ADAPTIVE LOAD BALANCING FOR PARALLEL GNN TRAINING

Qidong Su[1] Minjie Wang[1] Da Zheng[2] Zheng Zhang[1]

ABSTRACT

The recent emergence of demand for running Graph Neural Networks (GNNs) on giant real world graphs requires more scalable system designs. Due to the sparse and irregular connections a graph has, parallel GNN training encounters the problem of load imbalance among workers. In this paper, we show that previous techniques based on graph partitioning is insufficient to address the load imbalance caused by GNN sampling algorithms. We thus propose a two-stage strategy to balance the workload adaptively during training. Our evaluation shows that the strategy effectively produces more balanced workloads which accelerates the training by 25%.

1 INTRODUCTION

Learning from relational data such as graphs plays a central role in many real world scenarios. The recent trend of geometric deep learning (Bronstein et al., 2017) gives rise to Graph Neural Networks (GNNs). GNNs combine the strength of deep neural networks and the message passing algorithms on graphs. This grants GNN the capability to learn from both features and graph topology, which are key to its success in fields like community detection (Kipf & Welling, 2017; Veličković et al., 2018), recommender system (Ying et al., 2018; Berg et al., 2017; Wang et al., 2019b), molecule property prediction (Shui & Karypis, 2020; Xiong et al., 2019), and so on.

As real world graphs can be gigantic, i.e., consisting of billions of nodes or edges, it is critical to support training GNNs at scale. Much resembling the stochastic gradient descent algorithm for CNNs or RNNs, the stochastic training algorithm of GNNs first extracts a surrounding subgraph (e.g., an ego-network) for the target nodes to compute representations for, then trains the GNN model on the subgraph, and repeats the steps until convergence. Previous studies (Hamilton et al., 2018; Chen et al., 2018; 2017) have shown that the GNNs trained with sampling yield competitive results.

To parallelize the training procedure, current GNN systems (Zheng et al., 2020; Alibaba, 2019; Yang, 2019) adopt a *synchronous data parallelism* approach; each worker performs sampling and training in parallel and synchronizes model parameters before the next iteration. However, due to the sparse and irregular connections a graph has, the sample sizes among workers can vary drastically, causing severe workload imbalance. Existing systems such as DistDGL (Zheng et al., 2020), AliGraph (Yang, 2019) and PaGraph (Lin et al., 2020) leverage off-the-shelf graph partitioning algorithms to address the problem. The issue is that those graph partitioning algorithms are designed for graph analytical applications where workload distribution can be well modeled by the partition size. By contrast, GNN workload is more complicated and the choice of sampling algorithms can have noticeable impact on the workload distribution.

In this paper, we propose a novel load balancing strategy designed for parallel training of GNNs. Our contributions are as follows:

- We conduct analytical studies to reveal the key deciding factors of a GNN workload and further show that previous techniques fail to balance these factors.
- We propose a two-stage load balancing strategy, which first distributes the workload by graph partitioning and then adjusts the workload dynamically during training. The strategy is general and applicable to a wide range of sampling algorithms.
- We implement our proposed approach into one of the state-of-the-art GNN frameworks – Deep Graph Library (DGL) (Wang et al., 2019a) and test its advantage over previous strategies.

2 BACKGROUND AND MOTIVATION

2.1 GNN Mini-batch Training and Neighbor Sampling

At its core, GNNs aim at learning from both topology and node/edge attributes from graphs. This is achieved via the

[1] AWS Shanghai AI Lab [2] AWS AI. Correspondence to: Minjie Wang <minjiw@amazon.com>.

Proceedings of the 4^{th} MLSys Conference, San Jose, CA, USA, 2021. Copyright 2021 by the author(s).

