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

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

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

Abstract

The existing Graph Neural Network (GNN) systems adopt graph partitioning to divide the graph data for multi-GPU training. Although they support large graphs, we find that the existing techniques lead to large data loading overhead. In this work, we for the first time model the data movement overhead among CPU and GPUs in GNN training. 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. The experiments show that our technique achieves smaller data loading time compared with the existing graph partitioning methods.

CCS Concepts: • **Software and its engineering → Massively parallel systems**.

Keywords: graph neural network, data loading

1 Introduction

Graph Neural Networks (GNNs) have emerged as the state-of-the-art models for machine learning tasks on graphs. Due to their superior accuracy, GNNs play an important and increasing role in many application domains, including content recommendation, traffic prediction, and molecular property prediction. Different from the traditional graph processing algorithms, GNNs make predictions on graphs with node features. 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. The node features exceed the memory capacity of most GPUs, making it challenging to train GNNs efficiently on large graphs.

To handle large graphs, the current GNN systems either take an off-the-shelf graph partitioning algorithm or use heuristic partitioning methods. For example, DGL [3] uses METIS [1] for graph partitioning. It stores the graph on GPU to avoid the data copying from CPU to GPU. However, it does not work for large graphs that exceed the memory capacity of multiple GPUs. PaGraph [2] partitions the graph based on the training nodes. It stores the graph on CPU and buffers the most frequently accessed nodes of each partition on GPU. In order to achieve high hit rates on GPU buffers, PaGraph only allows local shuffling of training nodes, which often leads to models with lower accuracy.

We find that the existing graph partitioning methods are unsatisfactory for GNN training. Figure 1 shows the training time per epoch on `reddit` graph with different partitioning methods and buffer sizes. We can see that loading the input features is a performance bottleneck when the GPU buffer is small. With PaGraph (PG), data loading takes 50% of the total execution time if we use two GPUs and store 10% of the most frequently accessed nodes on each GPU. When the GPU buffer size increases to 20% of the nodes, the data loading time slightly decreases to 47% of the total execution time. With DGL, the data loading time is slightly better but still takes about 40% of the total execution.

To improve the performance of the existing systems, we study the data movements among CPU and GPUs for GNN training and propose an efficient algorithm for dividing the graph data onto multiple GPUs.

2 Minimizing Data Movement For Input Features

We consider the data movement problem in data-parallel GNN training with n GPUs on a single machine. We assume that each GPU-i can only store the feature vectors of a set of nodes B_i. With this setup, we now give a performance model of the data movement of input features and provide an efficient algorithm to minimize the data movement overhead.

2.1 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 nodes on the same GPU is zero, the cost of reading nodes 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

$$R_i = S_i \setminus B_i = S_i - (S_i \cap B_i). \quad (1)$$

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9204-4/22/04.
https://doi.org/10.1145/3503221.3508435

For every node in R_i, we need to decide where to fetch it. 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 $R_{i,gpu} \in B_{gpu}$ are from other GPUs. The cost of reading S_i can 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

$$C_i(B) = \begin{cases} C_{cpu}|R_i \setminus B_{gpu}| + C_{gpu}|R_i \cap B_{gpu}|, & \text{if } C_{gpu} < C_{cpu}; \\ C_{cpu}|R_i|, & \text{if } C_{gpu} \geq C_{cpu}. \end{cases} \tag{2}$$

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:

$$\begin{aligned} \min \quad & \max_{i \in [1,n]} \left(\mathbb{E}_{S_i \sim \mathcal{D}}[C_i(B)]\right), \\ \text{subject to} \quad & |B_i| \leq BSIZE, \quad i = 1, \ldots, n \end{aligned} \tag{3}$$

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 minimizes $\max_i(\mathbb{E}_{S_i}[C_i])$.

