature vector of node-i

(b) Compute the output embedding of node-2 with full neighbor aggregation

(c) Estimate the output embedding of node-2 with sampled neighbor aggregation

Figure 2. Computation of a two-layer GCN.



(a) NVLink Bridge **(b)** NVSwitch

Figure 3. Typical configurations of multi-GPU systems with interconnects among GPUs. We store the input features of a set of nodes B_i on each GPU-i.

hold the entire graph along with the GNN model and intermediate embeddings, the problem will be trivial. However, this is not the case for large graphs and the current GPUs of 10~30GB of memory. With this setup, we now give a performance model of the data movement of input features. Then, we will provide an efficient algorithm to minimize the data movement overhead.

3.1 Performance Model

Since GPU-i stores a set of nodes B_i, the nodes that are stored on all GPUs are

$$B_{gpu} = B_1 \cup \ldots \cup B_n.$$

We assume that the cost of reading a node on the same GPU is zero, the cost of reading a node on a different GPU is C_{gpu}, and the cost of reading from CPU is C_{cpu}. In every training iteration, each GPU needs to read input features from either CPU or one of the GPUs. Suppose GPU-i needs to read a set of nodes S_i. We denote the remote nodes that are not on GPU-i as

$$R_i = S_i \setminus B_i = S_i - (S_i \cap B_i). \tag{2}$$

For every node in R_i, we need to decide whether to fetch it from CPU or from a different GPU. That is, we need to divide R_i into two sets $R_{i,cpu}$ and $R_{i,gpu}$, where $R_{i,cpu}$ are the nodes read from CPU and $R_{i,gpu} \in Bgpu$ are the nodes read from other GPUs. The cost of reading S_i can then be written as

$$C_{cpu}|R_{i,cpu}| + C_{gpu}|R_{i,gpu}|.$$

If $C_{gpu} < C_{cpu}$, we should fetch as many nodes as possible from GPUs. The minimum cost is achieved when $R_{i,gpu} = R_i \cap B_{gpu}$. If $C_{gpu} \geq C_{cpu}$, we should fetch all nodes from CPU, i.e., $R_{i,gpu} = \emptyset$ and $R_{i,cpu} = R_i$. More formally, we define the cost of reading S_i by GPU-i as

$$C_i(B) = \begin{cases} C_{cpu}|R_i \setminus B_{gpu}| + C_{gpu}|R_i \cap B_{gpu}|, & \text{if } C_{gpu} < C_{cpu}; \\ C_{cpu}|R_i|, & \text{if } C_{gpu} \geq C_{cpu}. \end{cases} \tag{3}$$

Note that C_i is a function of $B = \{B_1, \ldots, B_n\}$ and is dependent on S_i. The value we want to minimize is its expectation $\mathbb{E}_{S_i \sim \mathcal{D}}[C_i]$. Since the GPUs run in parallel, we minimize the maximum cost across all GPUs. The problem can then be formulated as a constrained optimization problem:

$$\begin{aligned} \min \quad & \max_{i \in [1,n]} \left(\mathbb{E}_{S_i \sim \mathcal{D}}[C_i(B)]\right), \\ \text{subject to} \quad & |B_i| \leq BSIZE, \quad i = 1, \ldots, n \end{aligned} \tag{4}$$

where $BSIZE$ is the maximum number of nodes that can be stored on each GPU. Our goal is to find the optimal configuration of B_i that minimize $\max_i(\mathbb{E}_{S_i}[C_i])$ within the memory size limit.

3.2 An Efficient Solution

To solve Problem (4), we first consider the trivial case where $C_{gpu} \geq C_{cpu}$, i.e., the GPU is not connected with other GPUs through NVLink. In this case, we should read data from CPU. According to (3) and (2), the optimization objective can be written as

$$\begin{aligned} \max_i(\mathbb{E}_{S_i}[C_i]) &= C_{cpu} \max_i(\mathbb{E}_{S_i}[|R_i|]) \\ &= C_{cpu} \max_i(\mathbb{E}_{S_i}[|S_i \setminus B_i|]) \\ &= C_{cpu}\mathbb{E}_S[|S|] - C_{cpu} \min_i \mathbb{E}_{S_i}[|S_i \cap B_i|]) \end{aligned} \tag{5}$$

where $\mathbb{E}_S[|S|]$ is the expected number of sampled input nodes and is a constant number for all GPUs. It is easy to verify that, when $\mathbb{E}_{S_i}[|S_i \cap B_i|]$ is maximized for every GPU-i, (5) achieves the minimum value. Suppose there are N nodes in the graph. We can represent B_i as a vector $\mathbf{b_i} \in \{0,1\}^N$ where $\mathbf{b_i}[j] = 1$ indicates that node-j is stored on GPU-i. We can also represent S_i as a vector of random variables $\mathbf{s_i} = [\xi_1, \xi_2, \ldots \xi_N]^T$ where $\xi_j \sim$ Bernoulli(p_j) and p_j is the probability of node j being sampled. Then, we can rewrite the objective in (5) as

$$C_{cpu}\mathbb{E}[|\mathbf{s}|] - C_{cpu} \min_i \left(|\mathbf{b_i}^T \mathbb{E}[\mathbf{s_i}]|\right)$$

where $|\cdot|$ represents the ℓ_1 norm and $\mathbb{E}[\mathbf{s_i}] = [p_1, p_2, \ldots, p_N]^T$. To minimize the cost, we need to maximize $|\mathbf{b_i}^T \mathbb{E}[\mathbf{s_i}]|$ for each GPU-i. This leads to the first rule of our data placement strategy:

> Rule 1: If a GPU is not connected to other GPUs, we store nodes with the highest sampling probability on it.