Algorithm 1 Neighbor Sampling

```
input $G$, $V_{target}$, fan-out $\langle f_1, f_2, \ldots, f_L \rangle$, node-wise sampler $q$
  $V_L \leftarrow V_{target}$
  for $i = L$ to $1$ do
    $SG_i \leftarrow \phi, V_{i-1} \leftarrow \phi$
    for $v$ in $V_i$ do
      Draw $f_i$ samples $\langle u_1, \ldots, u_{f_i} \rangle$ using sampler $q_v$.
      $SG_i \leftarrow \{(u_1, v), \ldots, (u_{f_i}, v)\} \cup SG_i$
      $V_{i-1} \leftarrow \{u_1, \ldots, u_{f_i}\} \cup V_{i-1}$
    end for
  end for
  return $\langle SG_L, \ldots, SG_1 \rangle$
```

Algorithm 2 Parallel Mini-batch Training for GNNs

```
  Each trainer gets a subset of the full training set.
  loop
    $seeds \leftarrow$ next batch of training nodes.
    $subg \leftarrow NeighborSampling(seeds, fanout)$
    $feats \leftarrow$ feature data of input nodes in $subg$.
    Run forward and backward computation.
    Synchronize gradients among trainers.
    Update parameters.
  end loop
```

message passing mechanism (Gilmer et al., 2017); at each GNN layer, nodes collect *messages* from their neighbors, aggregate and combine them with the nodes' current representations to generate ones for the next layer. Define *target nodes* to be the nodes to compute representation for and *receptive field* to be the nodes which target nodes to gather messages from. Then the size of receptive field grows exponentially with more layers in a GNN model. The induced cost of communication and computation essentially prohibits training deep GNN models on large graphs.

To mitigate the problem, one common strategy is to collect messages from a sampled neighborhood (Algorithm 1) Figure 1 illustrates one sampled subgraph starting from target nodes A and B. The choice of sampling algorithms is an active research, with proposals such as importance-based sampling (Huang et al., 2018), type-aware sampling (Hu et al., 2020), bandit sampler (Liu et al., 2020), etc. The training procedure then resembles the iterative process of stochastic gradient descent; at each iteration, the GNN model performs forward propagation using the sampled subgraph and the extracted node/edge features, computes gradients by backward propagation and updates the parameters.

2.2 Parallel GNN Training and Workload Balancing

The mini-batch training algorithm mentioned above can be easily parallelized in a synchronous data parallelism paradigm, as implemented in some distributed GNN systems (Zheng et al., 2020; Alibaba, 2019; Yang, 2019). As shown in Algorithm 2, the training set is first distributed to the trainers. In each iteration, each trainer samples a subgraph from the full graph, fetches the required feature data and copies them to computing device such as GPU. After forward and backward computation, all trainers need to synchronize gradients with each other before updating the parameters. This global synchronization requires all trainers to have balanced workload.

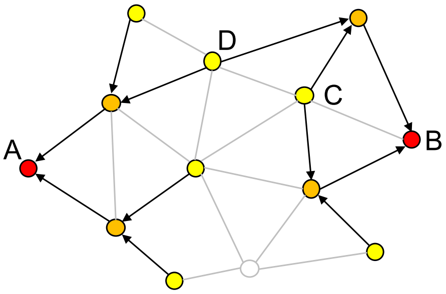


Figure 1. A mini-batch generated by a 2-layer neighbor sampling from two target nodes A and B. Orange nodes are the one-hop neighbors and yellow nodes are two-hop neighbors. Node C is shared by two neighbors of node B while Node D is covered by the receptive fields of both A and B.

To speed up the training process, systems like PaGraph (Lin et al., 2020) and DistDGL (Zheng et al., 2020) adopt non-trivial workload distribution algorithms. PaGraph proposes a graph partitioning algorithm to improve the cache efficiency and reduce cross-partition visits. DistDGL uses the METIS algorithm (Karypis & Kumar, 1998a;b) to generate hierarchical graph partition scheme to minimize the cut edges set across partitions, which is friendly for multi-machine-multi-GPU training. As illustrated in Figure 2, DistDGL first partitions a graph to multiple machines. Each machine can have multiple trainers, each owning one GPU. Trainers on the same machine can access the local partition data via shared memory, or issue network requests to remote machines. DistDGL also splits training sets hierarchically and assigns them based on locality constraints.