2.2 An Efficient Solution

To solve Problem (3), 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 (1) and (2), for each GPU we need to minimize

$$\max_i(\mathbb{E}_{S_i}[C_i]) = C_{cpu}\mathbb{E}_S[|S|] - C_{cpu} \min_i \mathbb{E}_{S_i}[|S_i \cap B_i|])$$

where $\mathbb{E}_S[|S|]$ is the expected number of sampled input nodes and is a constant numbe for all GPUs. Since nodes with higher sampling probabilities are more likely shown in S_i, when nodes with the highest sampling probabilities are stored on each GPU, $\mathbb{E}_{S_i}[|S_i \cap B_i|]$ is maximized and the data loading overhead is minimized.

When $C_{gpu} < C_{cpu}$, i.e., the GPU is connected with other GPUs through NVLink. For each GPU we need to minimize

$$\begin{aligned} \max_i \left(\mathbb{E}_{S_i}[C_i]\right) = & C_{cpu}\mathbb{E}_S[|S|] - \min_i \left(C_{gpu}\mathbb{E}_{S_i}[|S_i \cap B_i|]\right) \\ & - \left(C_{cpu} - C_{gpu}\right) \mathbb{E}_S\left[|S \cap B_{gpu}|\right]. \end{aligned} \tag{4}$$

The last two terms reveal a tradeoff. The second last term suggests that each GPU stores the same set of nodes with the highest sampling probabilities, while the last term suggests we should store as many nodes as possible on all GPUs. To explore the tradeoff, we first obtain an ordered set of the nodes and store nodes from $V[0]$ to $V[BSIZE-1]$ with the highest sampling probability on each GPU. The algorithm tries to iteratively replace the duplicate nodes on different GPUs with new nodes from $V[BSIZE]$ to $V[N-1]$ and keep at least one copy of the node on GPU. In the first iteration, it tries to replace $V[BSIZE-1]$ in B_1 with $V[BSIZE]$. If the increase of the second last term of (4) is greater than the decrease of the last term, i.e., $C_{cpu}p_{(V[BSIZE])} > C_{gpu}p_{(V[BSIZE-1])}$, the replacement is beneficial, and we perform the replacement. To ensure GPUs have similar data movement costs, we select the GPU with the lowest sum of sampling probabilities in every replacement step. An advantage of our algorithm over the existing graph partitioning methods is that it has $O(N)$ time and space complexity.

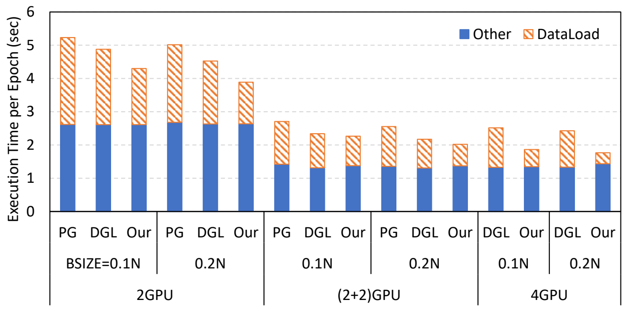

Figure 1. Breakdown execution time on `reddit` graph.

3 Evaluation

We evaluate our technique on four Nvidia V100 GPUs with two interconnect configurations: 1) every two GPUs are connected through NVLink Bridge (denoted as '(2+2)GPU'); 2) all GPUs are connected through NVSwitch (denoted as '4GPU'). By only allowing a GPU to read data from the GPU next to it, we simulate systems with NVLink Bridges.

As shown in Figure 1, loading the input features is a performance bottleneck when the GPU buffer is small. Our data placement strategy achieves smaller data loading time than both PaGraph and DGL. We can see that our technique is most effective on '4GPU', reducing the data loading time by 2.3x compared to DGL when $BSIZE = 0.1N$ and 3.3x when $BSIZE = 0.2N$. On '2GPU' and '(2+2)GPU', because only two GPU buffers are used together, the improvement is not as good as on '4GPU', but we still achieve 1.5x to 1.9x speedup against PaGraph and 1.2x to 1.5x speedup against DGL.

References

[1] 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.

[2] 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.

[3] 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.