Efficient Data Loader for Fast Sampling-Based GNN Training on Large Graphs

Youhui Bai, Cheng Li, *Member, IEEE*, Zhiqi Lin, Yufei Wu, Youshan Miao, Yunxin Liu, *Senior Member, IEEE*, and Yinlong Xu, *Member, IEEE*

Abstract—Emerging graph neural networks (GNNs) have extended the successes of deep learning techniques against datasets like images and texts to more complex graph-structured data. By leveraging GPU accelerators, existing frameworks combine mini-batch and sampling for effective and efficient model training on large graphs. However, this setup faces a scalability issue since loading rich vertex features from CPU to GPU through a limited bandwidth link usually dominates the training cycle. In this article, we propose PaGraph, a novel, efficient data loader that supports general and efficient sampling-based GNN training on single-server with multi-GPU. PaGraph significantly reduces the data loading time by exploiting available GPU resources to keep frequently-accessed graph data with a cache. It also embodies a lightweight yet effective caching policy that takes into account graph structural information and data access patterns of sampling-based GNN training simultaneously. Furthermore, to scale out on multiple GPUs, PaGraph develops a fast GNN-computation-aware partition algorithm to avoid cross-partition access during data-parallel training and achieves better cache efficiency. Finally, it overlaps data loading and GNN computation for further hiding loading costs. Evaluations on two representative GNN models, GCN and GraphSAGE, using two sampling methods, Neighbor and Layer-wise, show that PaGraph could eliminate the data loading time from the GNN training pipeline, and achieve up to 4.8× performance speedup over the state-of-the-art baselines. Together with preprocessing optimization, PaGraph further delivers up to 16.0× end-to-end speedup.

Index Terms—Graph neural network, cache, large graph, graph partition, pipeline, multi-GPU

1 Introduction

Recently, graph neural networks (GNNs) [1], [2], [3], [4] have been proposed to extend deep neural networks from handling unstructured data such as images and texts to structured graph data, and have been successfully applied to various important tasks including vertex classification [2], [3], [4], link prediction [5], relation extraction [6] and neural machine translation [7]. However, on the one hand, traditional graph processing engines like PowerGraph [8] lack auto-differentiation and tensor-based abstractions. On the other hand, current popular deep learning frameworks like TensorFlow [9] have insufficient support on vertex programming abstractions and graph operation primitives. To bridge the gap between these two fields, new systems like DGL [10], Python Geometric [11], MindSpore GraphEngine [12], and NeuGraph [13] were proposed to provide convenient and efficient graph operation primitives compatible with GNNs.

Many real-world graphs are on a *giant* scale. For instance, a recent snapshot of the Facebook friendship graph is comprised of 700 million vertices and more than 100 billion edges [14]. Additional to the graph structure data, the high-dimensional features (typically ranging from 300 to 600) associated with vertices and edges lead to greater computation and storage complexity. Under the time and resource constraints, it would be no longer efficient or even feasible to make a full giant graph train totally as a batch. So, a typical practice is sampling [2], [15], [16], which repeatedly samples subgraphs from the original graph as the input of a mini-batch, reducing the single mini-batch computation while still converging to expected accuracy.

Unfortunately, the sampling-based GNN training faces a severe performance problem of data loading. Specially, the data movement from host memory to GPU memory is incredibly time-consuming, when loading high-dimensional vertex features (e.g., user description, paper semantic embeddings [17]). For instance, we observe 74 percent of training time spent on data loading when training GCN [3] model over LiveJournal [18] dataset using a single GPU. The primary reason is that compared with the massive data size, a GNN model often uses deep neural networks with relatively low computational complexity. Thus, the GNN computation performed in GPU takes much shorter time than the data loading. Even worse, when multiple GPUs within a single machine are used to speed up the training, the demand for data samples loaded from CPU to GPU grows proportionally. Some optimized strategies, such as preprocessing [19], prunes the GNN models for better training performance. However, even with these optimized strategies, data movement still dominates the training process.

- *Youhui Bai, Cheng Li, Zhiqi Lin, and Yufei Wu are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui 230026, China. E-mail: {byh0912, zhiqi.0, yufeiwu}@mail.ustc.edu.cn, chengli7@ustc.edu.cn.*
- *Youshan Miao and Yunxin Liu are with the Microsoft Research, Beijing 100080, China. E-mail: {yomia, yunxin.liu}@microsoft.com.*
- *Yinlong Xu is with the School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui 230026, China, and also with the Anhui Province Key Laboratory of High Performance Computing, Hefei, Anhui 230026, China. E-mail: ylxu@ustc.edu.cn.*

Manuscript received 17 Dec. 2020; revised 28 Feb. 2021; accepted 3 Mar. 2021. Date of publication 12 Mar. 2021; date of current version 13 Apr. 2021.
(Corresponding author: Cheng Li.)
Recommended for acceptance by J. Zhai.
Digital Object Identifier no. 10.1109/TPDS.2021.3065737

This paper focuses on accelerating the sampling-based GNN training over large graphs on a multi-GPU machine. Our key idea is to reduce the data movement overhead between CPU and GPUs. This work is mainly based on the following observations. First, due to referencing vertices along with graph structure in GNN computation, different training iterations may use overlapped mini-batch data and exhibit a redundant vertex access pattern. As shown in the following case study (Section 2.3), compared to the vertex population of the target graph, it can load up to more than 4× the number of vertices during an epoch. Second, in each iteration, the training computation only needs to keep the sampled subgraphs corresponding to the current mini-batch, which only consumes a small fraction (e.g., no more than 10 percent) of GPU memory. Thus, the rest of GPU memory can be used to cache the features of the most frequently visited vertices for free, avoiding repeatedly loading from host memory. However, the current data loader hasn't explored the availability of spare GPU memory to achieve it.