However, the approach based on graph partitioning algorithms are not sufficient to balance the GNN workload among trainers. To highlight it, we trained a GraphSAGE model using DistDGL on four trainers and plotted the iteration time of each trainer in Figure 3. It clearly shows a severe workload imbalance as trainer 3 is 2x faster than the others. This motivates our work to first understand the root cause behind and further proposes new load balancing techniques.

Other notable related work include NeuGraph (Ma et al., 2019), which is aimed at parallel full graph training. Tra-

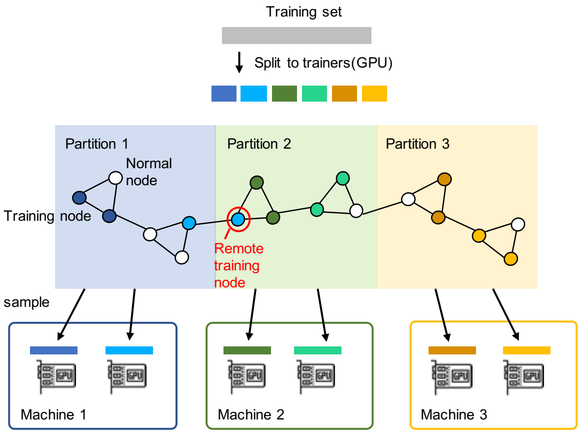


Figure 2. Graph partitioning and training set assignment in DistDGL.

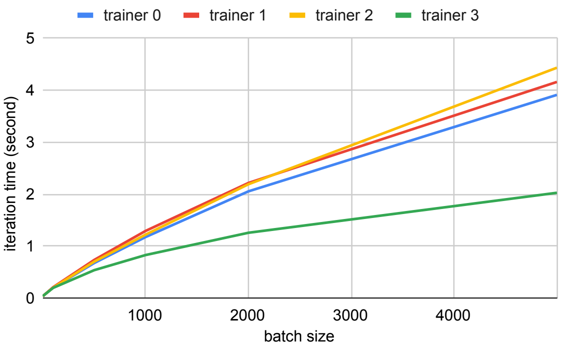


Figure 3. The iteration time when training a GraphSAGE model with four trainers using DistDGL. The running time varies across trainers indicating workload imbalance. Larger batch size increases running time but not linearly.

ditional graph analytic systems also pay attention to load balancing. Gemini (Zhu et al., 2016) adopts locality-aware chunking and fine-grained work-stealing for improving both inter-node and intra-node load balance. Mizan (Khayyat et al., 2013) monitors the runtime characteristics of the system and performs vertex migration to balance workload. Gunrock (Wang et al., 2016) proposes load balancing strategies for efficient graph algorithms in CUDA.

3 Understand Load Imbalance in Parallel GNN Training

In this section, we breakdown the parallel GNN training algorithm (Alg. 2) and analyze their contributions to balancing workload.

- *Graph sampling*: Despite the fact that all trainers share the same sampling configurations (e.g., batch size, fanouts, etc.), the sampling workload can still vary across trainers. As shown in Alg. 1, the complexity is affected by not only by the size of V_{target} (i.e., batch size) but also the size of each node frontier V_i. The choice of nodes in V_i has a large impact on the size of V_{i-1} even under the same algorithmic setting as shown in Figure 5. Another contributing factor is how many nodes in each V_i are remote nodes, as they will add to the cost of issuing network requests to sample from other machines.

- *GNN computation*: There are two parts of computation workload: (1) the forward and backward propagations of the GNN model on the sampled subgraphs, and (2) updating model parameters by the synchronized gradients. Because the GNN model is replicated across all trainers, the workload size is thus decided by the structure of the sampled subgraphs on each trainer.