Algorithm 1: Distributing node features onto multiple GPUs with fast interconnects

Input: α; #nodes: N; #GPUs: n; Sampling probability: p; Buffer size: $BSIZE$
Output: $B = \{B_1, \ldots, B_n\}$
/* Sort nodes by probability p in descending order */
1 V = sort_nodes(N, p);
/* Initialize buffer on each GPU with nodes of highest sampling probabilities */
2 **for** $i = 1$ **to** n **do**
3 $B_i = [V[0], V[1], \ldots V[BSIZE-1]]$;
4 $p\_sum = [0.0, \ldots, 0.0]$;
5 **for** $i = 0$ **to** $(\min(N, n \cdot BSIZE) - BSIZE - 1)$ **do**
6 **if** $i \bmod n == 0$ **then**
 /* Sort GPUs by p_sum in ascending order */
7 $ordered\_devices$ = sort_device(n, p_sum);
 /* Do not change last device in each round */
8 **if** $i \bmod n == n - 1$ **then continue**;
 /* Get device in the sorted order */
9 $device = ordered\_devices[i \bmod n]$;
 /* Select the next node in V */
10 $new\_node = V[i + BSIZE]$;
 /* Select a duplicate node on device */
11 $old\_node\_idx = BSIZE - 1 - \lfloor i/n \rfloor$;
12 $old\_node = V[old\_node\_idx]$;
 /* Check if the replacement is beneficial */
13 **if** $p_{(new\_node)} > \alpha \cdot p_{(old\_node)}$ **then**
 /* Replace old_node on device with new_node */
14 $B_{device}[old\_node\_idx] = new\_node$;
 /* Update p_sum */
15 $p\_sum[device] += p_{(new\_node)}$
16 **else break**;
17 **return** $\{B_1, \ldots, B_n\}$

This rule is intuitive – For a standalone GPU, storing the nodes that are most frequently accessed on the GPU gives us minimum overall data loading time.

We next consider the case where $C_{gpu} < C_{cpu}$, i.e., the GPU is connected with other GPUs through NVLink. According to (3) and (2), the optimization objective can be written as

$$\begin{aligned} &\max_i \left(\mathbb{E}_{S_i}\left[C_i\right]\right) \qquad (6)\\ =&\max_i \left(C_{cpu}\mathbb{E}_{S_i}\left[\left|R_i \setminus B_{gpu}\right|\right] + C_{gpu}\mathbb{E}_{S_i}\left[\left|R_i \cap B_{gpu}\right|\right]\right)\\ =&\max_i \left(C_{cpu}\mathbb{E}_{S_i}\left[|R_i|\right] - \left(C_{cpu} - C_{gpu}\right)\mathbb{E}_{S_i}\left[\left|R_i \cap B_{gpu}\right|\right]\right)\\ =&C_{cpu}\mathbb{E}_S\left[|S|\right] - \left(C_{cpu} - C_{gpu}\right)\mathbb{E}_S\left[|S \cap B_{gpu}|\right]\\ &\qquad - \min_i \left(C_{gpu}\mathbb{E}_{S_i}\left[|S_i \cap B_i|\right]\right). \end{aligned}$$

To minimize the cost, we need to maximize the last two terms. The last term suggests that each GPU stores the same set of nodes with the highest sampling probability. This configuration however hurts the overall performance because the second last term $\left(C_{cpu} - C_{gpu}\right)\mathbb{E}_S\left[|S \cap B_{gpu}|\right]$ suggests that we should store as many nodes as possible on all GPUs. The two terms illustrate a tradeoff between the data access efficiency on a single GPU and the overall efficiency on all GPUs. We now propose an algorithm that explores the tradeoff and find an optimal configuration.

As shown in Algorithm 1, we first obtain an ordered set of the nodes (V) such that $p_{(V[i])} \geq p_{(V[i+1])}$, and we store the nodes with the highest sampling probability on each GPU (line 2 and 3). This initial configuration achieves the maximum value for the last term in (6) but results in a small value for the second last term. To explore the tradeoff between the two terms, the algorithm tries to iteratively replace the duplicate nodes on different GPUs with new nodes from $V[BSIZE]$ to $V[N-1]$ (line 5). In the first iteration, it tries to replace $V[BSIZE-1]$ in B_1 with $V[BSIZE]$. By doing so, we decrease the last term of (6) by $C_{gpu}(p_{(V[BSIZE-1])} - p_{(V[BSIZE])})$. However, it also increases the second last term by $\left(C_{cpu} - C_{gpu}\right)p_{(V[BSIZE])}$. If the increase is greater than the decrease, i.e.,

$$C_{cpu}p_{(V[BSIZE])} > C_{gpu}p_{(V[BSIZE-1])},$$

this replacement is beneficial. More generally, in each iteration of the algorithm, we try to replace the duplicate node that has the lowest sampling probability (old_node) on one of the GPUs with the highest probability node that is not stored on GPU (new_node). If

$$p_{(new\_node)} > \frac{C_{gpu}}{C_{cpu}}p_{(old\_node)}, \qquad (7)$$

the replacement is beneficial, and we perform the replacement (line 14). Otherwise, the algorithm stops (line 16). Note that in the actual implementation we use a parameter $\alpha \in [0, 1)$ to check the condition in (7) because C_{cpu} and C_{gpu} are unknown. Since C_{cpu} and C_{gpu} are mostly determined by the hardware and are independent from the workload, we can tune this α for a hardware configuration once and use it for all the training tasks. To ensure each GPU has a similar data movement cost, the algorithm maintains the sum of sampling probabilities of the new nodes (p_sum) on each GPU. In each round of replacement, it selects the GPUs with p_sum in ascending order (line 6,7,9). We try to replace the same node on the first $n-1$ devices with new nodes in V. The last device is unchanged because we need to keep at least one copy of the node on GPU (line 8). Because the sampling probabilities of the new nodes are decreasing, this replacement order ensures that different GPUs have similar values of p_sum.