Introducing caching to accelerate sampling-based GNN training faces the following system challenges. First, due to the random accessing patterns by the sampling nature, it's hard to predict which graph data will be visited in the next mini-batch. Instead of predicting the next visited graph data, we leverage the fact that the vertices with higher out-degrees have a higher probability of being sampled into a mini-batch. Following this, as our first contribution, we adopt a static caching strategy to keep frequently accessed graph data in GPU memory and introduce a new caching-enabled data loading mechanism. When a sampled mini-batch arrives at GPU, the required feature data will be fetched from both the local GPU cache and the host memory managed by the original Graph Storage Server.

Second, the current system design balances the training workloads across multiple GPUs while sharing a single copy of full graph data among them. This single graph parallelism makes the above caching solution inefficient. To avoid this inefficiency, as our second contribution, we adapt "data parallelism" to sampling-based GNN training with caching in a single multi-GPU machine, where each GPU works on its graph partition. Clearly, the data parallelism benefits are that data locality can be improved, and the number of cached vertices in total will be increased. To achieve this, we design a novel GNN-aware graph partition algorithm, which differs from the traditional general-purpose graph partition algorithms [20], [21] in two aspects. First, to balance the training workload between GPUs, we have to ensure each partition contains a similar number of training target vertices correlated to workload, rather than arbitrary vertices. Second, to avoid cross-partition visits when sampling, we replicate all its L-hop reachable neighbors for every train vertex in its partition. Since L is often not greater than 2 [2], [3], [4], and our partition algorithm is highly optimized, the resulted redundancy is acceptable.

Third, the performance improvement enabled by caching is proportional to the number of cached vertices with high out-degree. Given that GPU memory is often limited to 10–30 GBs, the caching mechanism may not be sufficient for supporting GNN computation over large graphs, most of whose vertices cannot be cached. For instance, for training GCN [3] over enwiki [22] graph, using a single GPU would be possible to cache only 51.2 percent of its vertices at maximum. This reduces the data loading time from 38.9 to 9.7 ms but still accounts for 34.3 percent of a single iteration training time. To complement caching and partitioning, we further explore the opportunity to hide the data loading overhead into the computation time. This requires us to design a new pipeline to parallelize the current mini-batch computation and the prefetching graph data for the next mini-batch.

We incorporate the above design ideas into PaGraph, a novel, efficient data loader for supporting sampling-based GNN training over large graphs on a single multi-GPU machine. PaGraph is further integrated with the Deep Graph Library (DGL) [10] and PyTorch [23]. We train two famous and commonly evaluated GNNs of GCN [3] and GraphSAGE [2] on seven large real-world graphs using two important sampling algorithms, namely, neighbor sampling [2] and layer-wise sampling [15]. Experimental results show that PaGraph completely masks the data loading overhead for each training epoch and achieves up to 4.8× speedup over DGL while converging to approximately the same accuracy within the same number of epochs, where the caching and pipeline strategies contribute about 70-75 and 25–34 percent to the overall speedup, respectively. Together with preprocessing optimization, PaGraph even achieves up to 16.0× speedup over DGL.

In summary, we make the following contributions:

- An analysis of sampling-based GNN training on single-server with multi-GPU, demonstrating that the training performance has become bottlenecked by data loading.
- A novel data loader PaGraph to incorporate a static GPU cache, reducing the data volume shipped from host memory to GPU. Its cache efficiency is further improved by (1) partitioning graphs for leveraging better data locality for multi-GPU GNN training, and (2) pipelining the data loading and GNN computation, which mitigates the potential GPU memory contention between on-GPU caching and computing.
- An in-depth evaluation of PaGraph compared to DGL, a state-of-the-art GNN library, over two representative GNN models and seven real-world graphs.
- A visionary discussion on the limitations, application scope, generalization opportunities and challenges, and future directions of PaGraph.

2 Background and Motivation

2.1 Graph Neural Networks

In this work, we target attributed graphs, where vertices or edges are associated with a large number of features (often more than hundreds), in addition to the graph structural information. A GNN model often consists of multiple layers. Fig. 1 shows a two-layer GNN model's architecture, where the blue and green color represents the first and second layers, respectively.