- *Data communication*: Data communication happens at two locations. First, before calculating the forward and backward propagations, each worker needs to fetch the input feature data of the receptive fields (either from local storage or via network). Second, at the end each iteration, all trainers need to send out the local gradients for synchronization. Similarly, due to parameter sharing, the gradient synchronization workload is evenly split among trainers. For the cost to fetch input features, there are further two deciding factors: the *input feature size* and the *input feature locality* which determines the amount of data to fetch via network communication.

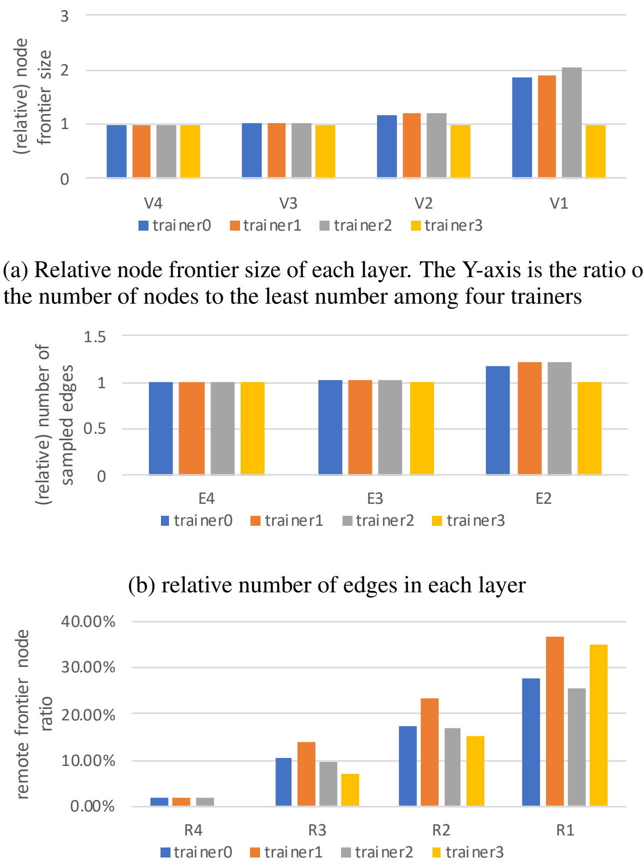
The general principle throughout the analysis is to ignore factors that are model dependent such as parameter synchronization, the choices of sampling configurations and the choices of GNN model configurations (e.g., feature, hidden dimension) because the models are replicated across trainers. While for the influence from the sampled subgraphs, we can measure it by the following metrics:

- *Node frontier sizes* $|V_L|, \ldots, |V_1|$, which affects graph sampling and GNN computation. $|V_1|$, in particular, determines the amount of feature data to fetch at each training iteration.

- *Number of sampled edges* $|E_L|, \ldots, |E_2|$, which affects GNN computation. In many cases, $|E_i| = |V_i| * f_i$.

- *Remote frontier node ratio* $R_L, \ldots, R_1$, which affects graph sampling and input feature fetching.

To evaluate how existing systems balance these factors, we train a 3-layer GraphSAGE model on the OGBN-PAPER100M dataset (Hu et al., 2021) (111,059,956 nodes,



(a) Relative node frontier size of each layer. The Y-axis is the ratio of the number of nodes to the least number among four trainers

(b) relative number of edges in each layer

(c) remote frontier node ratio of each layer

Figure 4. The current graph partition algorithm fails to balance the sample size

1,615,685,872 edges) using DistDGL (Zheng et al., 2020) in a cluster of 4 trainers. The batch size is 5000 for each trainer and the sampling fan-outs are 15, 10, 5 for each layer. Figure 4(a) and (b) compares the node frontier sizes and the number of sampled edges of each trainer normalized by the results of trainer 3. Figure 4(c) plots the remote frontier node ratio of each trainer. These metrics are averaged across 600 batches (10 epochs) and we find that the variance is negligible (<1%). The results indicate a severe imbalance among different trainers. For example, trainer 3 has almost 50% fewer nodes in the input frontier compared with other trainers. The ratio of remote nodes in each frontier also varies among trainers. All these factors contribute to the imbalanced running time observed in Figure 3.