Algorithm 1 has $O(N)$ time complexity because it runs for at most N iterations and each iteration takes a constant time (assuming the number of devices n is a small constant). The space complexity is also $O(N)$.

Rule 2: For a group of GPUs with fast interconnects, we use Algorithm 1 to distribute the nodes onto different GPUs.

Note that Algorithm 1 may put duplicate nodes on different GPUs. This is the main difference between our data placement strategy and graph partitioning. Intuitively, for nodes with high sampling probability, we should store a copy of them on each GPU so that their features can be read locally. The number of duplicate nodes is also affected by the parameter α in the algorithm. A larger α means that the communication among GPUs is relatively expensive. The larger the α, the more likely the algorithm stops early, and thus the more duplicate nodes we may have.

Example: Let us consider the graph in Figure 2a. According to Figure 2b, to compute the output embedding of node-2 we need to access all of the six nodes. Similarly, to compute the output embedding of node-0 we need to access node-0, 1, 2, 3. It is easy to see that the access frequency of a node is the number of training nodes in

its L-hop neighbors where L is the number of GNN layers. In this example, the access frequency of the six nodes is $f = [4, 6, 6, 6, 5, 5]$. We use f/N_t (N_t is the number of training nodes) as an estimation of the sampling probability and give $p = [4/6, 1, 1, 1, 5/6, 5/6]$ to Algorithm 1. Suppose we have two GPUs connected with NVLink and each GPU can store at most 2 nodes. The algorithm first sorts the six nodes by p and obtain an ordered set of nodes $V = [1, 2, 3, 4, 5, 0]$. Initially, the algorithm stores the two nodes with highest sampling probability on both GPUs, i.e., $B_1 = [1, 2]$ and $B_2 = [1, 2]$. Suppose we set α to 0.3. In the first iteration of the algorithm, we try to replace node-2 on GPU-1 with node-3 in V. Because $p_3 > \alpha p_2$ ($1 > 0.3 * 1$), which means the replacement is beneficial, we perform this replacement, and B_1 becomes $[1, 3]$. The second iteration is skipped by line 8 in the algorithm because we want to keep node-2 on GPU-2. In the third iteration, we sort the two devices based on their p_sum. The p_sum of GPU-2 is smaller, so we try to replace node-1 on GPU-2 with node-4 in V. Because $p_4 > \alpha p_1$ ($5/6 > 0.3 * 1$), we change B_2 to $[4, 2]$.

During the training process, each GPU loads the input features from three places: its own memory, the memory of other GPUs in the same group, or the CPU memory. Each GPU has a dev_num array and a buf_pos array for locating the nodes: $dev\_num[i]$ indicates which device we should read node-i from and $buf\_pos[i]$ is the index of node-i in $dev\_num[i]$'s buffer. In the above example, we have $dev\_num = [-1, 1, 2, 1, 2, -1, -1]$, $buf\_pos = [0, 0, 1, 1, 0, 5, 6]$ on GPU-1. Here, we use -1 as the CPU device number. The two arrays indicate that node-0 should be read from location 0 of the CPU buffer, and node-1 should be read from location 0 of GPU-1's buffer. The dev_num and buf_pos array could be different on different GPUs. If two GPUs hold the same node, they will read the node from their local memory. In this example, the two arrays are the same on GPU-1 and GPU-2 because there is no duplicate nodes on the two GPUs.

4 Locality-Aware Neighbor Sampling

The same as any caching system, our data placement strategy works most efficiently when the access frequency is skewed, i.e., most accesses are made to a small number of nodes. This assumption holds in most cases due to the irregularity of real-world graphs. However, when the access frequency is less skewed and the GPU memory is small, the data loading might be expensive even with the optimal placement of graph data. To further reduce the data movement overhead, we present a *locality-aware neighbor sampling* technique in this section.

Our main idea is to increase the sampling probability of nodes on GPU. If a GPU accesses its local nodes or nodes on other GPUs in the same group more frequently, the overall data loading time will be reduced. More specifically, suppose B is the set of nodes stored on a group of GPUs. Our goal is to increase the access frequency of B in the input layer. To achieve this, we can directly increase the q_j in Formula (1) for j in B in the input layer. This however may not be effective because the nodes that are sampled in the input layer are determined by the nodes sampled in the second layer. If the second layer does not have nodes connected to nodes in B, we will not have q_j to increase in the input layer. Therefore, to have more nodes in B in the first layer, we can increase the q_j for j in $Neighbor(B)$ in the second layer. Similarly, to have more nodes in $Neighbor(B)$ in the second layer, we want to increase the q_j for nodes in $Neighbor(Neighbor(B))$ in the third layer.

Algorithm 2: Adjusting scale factor s based on data loading overhead

Input: lt; ut; $MAXS$; Ratio of data loading time to total execution time in previous epoch: r; Previous value of s: pre; Is s $fixed$

```
1 if fixed == false then
2     if s ≥ MAXS then fixed = true ;
3     else if r > ut then pre = s; s = s * 2 ;
4     else if r < lt and s! = 1 then s = (s + pre)/2 ;
5     else fixed = true ;
```

In order to preserve the relative importance of different nodes, we increase the sampling probability of nodes in the neighbor sets by multiplying them with the same factor s. To ensure the same number of neighbors are sampled in expectation, we need to rescale the new sampling probability so that their sum equals to the sum of the original sampling probability. More formally, we compute the new sampling probability of nodes with the following formula:

$$q_j^{new} = \begin{cases} sq_j/M & \text{if } j \in Q_l \\ q_j/M & \text{if } j \notin Q_l \end{cases} \tag{8}$$

where $M = \sum_j q_j / \left(\sum_{j \in Q_l} sq_j + \sum_{i \notin Q_l} q_j \right)$ is the normalization factor, and Q_l represents the neighbor set in layer-l. We have $Q_1 = B$ in the input layer, $Q_2 = Neighbor(B)$ in the second layer, and so on. The formula ensures that $q_i^{new}/q_j^{new} = q_i/q_j = x, \forall i, j \in Q$, meaning that node-$i$ is still x times more likely to sampled than node-j after probability increasing. The normalization factor ensures $\sum_j q_j^{new} = \sum_j q_j$, so the same number of neighbors will be sampled in expectation.