The computation crossing different layers follows a traditional *vertex-centric graph iterative processing* model [8], [24]. At each layer, every vertex follows its incoming edges to

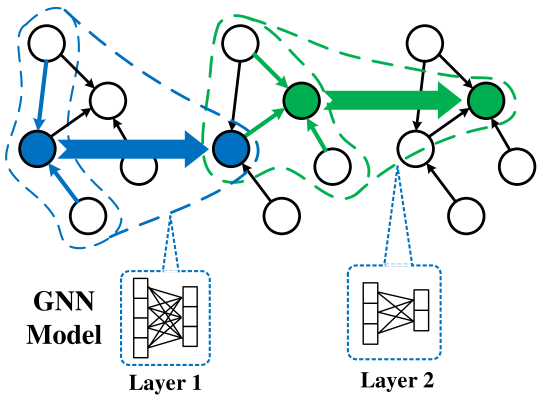


Fig. 1. A two-layer GNN architecture.

aggregate features from neighbors (thin arrows), then uses neural networks to transform the features into an output feature (thick arrows), which will be fed into the next layer as input feature [2]. Within the same layer, all the vertices share the same aggregation neural network and transformation neural network. A single GNN layer could only achieve information passing from direct (1-hop) in-neighbor vertices. With 2 GNN layers, by ingesting the previous layer's output into the next layer as input, we could connect 2-hop in-neighbors. Similarly, in a *L-layer* GNN, vertices can collect information from *L-hop* in-neighbors. The output of the last layer could be used to enhance tasks such as vertex classification [2], [3], or used as embeddings for relation extraction [6].

2.2 Sampling-Based Training With GPU

Training a GNN model to reach a desirable accuracy often requires a few tens of *epochs*, each of which is defined as a full scanning of all train vertices of a target graph. An epoch consists of a sequence of iterations, during each a mini-batch of train vertices will be randomly selected for evaluating and updating that model. However, unlike training data such as images and sentences where each data sample is independent, graph data is highly structural connected. As a result, training on a vertex requires not only loading the vertex associated data, but also the data of its linked edges and connected vertices, making the data loading in GNN quite different from traditional machine learning training.

The unique data access pattern of GNN computation makes it hard to efficiently handle the exponential growth of the number of vertices and their associated features for training w.r.t the number of GNN layers. To address this issue, instead, sampling, as a typical optimization solution, is widely adopted [2], [15], [16], [19]. Sampling-based training selects a limited number of a mini-batch's layer-hop reachable neighbors and achieves almost the same training accuracy with much less computational cost [19].

In addition, the graphics processing unit (GPU) provides unprecedented computational density and programming flexibility, supported by almost every single deep learning frameworks including PyTorch [23], TensorFlow [9] and MXNet [25], making it the de facto standard accelerator for deep learning training. A GPU consists of a large number of powerful processors and high-bandwidth dedicated GPU memory, and is a great choice for performing tensor-related computations. As such, existing GNN libraries such as DGL

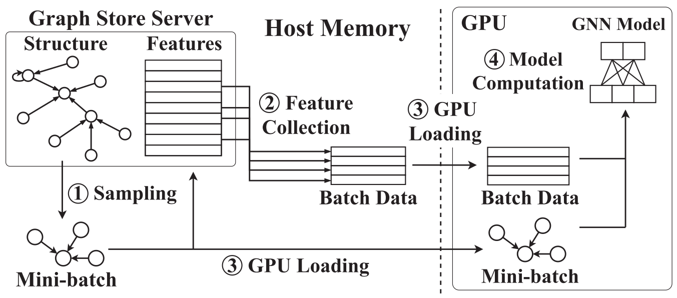


Fig. 2. Sampling-based GNN training workflow, where data loading corresponds to stages ②- ③.

follow the same practice to use GPUs for training acceleration. Our experiments show that when training the GCN [3] model over the LiveJournal [18] dataset, a single 1080Ti GPU provides 3× the performance of the CPU-only (16 cores) setup. And the performance gap would be gradually amplified as more GPUs are involved.

Fig. 2 diagrams the workflow of sampling-based GNN training over GPU. There is a global Graph Store Server, which manages the full graph structure along with feature data in CPU memory. Every training iteration involves three major steps, namely, sampling(①), data loading(②-③) and model computation(④). In each iteration, a *Data Sampler* randomly collects a number (mini-batch size) of train vertices, and then traverses the graph structure and samples their *L-hop* neighbor vertices to form the input data samples(①). As a representative example among sampling methods, neighbor sampling (NS) [2] simply picks a small number of random neighbors from the currently visiting vertex. For instance, in a 2-neighbor sampling, as shown in Fig. 3, for train vertex v_1, it selects v_2, v_5 from its 1-hop in-neighbors. Then, it samples 2-hop in-neighbors v_6, v_2, v_9, v_8 from the selected 1-hop vertices v_2, v_5's direct neighbors. The data loading step prepares the feature data for GPU computation. Here we would emphasize that data loading is not just transferring data from CPU to GPU via PCIe. Instead, the data loading procedure goes through two stages as follows. A *Data Loader* picks up a mini-batch and queries the graph store to gather the features of all vertices from that batch (②), and loads these samples into GPU memory via a PCIe link (③). Finally, a *Trainer* at CPU launches the GPU kernels to perform GNN computation on GPU over the loaded data samples (④). The process will run iteratively until the target model converges.