4 Proposed Method

The fundamental reason why current graph partitioning algorithms fail to balance workloads is that they are not particularly designed for parallel GNN training. Many graph partitioning algorithms aim at minimizing the number of cross-partition edges or the number of replicated nodes while balancing the number of nodes and edges in each partition. However, we have shown that the workload balance of parallel GNN training is characterized by a different set of metrics and thus demands different solutions.

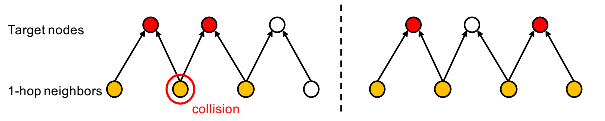


Figure 5. Even with the same batch size and sampling setting, different target node sets can lead to different node frontier sizes. In the left examples, the two target nodes share a common neighbor so it generates a node frontier size of three, while in the right example, there are four nodes in the next frontier.

Moreover, it is worth emphasizing that the sampling algorithm has a remarkable impact on the workload balance. Since users may search for suitable sampling algorithms during development or deployment, the load balancing strategy must adapt to them too. However, dynamically adjusting graph partitions or re-partitioning can be highly time-consuming, especially for giant graphs.

We thus propose to tackle the challenge of load balance in two folds. Before training, we employ the graph partitioning algorithm in DistDGL to generate static graph partition scheme that minimizes the cut edge set and meanwhile roughly balances the number of nodes and edges among partitions. During training, we keep track of the training time of all trainers and adjust their workload accordingly. We utilize the fact that a training process typically involves thousands or hundreds of thousands of mini-batches, so runtime adjustment in the first few mini-batches can benefit a longer run. In the following sections, we explain how we adjust per-trainer workload by batch size and how we further consider data locality in workload assignment.

4.1 Dynamic Adjustment of Per-trainer Workload

Our basic idea is to increase the workload of trainers that finish one iteration faster while do the opposite for those that are slower. There are a number of ways to achieve this goal, but there are several criterion. First, the solution shall be *universal*, i.e., applicable to all kinds of neighbor sampling algorithms. Second, it shall be *light-weighted* and incurs as little runtime overhead as possible. Third, it shall not change the training results like model accuracy.

We choose to adjust the batch size of each trainer according to their training time of one mini-batch. The insight is that although there are multiple influencing factors to the workload size of each mini-batch, the running time still has a positive correlation with the batch size (Figure 3). In

Algorithm 3 Training with adaptive batch size tuning

```
input total batch size $B$, number of trainers $N$, rank of the current trainer $i$, tuning period $P$
  $\mathbf{W} \leftarrow [1, 1, \ldots, 1]$, $B_i \leftarrow \frac{B}{N}$
  loop
    Perform $P$ training iterations and record iteration time $T_i$
    Synchronize across trainers and get their recorded time
    $\mathbf{T} \leftarrow [T_1^{-1}, T_2^{-1}, \ldots, T_N^{-1}]$
    $\mathbf{W} \leftarrow \mathbf{W} \odot \mathbf{T}$
    $\mathbf{W} \leftarrow \frac{1}{\sum_{j=1}^{N} W_j} \cdot \mathbf{W}$
    $B_i \leftarrow W_i \cdot B$
    Re-assign training nodes.
  end loop
```

addition, the approach is simple and needs no knowledge of the neighbor sampling algorithm in use. We also keep the total batch size unchanged to avoid impact on model accuracy.