It is easy to see that our locality-aware neighbor sampling does not affect the unbiasedness of the estimation result. That is, we still have $\mathbb{E}\left[\sum_j \frac{1}{q_j} \xi_j w_{ij} \mathbf{x}_j\right] = \sum_j w_{ij} \mathbf{x}_j$. However, as the sampling probability is skewed towards the nodes on GPU, the estimation variance may increase, which may lead to slower convergence of the training algorithm. We explore this tradeoff between data loading efficiency and convergence rate by adjusting s adaptively. Initially, we set $s = 1$, and the original neighbor sampling method is used. At the end of each epoch, we check if the ratio of data loading time to total execution time is greater than an upper threshold ut. If it is greater, we multiply s by 2. If not, we check if the ratio is smaller than a lower threshold lt. If the ratio is greater than lt, we fix s and use it for the rest of the training process. If the ratio is smaller than lt, we set s to the average of its current value and its previous value. Algorithm 2 summarize this procedure. We set a maximum value $MAXS$ for s to ensure good convergence of the training algorithm. In our experiments, we find that if we set $MAXS$ to 8, the model can be trained to the same accuracy with the same number of training iterations.

For large graphs, the neighbor set Q_l can grow fast as the neural network goes deeper. Instead of maintaining all the neighboring nodes, we only store a small subset Q_l' that have the most connections to B in each layer. This is because the nodes are multiplied with the same scale factor and increasing the probability of many nodes cannot concentrate the memory accesses to a small number

Table 1. Graph datasets.

	reddit	yelp	products	papers100M	MAG240M
#nodes	233K	717K	2.4M	111M	122M
#edges	11.6M	7.0M	62M	1.6B	1.3B
feat_size	535MB	820MB	934MB	53GB	174GB

of nodes. In the extreme case, if Q_l contains all the nodes in the graph, the new sampling probability computed with (8) is the same as the original sampling probability. To obtain Q'_l, we represent B as a vector $v_1 \in \{0,1\}^N$ where $v_1[i] = 1$ indicates that node-i is in B. In the input layer, we simply set $Q'_1 = B$. In the second layer, we multiply v_1 with the adjacency matrix of the graph A and obtain a vector $v_2 = v_1 A$. The vector has N elements with $v_2[i]$ being the number of paths with one edge from node-i to the nodes in B. We sort the nodes based on v_2 and add the nodes with the largest values to Q'_2. Similarly, we can obtain $v_3 = v_2 A$. Each element $v_3[i]$ is the number of paths with two edges from node-i to the nodes in B. The nodes with the largest values of v_3 are added to Q'_3. This procedure is done before the training process after Algorithm 1 returns B. During the training process, we obtain the nodes to increase sampling probability by computing the intersection of the sampled nodes and the Q'_l in the corresponding layer. Since both sets are small, our locality-aware sampling incurs little overhead to the original sampling procedure.

Example: Let us consider again the graph in Figure 2a. According to the example in §3.2, node-1, 3 are stored on GPU-1 and node-2, 4 are stored on GPU-2. This gives us $Q'_1 = 1, 2, 3, 4$ and $v_1 = [0, 1, 1, 1, 1, 0]$. For the second layer, We have $v_2 = v_1 A = [1, 3, 4, 3, 2, 1]$. Suppose the maximum number of nodes we can store in Q'_l for $l > 1$ is three. The three nodes with largest value of v_2 (i.e., node-2, 1, 3) are added to Q'_2.

5 Evaluation

This section presents an evaluation of our data placement strategy and locality-aware sampling technique for GNN training on multiple GPUs.

5.1 Experimental Setup

Platform: Our experiments are conducted on a GPU workstation with two Intel Xeon Gold 6248 CPUs and eight Nvidia Tesla V100 GPUs connected all together through NVSwitch. The CPU RAM size is 512GB, and GPU memory size is 32GB. We run the experiments on 2~8 GPUs with two types of interconnections. The first type has every two GPUs connected to each other, as shown in Figure 3a. By only allowing a GPU to read data from the GPU next to it, we simulate systems with NVLink Bridges. We use '(2+2)GPU' to denote four GPUs with each two connected with NVLink, and '(2*4)GPU' to denote eight GPUs with each two connected with NVLink. The second interconnection type has all GPUs connected together, similar to the configuration in Figure 3b. We use '4GPU' ('8GPU') to denote four (eight) all-connected GPUs.

Datasets: We evaluate our system on five graphs as listed in Table 1. The first two graphs, `reddit` and `yelp`, are adopted from GraphSAINT [32]. The other three graphs, `products`, `papers100M` and `MAG240M`, are from the Open Graph Benchmark [10]. Among the graphs, `reddit`, `yelp` and `products` are relatively small. While they can be entirely put on our GPU, we use a limited buffer size on GPU to show how effective our techniques are on devices with smaller memory. The `papers100M` and `MAG240M` are large graphs. The original `MAG240M` graph is a heterogeneous graph with 122M paper nodes and 122M author nodes. We only use the paper nodes in our experiments since only the paper nodes have input features. The feature vectors of the two graphs cannot be entirely put on a GPU. We use them to show the effectiveness of our techniques for large-scale GNN training.