2.3 Problems and Opportunities

Though levering the powerful computational resource offered by GPU, there is still a large room for improving GNN training

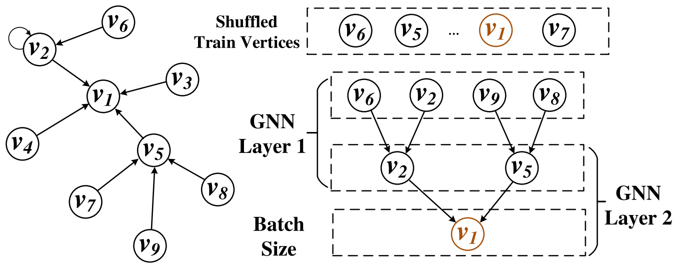


Fig. 3. Sampling a mini-batch for a 2-layer GNN using the neighbor sampling algorithm. (sampling neighbor = 2, mini-batch size = 1).

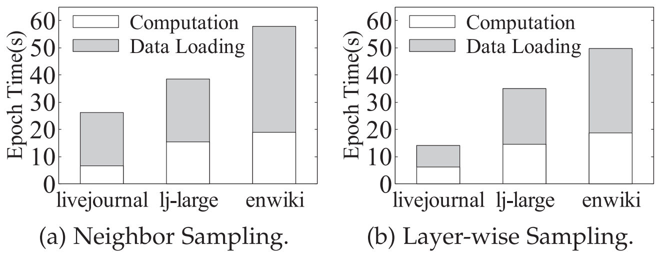


Fig. 4. Data loading and computation time in an epoch of training a 2-layer GCN using two sampling methods.

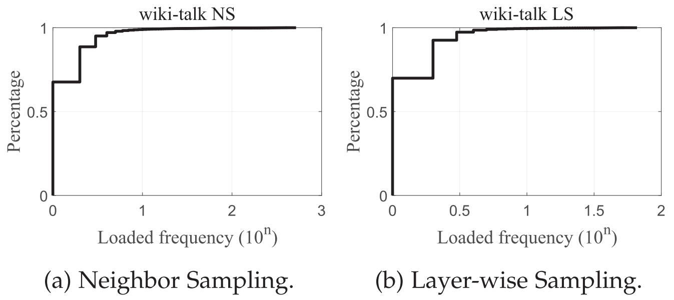


Fig. 5. CDF (Cumulative Distribution Function) for the access frequencies of vertices when training GCN over the wiki-talk dataset using Neighbor Sampling (NS) and Layer-wise Sampling (LS), respectively.

with GPUs. In particular, sampling-based GNN training over GPU suffers from a severe data loading problem that needs to be resolved. To understand this, we train a 2-layer GCN [3] as our walk-through example over 1 to 4 GPUs. We use the widely-used neighbor sampling and layer-wise sampling (LS) [15] to create mini-batches of vertices for each training iteration. LS is almost identical to NS except it considers a layer as a whole and constrains the total number of sampled vertices per layer rather than per vertex, therefore avoiding the number of sampled vertices growing exponentially with deeper layers. As suggested by existing work [15], [19], the neighbor sampling method here selects 2 neighbors for each vertex, while the layer-wise sampling method limits the number of vertices sampled per layer to 2,400. As follows, we report our major observations of performance inefficiencies and reveal their root causes, which together motivate the design of our work. We begin with the single-GPU training with the popular DGL library [26] and a GTX-1080Ti GPU.

The Data Loading Dominates the Training Time. Throughout experiments with 3 large real-world graphs, livejournal [18], lj-large [27] and enwiki [22], we find that data loading from CPU to GPU usually dominates the end-to-end GNN training time. In the experiments, we adopt a representative GNN model, a 2-layer GCN [3], and use the widely-used neighbor sampling [2] and layer-wise sampling [15] to create mini-batches of vertices for training performance understanding. Fig. 4 summarizes the training epoch time over different graphs and shows the time break down into data loading and computation. Note that we omit the sampling overhead since sampling runs faster than and is overlapped with data loading already. Clearly, across all three graph datasets, data loading takes much longer time than computation. For example, GCN on livejournal spends 74 and 56 percent of the end-to-end training time on data loading when using neighbor and layer-wise sampling, respectively. This situation will become worse, when multiple GPUs are used to train a shared model collectively, since the computation time will be reduced and the data loading overhead will become more dominating.

With a deep understanding of results, we identify the following factors that seriously slow down the whole training process. Meanwhile, they also provide us with opportunities to further improve the performance of sampling-based GNN training with GPUs.

Redundant Vertex Access Pattern. We continue to understand the total amount of data sent from CPU to GPU for completing a training epoch. Surprisingly, the loaded data volume can be up to more than 4× the total number of vertices of the target graphs in our experiments. This indicates that some vertices are loaded multiple times. To validate this, we collect the vertex visiting trace across different training jobs and count the number of visits per-vertex basis. Fig. 5 is a CDF graph for the access frequencies of loaded vertices for training GCN over the wiki-talk graph, when using both the NS and LS Sampling method. We observe more than 32.4 percent of vertices have been repeatedly used by up to 519 times. This redundant vertex access pattern exacerbates the data loading burden, and creates data loading of tens of gigabytes for each epoch. We further discover that those vertices have higher out-degree than other less frequently visited or non-visited vertices. This is because a vertex with a high out-degree in a graph is likely connected with multiple train vertices, making it have chances to be selected multiple times by different mini-batches.

Based on this observation, we draw an inspiration that caching in GPU memory the feature information of vertices with high out-degrees will reduce the data loading volumes shipped from CPU to GPU, and thus accelerate the sampling-based GNN training. However, this optimization would impose a few challenges, such as contending GPU memory with GNN computation, incurring overhead for maintaining such a cache, etc.