More formally, given the running time to train one mini-batch of N trainers $T_1, T_2, \ldots, T_N$ and a total batch size B, we set the batch size of the i-th trainer as

$$B_i = \frac{T_i^{-1}}{\sum_{j=1}^{N} T_j^{-1}} \cdot B \quad (1)$$

Because the relation between running time and batch size is not linear, we propose an iterative algorithm to repeat the profiling-then-tuning process multiple times until all trainers finishes one iteration similarly fast as shown in Algorithm 3.

4.2 Locality-aware Training Set Assignment

After adjusting the batch size of each trainer, we then need to decide the assignments of training node set. The assignment happens at two levels: 1) splitting the training set to each machine and 2) further splitting the training set to the trainers on the same machine when each machine has multiple GPUs. For the first level, we follow the range partitioning practice from DistDGL which is friendly for looking up which partition a node belongs to, but adjust the range each machine owns based on the ratio of the total batch size of their trainers. For the second level, we further take data locality into consideration. Because training nodes that belong to partitions on other machines require network requests to perform neighbor sampling, it is important to balance them among trainers. To address this, we separate the local and remote training nodes, split them according to the batch size of each trainer respectively, and then concatenate local and remote nodes assigned to the same GPU (Figure 6).

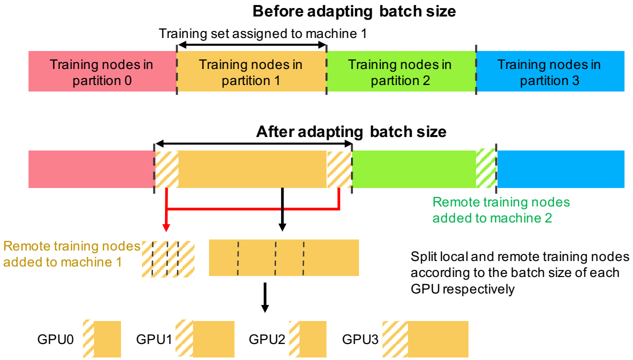

Before adapting batch size
Training set assigned to machine 1
Training nodes in partition 0
Training nodes in partition 1
Training nodes in partition 2
Training nodes in partition 3
After adapting batch size
Remote training nodes added to machine 2
Remote training nodes added to machine 1
Split local and remote training nodes according to the batch size of each GPU respectively
GPU0
GPU1
GPU2
GPU3

Figure 6. The training set assignment algorithm that takes locality into consideration. Here, the cluster has four machines and each machine has four GPUs. Suppose the total batch size of machine 1 increases after workload adjustment, so it is in charge of some remote training nodes stored on machine 0 and machine 2. When further assigning the workload to per-GPU trainers, we split local and remote nodes respectively according to their batch sizes, keeping the proportion of local and remote nodes equal for these four GPUs.

5 Evaluation

In this section, we evaluate our adaptive load balancing method to show its effectiveness of balancing the workloads in the cluster and accelerating the training.

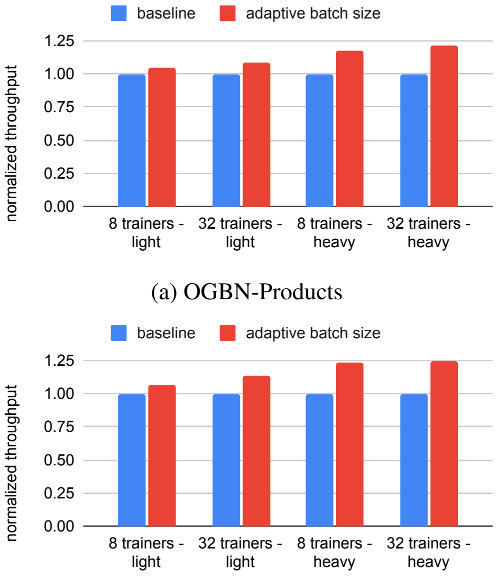
Environment Our experiments are performed on 4 AWS EC2 P3dn.24xlarge instances with 8 V100 GPUs on each machine. We use PyTorch 1.7 as the backend.