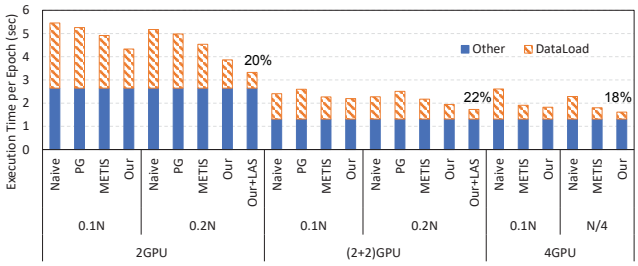


Figure 4. Breakdown execution time on `reddit` graph with feature buffer size 0.1N (53MB), 0.2N (107MB), and N/4 (134MB) on each GPU.

Baseline: We compare our data placement strategy with four graph partitioning methods: naive partitioning (`NAIVE`), random partitioning (`RAND`), METIS partitioning [15], and PaGraph partitioning (`PG`) [20]. Naive partitioning evenly divides graph nodes according to their indices and puts nodes of consecutive indices in each partition. Random partitioning performs random permutation to nodes and then divides the permuted nodes with naive partitioning. PaGraph partitions a graph based on training nodes. It iterates over all training nodes and checks the connections between a node and the nodes in previous iterations. Based on the connections, PaGraph assigns the node to a partition so that the best load balance is achieved. Once it gets a partitioning of the training nodes, it assigns all other nodes in L-hop neighbor of training nodes to each partition. PaGraph does not use the interconnection among GPUs – each GPU loads data either from its local memory or from CPU.

Settings: Two GNN models are used in our evaluation. We train a GraphSAGE model [7] on `reddit`, `papers100M` and `MAG240M`, and a GCN model [16] on `yelp` and `products`, with layer-wise neighbor sampling [41]. All the GNNs have three convolutional layers. The number of sampled nodes in each layer is set to 8192. The dimension of the intermediate embeddings is set to 512. We use Adam SGD as the training algorithm and run it for 30 epochs. The batchsize is set to 512 for `reddit` and `products` and 2048 for other graphs. The learning rate is set to $0.01n$ where n is the number of GPUs used for training. For Algorithm 1, we set $\alpha = 0.2$ on '(2+2)GPU' and '(2*4)GPU' and $\alpha = 0$ on '2GPU', '4GPU' and '8GPU'. For locality-aware neighbor sampling, we set $lt = 0.15$, $ut = 0.2$ and $MAXS = 8$ in Algorithm 2, and set the size of neighbor set Q'_l to 8192 in all layers. We run the experiments for five times and report the average execution time of five runs.

5.2 Results on Small Graphs

Figure 4 shows training time per epoch on `reddit` graph with different partitioning methods and buffer sizes on 2 or 4 GPUs. We can see that loading the input features is a performance bottleneck

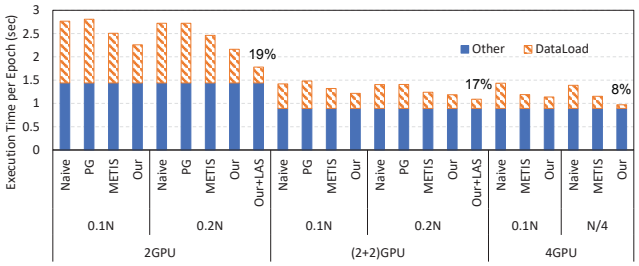


Figure 5. Breakdown execution time on `yelp` graph with feature buffer size 0.1N (82MB), 0.2N (164MB), and N/4 (205MB) on each GPU.

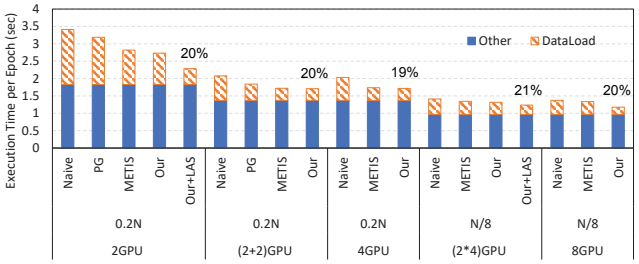
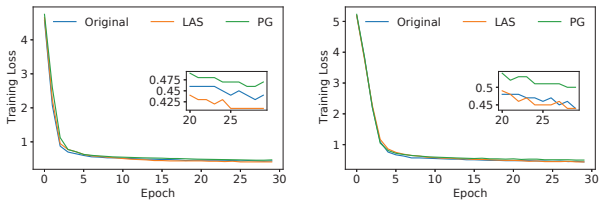


Figure 6. Breakdown execution time on `products` with feature buffer size N/8 (117MB) and 0.2N (187MB) on each GPU.

when the GPU buffer is small. With naive partitioning, data loading takes more than 50% of the total execution time if we use two GPUs and store 10% of the most frequently accessed nodes on each GPU (i.e., $BSIZE = 0.1N$). When the GPU buffer size increases to $0.2N$, the data loading time slightly decreases to 48% of the total execution time. The ratios are similar on '(2+2)GPU'. PaGraph has almost the same performance as naive partitioning. Because PaGraph does not utilize the interconnects among GPUs, its performance on '4GPU' is the same as on '(2+2)GPU'. With METIS partitioning, the data loading time is slightly better but still takes about 40% of the total execution.