GPU Resource Underutilized. Since the data loading phase precedes the GPU training phase, we further explore the negative impacts of the data loading bottleneck. We summarize the resource utilization in Table 1 where two GNN models (GCN and GraphSAGE) are being trained. Surprisingly, regardless of the sampling method, only a small fraction of the computing and memory resources on GPU have been utilized. For instance, with neighbor sampling, only around 20 percent of the GPU computational resources are in-use, with even less memory consumption, e.g., less than 10 percent. This is because the mini-batch data that CPU sent to GPU is not sufficient to fully explore the hardware parallelism in GPU. In the meanwhile, the GPU is idle waiting for training data samples to arrive at most time.

TABLE 1
GPU Resource Utilization of Training Different Models Using Both Neighbor and Layer-Wise Sampling

Utilization	GraphSAGE		GCN	
	Neighbor	Layer-wise	Neighbor	Layer-wise
Computation	21.22%	13.82%	17.12%	13.87%
Memory	7.17%	5.11%	7.94%	4.75%

This observation leads us to consider to leverage spare GPU resources for caching the feature information of frequently visited vertices as possible for cheap re-usage. Theoretically, this caching solution could eliminate the amount of feature information that should gathered and sent from CPU to GPU, which consequently eliminate the CPU feature collection and PCIe data transferring bottlenecks from the sampling-based GNN training.

High CPU Contention Between Sampling and Feature Collection. We then break down the time spent in different stages of the data loading process and find that the feature collection phase is CPU-intensive and takes much longer than the CPU-GPU data movement. For instance, it accounts for 50.4, 55.1, and 56.3 percent of the total data loading time, for the three used datasets, respectively. This surprising result leads to: (1) with a single GPU, we achieved about 8 GB/s PCIe bandwidth utilization at maximum (the capacity is 16 GB/s), while the average utilization is lower; (2) with multiple GPUs, concurrent workers for collecting features would contend CPU resources with samplers, e.g., the time for sampling and feature collection increased by 88 percent and 59 percent over the 1-GPU case, respectively, while the GPU computation time remains unchanged. In the 4-GPU case, the maximum PCIe bandwidth utilization drops to half and the average is even worse. This indicates that the CPU capacity cannot cope with the GPU computation demands, given the large amount of data required by each iteration. As a consequence, to reduce the feature collection cost, we have to consider to reduce the amount of data it should be gathered in this phase, as well as to isolate the resource allocation for both sampling and feature collection.

Sequential Execution of Data Loading and GNN Computation. With the dominating library DGL, though data loading consumes CPU and PCIe, and GNN computation is scheduled to GPU, they are still executed in a serial order. DGL does not leverage to overlap the two stages in the training pipeline, mainly because the data loading dominates the whole training space and the GNN computation runs faster. However, in our work, adopting caching significantly impacts the training pipeline by reducing the data loading cost while increasing the computation density as more data samples are fed. Therefore, the new situation provide us with opportunities to pipeline the data loading and GNN computation to hide one's cost into the other's, and vice versa. This pipeline design also introduces a positive effect to improve caching efficiency when facing large graphs as it can reduce the amount of data that need to be cached in GPU memory.

3 PaGraph

Motivated by the experimental results presented in Section 2.3, we propose PaGraph, a novel, efficient data loader to enable fast sampling-based GNN data parallel training on large graphs. We introduce three key techniques to PaGraph: 1) a GNN computation-aware caching mechanism for reducing the data loaded from CPU to GPU, 2) a cache-friendly data parallel training method to scale GNN training on multiple GPUs, and 3) a two-stage training pipeline to overlap data loading and GNN computation.

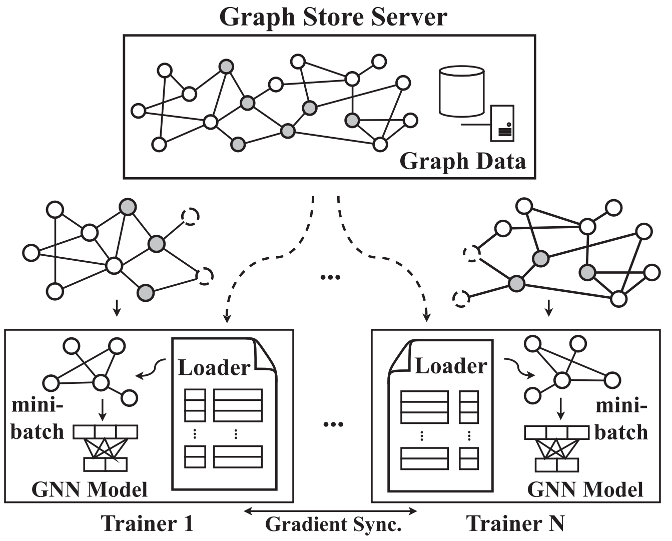


Fig. 6. Overall architecture of PaGraph. *For simplicity, directions of edges in graphs are omitted.*

Fig. 6 shows the overall architecture of PaGraph. For a single machine with N GPUs, the whole graph is partitioned into N sub-graphs. The graph structural information of these partitions together with the vertex feature data are stored in a global *Graph Store Server* in the CPU shared memory. There are N independent *Trainers*. Each *Trainer* is responsible for training computation on a dedicated GPU. It replicates the target GNN model, and only consumes its own data partition. Each GPU contains a *cache*, which keeps features of frequently visited vertices.