Dataset We use the two datasets from Open Graph Benchmark(Hu et al., 2021) shown in Table 1.

Hyper-parameter We run the experiments with two sets of hyper-parameters. One is from the GraphSAGE submission on the leader board of Open Graph Benchmark. It has three layers, of which the fan-out is 15, 10 and 5 respectively. The other one is heavier and has four layers, of which the fan-out is 10, 10, 5 and 5. The initial batch size is 2000 for each trainer, except for the heavy hyper-parameters on OGBN-Papers100M dataset, where the initial batch size is 5000 for each trainer.

We evaluate our method with different numbers of trainers. For each setting, we respectively run the experiments with 2 GPUs and 8 GPUs on each machine. In the latter scenario, the whole graph is partitioned to 32 parts in total. We adapt the batch size and re-split the training set at the end of every 10 training iterations.

We measure the increase in throughput (the number of samples processed per second) when the adaptive load balancing is enabled. As shown in Figure 7, for the three-layer Graph-



(a) OGBN-Products

(b) OGBN-Papers100M

Figure 7. The performance gain of adaptive load balancing strategy with different number of trainers and workload on two OGB datasets.

SAGE model, the adaptive load balancing strategy can bring a performance gain of 6% to 13% . As the model depth increases and the receptive field extends, the performance gain can achieve 25%.

Figure 8 shows the average iteration time of each trainer in the training process of a 4-layer GraphSAGE model on OGBN-Products dataset with a cluster of 4 machines which contains 2 GPUs. Because we can not take apart the backward computation and parameter synchronization in PyTorch, these parts are not included in the timing. In other words, we only take the time of sampling, data copying and forward computation into consideration. As shown in the figure, our method successfully balances the iteration time among trainers.

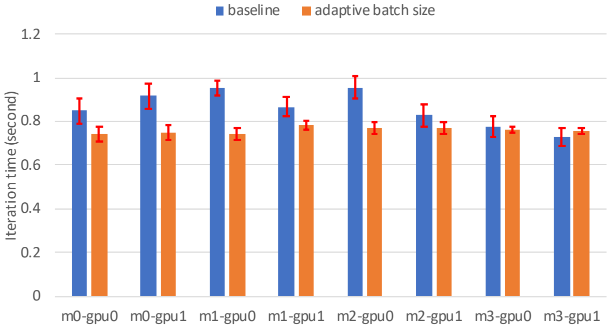


Figure 8. Iteration time (excluding backward and synchronization) of 8 trainers before and after batch size tuning

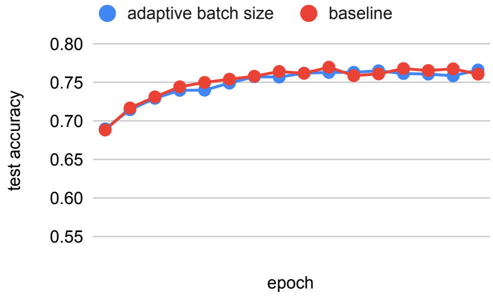


Figure 9. The test accuracy curve of training a GraphSAGE model on ogbn-products dataset before and after batch size tuning. The result shows they converge to the same accuracy.

In addition to the training speed, we also verify the training accuracy when the adaptive load balancing strategy is enabled. We run a three-layer GraphSAGE model on the OGBN-Products dataset using a cluster of 4 machines with 2 GPUs on each machine. Experiments show the accuracy converges to almost the same peak value achieved by the baseline.