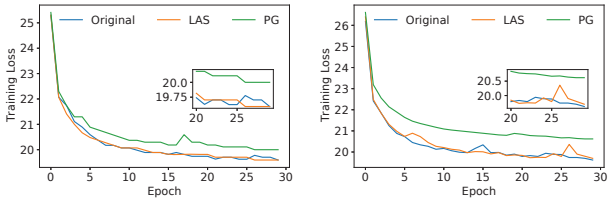
Our data placement strategy achieves smaller data loading time than other graph partitioning methods. On '2GPU' and '(2+2)GPU', because only two GPU buffers are used together, data loading still takes 30% of total execution time with our data placement strategy. After applying locality-aware sampling in these two cases, the data loading time is reduced to about 20% of total execution time (shown as Our+LAS in the figure). On '4GPU', because most (or all) of nodes are stored on GPU, our data placement alone is able to achieve good performance. Note that even all nodes are stored on GPUs in this case, our data placement strategy outperforms other partitioning methods, which indicates that it utilizes aggregate memory of multiple GPUs more efficiently.

Figure 5 and 6 show training time per epoch with different partitioning methods on `yelp` and `products`. The results are similar to `reddit`. Our data placement strategy achieves smaller data loading time than both PaGraph and METIS partitioning. We do not include the execution time of PaGraph on eight GPUs in Figure 6 because it cannot partition the `products` graph into eight parts within five hours. For cases where our data placement strategy alone cannot reduce the data loading time to 20% of total execution time, we



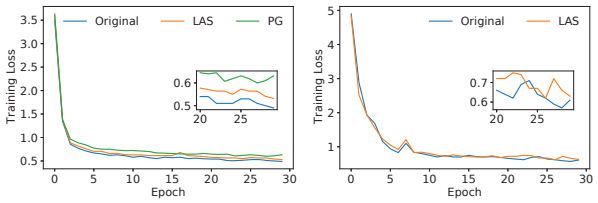
(a) 2GPU **(b)** (2+2)GPU

Figure 7. Training losses on `reddit` graph.



(a) 2GPU **(b)** (2+2)GPU

Figure 8. Training losses on `yelp` graph.



(a) `products` 2GPU **(b)** `products` (2*4)GPU

Figure 9. Training losses on `products` graph.

apply locality-aware sampling to further reduce the data loading time. The results are shown as Our+LAS in the figures.

The advantage of our techniques can be further verified by the CPU and GPU memory access sizes. For `reddit` graph on '2GPU' with $0.2N$ feature buffer, if METIS partitioning is used, each GPU needs to access 40% of nodes from local GPU memory, 27% from remote GPU memory, and 33% from CPU. Our data placement strategy with locality-aware sampling increases local accesses to 51% and reduces CPU accesses to 13%. On '4GPU' with $0.1N$ feature buffer, our data placement strategy increases local accesses by 72% and reduces CPU accesses by 51% compared with `Naive`, and increases local accesses by 13% and reduces CPU accesses by 12% compared with `METIS`. The same improvement is observed on `yelp` and `products` graph.

To show how locality-aware sampling affects the training process, we compare the training loss of different sampling methods for all the cases where locality-aware sampling is applied. Figure 7 shows the training loss over 30 epochs on `reddit` graph on '2GPU' and '2+2GPU'. Compared to the original sampling method, locality-aware sampling has almost the same convergence speed. The training loss is even smaller than the original sampling method

Table 2. Data preprocessing time in seconds with different graph partitioning methods.

	reddit	yelp	products
PaGraph	382	1976	4753
METIS	17	15	83
Our	0.49	0.76	3.6

at the end of training. While the result seems surprising, it is actually possible because the original sampling does not guarantee that the estimation variance for neighbor aggregation is always minimized [2, 41]. We also collect the training loss with PaGraph. PaGraph has slower convergence than our method because it only allows local shuffling of training nodes in order to achieve high hit rates on GPU buffers. The local shuffling violates the i.i.d. sampling assumption for the training algorithm and often leads to models with lower accuracy [25]. The results on `yelp` (Figure 8) and `products` (Figure 9) follow a similar pattern. The test accuracy of the models trained with locality-aware sampling is 0.964 (±0.001), 0.642 (±0.003), and 0.787 (±0.001) on `reddit`, `yelp`, and `products`, which match the accuracy of same models reported in previous work [32] and Open Graph Benchmark leaderboard.

Note that the above results are collected with $MAXS = 8$ for locality-aware sampling. The results indicate that locality-aware sampling is able to reduce the data loading time without affecting the convergence speed when $MAXS$ is small. However, when $MAXS$ is larger, the training will have slower convergence as skewed sampling leads to larger estimation variance for neighbor aggregation. In this sense, our locality-aware sampling illustrates a tradeoff between data loading time and convergence speed of training algorithm.

Besides the reduced data loading time, another advantage of our technique is that our algorithm is much faster than the previous graph partitioning algorithms. Table 2 lists the execution time of different graph partitioning methods for dividing the graphs into four parts. The performance of PaGraph is dependant on the number of partitions. It runs much longer for partitioning the graphs into eight parts. The execution time of METIS and our algorithm is not affected much by the number of partitions. The execution time of our algorithm include the execution time of Algorithm 1 and the time for computing an estimation of sampling probability p as described in §3.2. PaGraph partitioning is extremely slow because the algorithm has $O(N^2)$ time complexity. METIS has linear time complexity w.r.t. the number of nodes and number of edges, and thus it is much faster than PaGraph. Our algorithm is even faster. This is because we encode the edge information in the sampling probability p (which can be computed efficiently as sparse matrix-vector multiplication), and the partitioning procedure does no much more work than iterating over an array of N elements. Compared to the training time, the data preprocessing overhead with our algorithm is negligible.