During each training iteration, each *Trainer* receives the graph structural information of a mini-batch produced by a *Sampler*. As shown in Fig. 7, it also gets the associated vertex features gathered by a *Data Loader* by either directly fetching from its own GPU *cache* if they are cached (marked as green), or querying the *Graph Store Server* for missing features (marked as yellow). The former case saves the data copying overhead from CPU to GPU, while the latter one cannot. After that, the structural data, and both the cached and loaded (from the host) vertex features of that mini-batch are fed into the GNN model to compute gradients. *Trainers* do not interact with each other except synchronizing locally produced gradients at the end of each iteration among peers to update the model parameters.

3.1 GNN Computation-Aware Caching

Caching Policy. To generate better models, for each epoch, most training algorithms require a randomly shuffled sequence of training samples, which makes it impossible to predicate the vertices in each mini-batch at runtime. The neighbors of a vertex are also randomly selected and thus unpredictable as well during training. Therefore, it is hard to foretell which vertex is most likely to be accessed at the next mini-batch. However, due to the unique access patterns of the neighbor-sampling method, the out-degree of a vertex indicates the probability of it being selected throughout the whole epoch. This says that with a higher out-degree, a vertex is more likely to be an in-neighbor of other vertices, and thus more likely to be sampled in a mini-batch. Thus, it is sufficient to fill up the cache with high out-degree vertices.

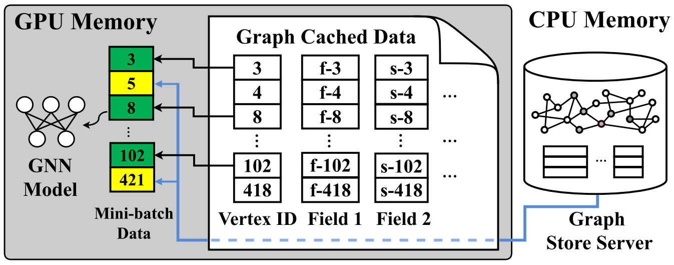


Fig. 7. New data loading flow with caching.

Usually, a dynamic caching policy is favored. However, it is not suitable in our situation where the *cache* is inside GPU. This is because GPU cannot work stand-alone, and all computations performed at GPU must be assembled into GPU kernels and launched by CPU. Most current GNNs are lightweight [1], and hence graph data swapping between CPU memory and GPU memory has intolerable overhead during training (see Section 2.3). Therefore, instead of making on-the-fly decisions on what to be cached, e.g., LRU [28], we use static caching to avoid the overhead of dynamic data swapping. To do so, we can pre-sort vertices by out-degree offline, and select top high out-degree vertices at runtime to fill up the GPU cache. We leave the exploration of efficient dynamic caching policies as future work. Though it is simple, as shown in Section 5, this static caching policy effectively achieves a high cache hit ratio.

Cache Memory Space. To avoid resource contention with the high priority training computation, we need to estimate the maximum amount of available GPU memory for the cache allocation. To achieve this, we leverage the fact that memory consumption is similar across training iterations. This is because the sampling-based mini-batch training uses almost the same amount of data samples as input and performs almost the same amount of computation to train a shared GNN model for each iteration. As a result, it is sufficient to decide the right cache size via a one-time sampling of GPU memory usage. In more detail, right after the first mini-batch training, we check the size of free GPU memory during training and allocate the available GPU memory for caching graph data accordingly (see Section 4 for more details).

Data Management. In the GPU *cache*, we manage the cached vertex features by maintaining two separate spaces. First, we allocate consecutive memory blocks for feature data. The cached feature data of vertices is organized as several large $[N, K_i]$ matrices, where N denotes the number of the cached vertices, and K_i is the dimension of features under the ith feature-name fields. Second, to enable fast lookup, we organize the vertex meta data into a hash table to answer whether the queried vertex is cached and where it locates for later retrieval. The meta data is far less than the cached feature data, e.g., no more than 50 MB for a partition with 10 million vertices.

Discussions. Less skewed or massive graph would limit the efficiency of the cache optimization. However, many real-world graphs including the evaluated ones are power-law graphs [8], [29], [30], and exhibit high skewness. Therefore, our solution could benefit common GNN training jobs. One possible solution for those with less skewed or large graphs would be combining prefetching and dynamic caching to achieve good cache efficiency. We address this issue in Section 3.3.

3.2 Data Parallel Training and Partition

The current design of GNN systems such as DGL, balance computation across multiple GPUs but make them share a single copy of graph data [26]. When directly applying the above GNN-aware caching method to this setting, we observe a cache inefficiency phenomena, i.e., the cache hit ratio keeps decreasing with the increasing number of GPUs. This is because the single graph serves the data visiting locality for parallel *Trainers* on multiple GPUs, and thus all GPU *caches* would keep similar vertices.