6 Conclusion

We propose a novel adaptive load balancing strategy for parallel training of GNNs. We conduct an empirical and analytical study of the load imbalance problem of the existing GNN systems. Based on the observation of the monotonic relation between batch size and running time, we propose a novel load balancing strategy, which adapts to the different sampling algorithms dynamically. We implement it into one of the state-of-the-art GNN frameworks Deep Graph Library(Wang et al., 2019a) and test its advantage over previous strategies. Our evaluation shows that the strategy effectively produces more balanced workloads which accelerates the training by 25%.

References

Alibaba. Euler. https://github.com/alibaba/euler, 2019.

Berg, R. v. d., Kipf, T. N., and Welling, M. Graph convolutional matrix completion. *arXiv preprint arXiv:1706.02263*, 2017.

Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., and Vandergheynst, P. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4): 18–42, 2017.

Chen, J., Zhu, J., and Song, L. Stochastic training of graph

Table 1. Dataset statistics from the Open Graph Benchmark (Hu et al., 2021).

Dataset	# Nodes	# Edges	Node Features
OGBN-PRODUCT	2,449,029	61,859,140	100
OGBN-PAPERS100M	111,059,956	3,231,371,744	128

convolutional networks with variance reduction. *arXiv preprint arXiv:1710.10568*, 2017.

Chen, J., Ma, T., and Xiao, C. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, 2017.

Hamilton, W. L., Ying, R., and Leskovec, J. Inductive representation learning on large graphs, 2018.

Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. Open graph benchmark: Datasets for machine learning on graphs, 2021.

Hu, Z., Dong, Y., Wang, K., and Sun, Y. Heterogeneous graph transformer. In *Proceedings of The Web Conference 2020*, pp. 2704–2710, 2020.

Huang, W., Zhang, T., Rong, Y., and Huang, J. Adaptive sampling towards fast graph representation learning. *arXiv preprint arXiv:1809.05343*, 2018.

Karypis, G. and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998a.

Karypis, G. and Kumar, V. Multilevel algorithms for multi-constraint graph partitioning. In *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pp. 28–28. IEEE, 1998b.

Khayyat, Z., Awara, K., Alonazi, A., Jamjoom, H., Williams, D., and Kalnis, P. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 169–182, 2013.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

Lin, Z., Li, C., Miao, Y., Liu, Y., and Xu, Y. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 401–415, 2020.

Liu, Z., Wu, Z., Zhang, Z., Zhou, J., Yang, S., Song, L., and Qi, Y. Bandit samplers for training graph neural networks. *arXiv preprint arXiv:2006.05806*, 2020.

Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., and Dai, Y. Neugraph: parallel deep neural network computation on large graphs. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pp. 443–458, 2019.

Shui, Z. and Karypis, G. Heterogeneous molecular graph neural networks for predicting molecule properties. *arXiv preprint arXiv:2009.12710*, 2020.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph Attention Networks. *International Conference on Learning Representations*, 2018.

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., and Zhang, Z. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019a.

Wang, X., He, X., Wang, M., Feng, F., and Chua, T.-S. Neural graph collaborative filtering. In *Proceedings of the 42nd international ACM SIGIR conference on Research and development in Information Retrieval*, pp. 165–174, 2019b.

Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., and Owens, J. D. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 1–12, 2016.

Xiong, Z., Wang, D., Liu, X., Zhong, F., Wan, X., Li, X., Li, Z., Luo, X., Chen, K., Jiang, H., et al. Pushing the boundaries of molecular representation for drug discovery with the graph attention mechanism. *Journal of medicinal chemistry*, 63(16):8749–8760, 2019.

Yang, H. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3165–3166, 2019.

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., and Leskovec, J. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference* *on Knowledge Discovery & Data Mining*, pp. 974–983, 2018.

Zheng, D., Ma, C., Wang, M., Zhou, J., Su, Q., Song, X., Gan, Q., Zhang, Z., and Karypis, G. Distdgl: Distributed graph neural network training for billion-scale graphs, 2020.

Zhu, X., Chen, W., Zheng, W., and Ma, X. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 301–316, 2016.