5.3 Results on Large Graphs

We next run experiments on `papers100M` and `MAG240M`. PaGraph cannot finish partitioning either of two graphs in five hours. METIS partitioning aborts due to out-of-memory error. Our algorithm takes 24 seconds to return a data placement strategy for `paper100M`

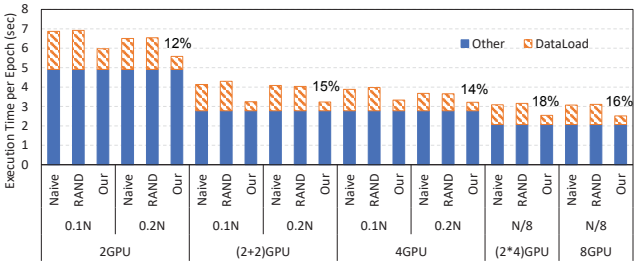


Figure 10. Breakdown execution time on `papers100M` with feature buffer size 0.1N (5.3GB), N/8 (6.6GB), and 0.2N (10.6GB) on each GPU.

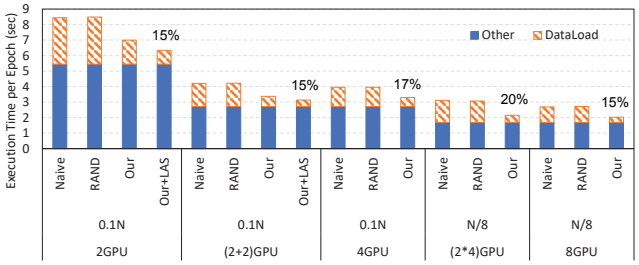
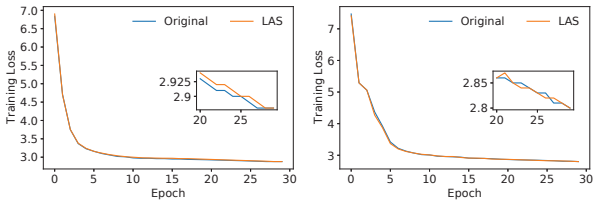


Figure 11. Breakdown execution time on `MAG240M` with feature buffer size 0.1N (17.4GB) and N/8 (21.8GB) on each GPU.



(a) `mag240m 2GPU` **(b)** `mag240m (2+2)GPU`

Figure 12. Training losses on `MAG240M` graph.

and 27 seconds for `MAG240M`. To show the effectiveness of our techniques, we run naive and random partitioning and compare the data loading time.

Figure 10 shows the time per epoch on `papers100M` graph with 2~8 GPUs. The graph has 53GB of node features. We set the $BSIZE$ on each GPU to $0.1N$ and $0.2N$. Although the buffer size is smaller than the GPU memory size, we cannot allocate more memory for input features because the GNN model and the intermediate variables also take space. From the figure, we can see that naive partitioning and random partitioning have almost the same data loading time. Our data placement strategy consistently outperforms both random partitioning and naive partitioning, reducing the data loading time by 2.4x to 4.0x on different number of GPUs compared to random partitioning. Our technique is most effective on '4GPU' where the GPUs are all connected together. On '(2+2)GPU' and '(2*4)GPU', the ratio of data loading time to total execution time is slightly higher but still lower than 20%.

Figure 11 shows the performance results on MAG240M. The graph has 174GB of node features. On '4GPU' and '8GPU', our data placement strategy alone decreases the data loading time to less than 20% of total execution time. On '2GPU' and '(2+2)GPU', our data placement strategy is also effective, achieving less than half data loading time than random graph partitioning. However, because the buffer size is small (20% of nodes in total on two GPUs), the data placement strategy alone cannot decrease data loading time to less than 20% of total execution time. Thus, we apply locality-aware sampling in these cases. With locality-aware sampling, the ratios of data loading time to total execution time are reduced to 15%. Figure 12 shows the training loss of locality-aware sampling and the original sampling method in the two cases. The two lines almost overlap. The test accuracy of trained model is 0.688 (±0.002), which matches the accuracy of the same model in Open Graph Benchmark. The results again validate that our locality-aware sampling has little effect to convergence speed of training when *MAXS* is small.

6 Related Work

Many systems have been proposed for GNN training on GPUs. The most popular two are PyG [5] and DGL [27]. PyG is a collection of GNN models and their common components. It does not support large-scale GNN training by itself because its implementation assumes the data fit in a single GPU. DGL aims to provide a uniform abstraction for building GNN models. It assumes that the entire graph can be stored on multiple GPUs. To minimize communication overhead among GPUs, DGL adopts METIS graph partitioning [15]. As shown in our experiments, our data placement strategy not only achieves faster data loading than METIS but also has a smaller data preprocessing overhead.

NeuGraph [21] is a system for GNN training with full neighbor aggregation on multiple GPUs. It uses METIS for graph partitioning. To support large graphs, it stores the graph on CPU and streams edge blocks with their associated node features onto GPUs. However, copying data from CPU to GPU incurs a large overhead. ROC [14] is another system for GNN training with full neighbor aggregation on multiple GPUs. It assumes that all the input features can be stored on GPUs and does not consider data loading problem for large-scale GNN training. It focuses on improving load balance among GPUs by partitioning the graph dynamically. Different from these systems, our work targets data-parallel GNN training with sampled neighbor aggregation, which is more commonly used for large-scale GNN training [9]. In this setting, the computation time is determined by the size of sampled subgraphs. Since the sampling configuration is the same on different GPUs, load imbalance is less of an issue. Graph partitioning is mainly designed for balancing data movements among GPUs.

PaGraph [1, 20] aims to reduce the data loading time for GNN training on large graphs that cannot be entirely stored on GPU. The main idea is to buffer the most frequently accessed nodes on GPU. For training on multiple GPUs, PaGraph uses a partition algorithm of quadratic time complexity. The partitioning procedure is so expensive that the data preprocessing time can be much longer than the GNN training time itself. Also, to achieve high hit rates on the GPU buffers, PaGraph only allows locally shuffling of training nodes, which slows down the convergence of training process and may lead to models with lower accuracy. Min *et al.* [22] also identify the large data loading overhead issue in GNN training and propose a GPU-oriented data communication technique to reduce the data loading overhead. The idea is to allow the GPU threads to directly access sparse features in CPU memory through zero-copy accesses so that expense data gathering can saved. They do not consider data partitioning and use DGL for evaluation. Our work is orthogonal to and can be combined with their GPU-oriented data communication technique.