To address this cache inefficiency, we introduce "data parallelism" to PaGraph, which has been widely applied to leverage multiple GPUs to train neural network models efficiently. In our system, rather than accessing a shared graph, a Trainer consumes its data partition (subgraph), performs the training computation to get local gradients, and then exchanges gradients among peers to update its model replica synchronously. Clearly, the benefits of data parallelism are that data locality can be improved and the number of cached vertices in total will be increased. To make this happen, although there exist numerous graph partition algorithms [8], [20], [21], we still need to design a new one to meet the following two goals specific to data parallel GNN training. First, it should keep computation balanced across different Trainers, as unbalanced computation may result in a different number of mini-batches per epoch for different Trainers. This will break gradient synchronization and get training stuck. Second, it needs to avoid cross-partition accesses from different Trainers as possible.

Computation Balance. To achieve computation balance across different Trainers, all the partitions should have a similar number of train vertices. Assume that we need K partitions. We scan the whole train vertex set, and iteratively assign the scanned vertex to one of K partitions. During every iteration t, a train vertex v_t is assigned with a K dimension score vector, where the ith element represents the feasibility for assigning the vertex to the ith partition for $i \in [1, K]$. The score is computed by Eq. (1).

$$score_{v_t}^{(i)} = |TV_i \cap IN(V_t)| \cdot \frac{TV_{avg} - |TV_i|}{|PV_i|}, \quad (1)$$

TV_i represents the train vertex set already assigned to the ith partition. $IN(V_t)$ denotes the L-hop in-neighbor set of train vertex v_t. PV_i controls the workload balance, and denotes the total number of vertices in the ith partition, including the replicated vertices. v_t is most likely to be assigned to a partition which has smallest PV. TV_{avg} is the expected number of train vertices in the final ith partition. To achieve computation balance, we set TV_{avg} as $\frac{|TV|}{K}$, which indicates that all partitions will get almost the same number of train vertices.

Self-Reliance. For the queries including the edges across different partitions, they must be forwarded to the Graph Store Server to get a full set of neighbors. Inspired by [8], [31], PaGraph introduces minimum extra vertices and edges in each partition to deal with cross-partition edges. Fig. 8 demonstrates how to partition a graph with self-reliance for a one-layer GNN model. The shaded vertices denote the train vertices, the white vertices are val/test vertices, and the dashed vertices are the introduced redundant vertices.

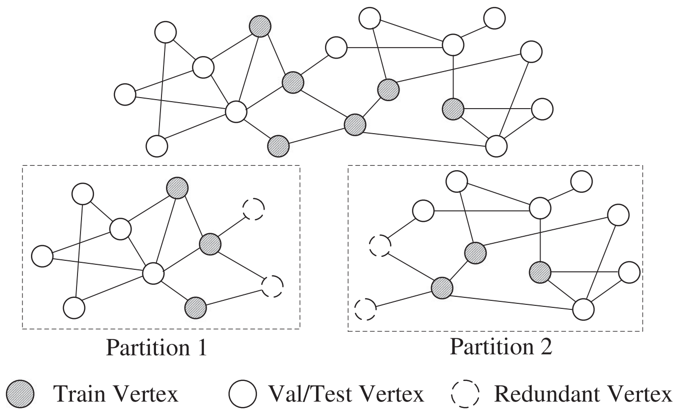


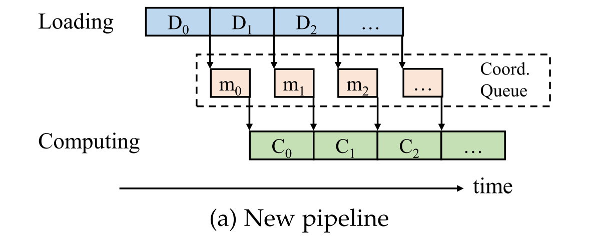
Fig. 8. Self-reliance graph partition for a one-layer GNN model. *For simplicity, the edge directions are omitted.*

For each partition, PaGraph extends the sub-graph with redundant vertices and edges to include all the neighbor vertices of required hops during sampling. For GNN models with L GNN layers, we will include L-hops in-neighbor vertices for each train vertex, e.g., a one-layer GNN model only requires to include direct in-neighbors of each train vertex. PaGraph only brings in necessary edges for the extended vertices to satisfy the required message flow during training. Note that the extended vertices may include train vertices. These extended train vertices are regarded as mirrors [8] and will not be trained. In this way, partitions are independent from each other. Each Trainer can sample mini-batches entirely from its own subgraph without accessing the global graph structure.

3.3 Pipelining Data Loading and GNN Computation

As mentioned before, caching's performance improvement is proportional to the number of cached vertices with high out-degree. Although most real-world graphs exhibit high skewness, given that GPU memory size is often limited to 10–30 GBs, the stand-alone caching mechanism may not be sufficient for supporting GNN computation over large graphs, most of whose vertices cannot be cached. Therefore, in those cases, data loading will remain as a bottleneck. To complement caching and partitioning, we further explore the opportunity to hide the data loading overhead into the computation time. This requires us to design a new pipeline to parallelize the current mini-batch computation and the prefetching graph data for the next mini-batch.