Graph partitioning is also used for distributed GNN training on CPUs [8, 26, 33, 39]. AliGraph [39] implements various graph partitioning methods for different types of graphs. AGL [33] proposes an edge-partitioning algorithm. Dorylus [26] partitions the input graph with an edge-cut algorithm [40]. These systems perform full-batch or large-batch neighbor aggregation and distribute the aggregation operation on multiple machines. Graph partitioning is used to reduce communication and improve load balance for the distributed neighbor aggregation operation. Therefore, the problem objective is different from the objective of our work.

There are also many works on accelerating GNN computation on GPUs [12, 13, 28]. These works mainly focus on improving the data locality and load balance for the graph convolution operation. FeatGraph [11] combines graph partitioning with feature dimension tiling to accelerate computations in GNNs. C-SAW [24] focuses on accelerating graph sampling on GPUs; it partitions graph edges for out-of-memory sampling. These works do not consider the overhead for loading input features.

Data movement problem has also been studied for graph embedding systems [17, 23, 38]. These systems also use graph partitioning to handle large graphs with limited memory capacity. For example, DGL-KE [38] adopts METIS graph partitioning, which has been shown inefficient for GNN training in our experiments. Because graph embedding models have different computation and memory access patterns from GNNs, these systems cannot be used for GNN training and cannot be directly compared with our work.

7 Conclusion

In this work, we aim to reduce the data loading overhead for large-scale GNN training on multiple GPUs. We propose a performance model of the data movement among CPU and GPUs in GNN training, and based on the performance model, we provide an efficient algorithm to find an optimal data placement strategy. We also propose a locality-aware neighbor sampling technique to further reduce the data loading overhead without affecting the accuracy. The experiments show that our techniques outperform the existing graph partitioning methods in terms of both data loading efficiency and preprocessing overhead.

Acknowledgements

This work was supported by NSF award CCF-2028825.

References

[1] Youhui Bai, Cheng Li, Zhiqi Lin, Yufei Wu, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2021. Efficient Data Loader for Fast Sampling-Based GNN Training on Large Graphs. *IEEE Transactions on Parallel and Distributed Systems* 32, 10 (2021), 2541–2556.

[2] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations*.

[3] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An efficient algorithm for training deep and large graph

convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 257–266.

[4] Alberto Garcia Duran and Mathias Niepert. 2017. Learning graph representations with embedding propagation. In *Advances in neural information processing systems*. 5119–5130.

[5] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[6] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *International Conference on Machine Learning*. 1263–1272.

[7] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in neural information processing systems*. 1024–1034.

[8] Loc Hoang, Xuhao Chen, Hochan Lee, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. [n.d.]. EFFICIENT DISTRIBUTION FOR DEEP LEARNING ON LARGE GRAPHS. *update* 1050 ([n. d.]), 1.

[9] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. Ogb-lsc: A large-scale challenge for machine learning on graphs. *arXiv preprint arXiv:2103.09430* (2021).

[10] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687* (2020).

[11] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. Featgraph: A flexible and efficient backend for graph neural network systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[12] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*.

[13] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and bridging the gaps in current GNN performance optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 119–132.

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

[15] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[16] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations (ICLR)*.

[17] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. Pytorch-biggraph: A large scale graph embedding system. *Proceedings of Machine Learning and Systems* 1 (2019), 120–131.

[18] Ruoyu Li, Sheng Wang, Feiyun Zhu, and Junzhou Huang. 2018. Adaptive Graph Convolutional Neural Networks. In *AAAI Conference on Artificial Intelligence*.

[19] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. 2018. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324* (2018).

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

[21] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. Neugraph: parallel deep neural network computation on large graphs. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 443–458.

[22] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021. Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2087–2100. https://doi.org/10.14778/3476249.3476264

[23] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. 2021. Marius: Learning Massive Graph Embeddings on a Single Machine. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 533–549.

[24] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. 2020. C-SAW: A framework for graph sampling and random walk on GPUs. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[25] Benjamin Recht and Christopher Ré. 2012. Toward a noncommutative arithmetic-geometric mean inequality: conjectures, case-studies, and consequences. In *Conference on Learning Theory*. JMLR Workshop and Conference Proceedings, 11–1.

[26] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. 2021. Dorylus: affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 495–514.

[27] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315* (2019).

[28] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for {GNN} Acceleration on GPUs. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 515–531.

[29] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*.

[30] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 974–983.

[31] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. 2018. Hierarchical graph representation learning with differentiable pooling. In *Advances in neural information processing systems*. 4800–4810.

[32] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *International Conference on Learning Representations*.

[33] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. 2020. AGL: A Scalable System for Industrial-Purpose Graph Machine Learning. In *VLDB Endowment*. 3125–3137.

[34] Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. 2018. Gaan: Gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294* (2018).

[35] Muhan Zhang and Yixin Chen. 2017. Weisfeiler-lehman neural machine for link prediction. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 575–583.

[36] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. In *Advances in Neural Information Processing Systems*. 5165–5175.

[37] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 36–44.

[38] Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis. 2020. Dgl-ke: Training knowledge graph embeddings at scale. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 739–748.

[39] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *Proc. VLDB Endow.* (2019), 2094–2105.

[40] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 301–316.

[41] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. 2019. Layer-Dependent Importance Sampling for Training Deep and Large Graph Convolutional Networks. In *Advances in Neural Information Processing Systems*. 11249–11259.