Fig. 9a shows our two-stage training pipeline design, where we break the original sequential execution into two parallel streaming executions, namely, loading and computing. We use a message queue to coordinate the execution of two streams. By taking the input from *Sampler*, the loading streaming executor is responsible for organizing the required feature information for the selected mini-batch vertices from both graph store and GPU cache (see Fig. 9b). When loading is done, it posts a ready message including the location of batched data in GPU memory to the shared message queue. On the other hand, the computing executor sits in a loop and periodically checks the arrival of new messages from the loading executor. It pops a message from the head of the shared queue and schedules the corresponding



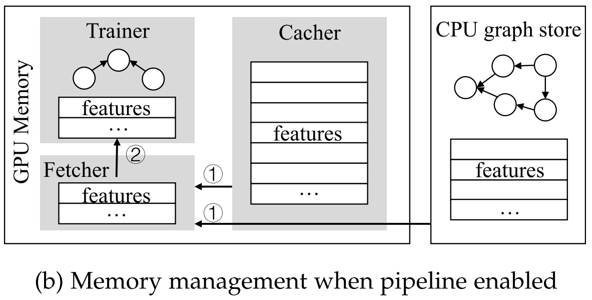


Fig. 9. The overall design for pipelining data loading and GNN computation.

GNN computation, which will consume the already prefetched data.

The new pipeline design leads us to further partition GPU memory into three parts for GNN computation, graph data caching, and buffering prefetched data, respectively (See Fig. 9b). However, we claim that the prefetching buffer will not exacerbate the GPU memory tension due to the following reasons. First, each mini-batch data consumes less than 900 MB memory space, taking only 8 percent of GPU memory on the one device used in our evaluation. Second, we put a limit on the maximal number of prefetching tasks to avoid memory contention. We also use the length of the message queue as feedback to guide the loading executor to adaptively slow down or speed up prefetching.

4 Implementation

We built PaGraph on the top of the Deep Graph Library (DGL (v0.4)) [10] and PyTorch (v1.3) [23]. Since the gradient synchronization is not a primary concern in GNN training, we just adopt an existing solution, NCCL RingAllReduce, to implement data parallel GNN training [32]. We use the Graph Store Server implemented by DGL to store graph structural data and feature data in CPU shared memory.[1] We store the full graph structure as an adjacency matrix of CSC format [33]. CSC format aggregates in-neighbor sets of each vertex inside a consecutive memory space and thus allows fast access to fetch them. The overall implementation consists of 1.47 K lines of code in Python, where 293 lines are for the *Data Loader*, 436 lines correspond to the partition algorithm, and 737 lines are changes made to the original sampling training workflow. We are refactoring the code base and planning to make it available online to support reproducibility.

Cache Initialization and Maintenance. We extend *Data Loader* to incorporate simple Python APIs for fetching required data from local GPU memory or CPU memory, as demonstrated in

1. For larger graph that cannot reside in the main memory, we leave the tiered storage for solving this problem in future work.

```
import PaGraph.storage as storage
# create loader
loader = storage.GraphCacheServer(
    graph_store_server, num_nodes,
    node_mapping, trainer_id)
# auto cache graph data
loader.auto_cache(graph, field_names)
# auto fetch data from CPU and GPU
loader.fetch_data(mini_batch)
```

Fig. 10. APIs to extend *Data Loader* for cache-enabled data loading.

Fig. 10. First, we initialize *Data Loader* by connecting it to Graph Store Server and making it aware of data parallel training configurations, e.g., how many GPUs are used, which GPU it serves for, etc. During the first mini-batch training, *Data Loader* will check the total GPU memory, denoted as $total\_mem$, and the peak GPU memory allocated by PyTorch, denoted by $used\_mem$. We also preserve a certain size of GPU memory unused for memory fluctuations during training (denoted as $preserved\_mem$), which we found 1.0 GB is enough in practice. After that, we call *loader.auto_cache*, which calculates the size of available memory and assigns this amount of memory to cache by subtracting $used\_mem$ and $preserved\_mem$ from $total\_mem$. Once the cache is allocated, *Data Loader* continuously loads into *cache* features (specified by *field_names*) of vertices with top high out-degrees until *cache* is full. To reduce the time cost of this initialization process, we offline analyze the sub-graph structure, and rank vertices based on their out-degrees. From the second iteration on, *Data Loader* fetches data from Graph Store Server and local GPU *cache* by calling *loader.fetch_data*, which is parameterized by *mini_batch*. This new loading flow is transparent to training jobs.

Algorithm 1. Computation-Balanced and Cross-Partition Access Free Graph Partition

```
  Input: graph $\mathcal{G}$, train vertex set $TV$, number of hops $L$, partition
         number $K$
  Output: graph partition $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_k$
1 for $i \leftarrow 1$ to $K$ do
2     $\mathcal{G}_i \leftarrow \varnothing$                    // Initialization
3 for each train vertex $v_t \in TV$ do
4     $IN(v_t) \leftarrow$ IN-NEIGHBOR($v_t, \mathcal{G}, L$)
5     $score_{v_t} \leftarrow$ SCORE($v_t, IN(v_t)$)
6     $ind \leftarrow \arg\max_{i \in [1,K]}\{score_{v_t}^{(i)}\}$
7     $\mathcal{G}_{ind} \leftarrow \mathcal{G}_{ind} \cup \{v_t\} \cup \{IN(v_t)\}$
8 return $\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_K\}$
```

Offline Graph Partition. Algorithm 1 presents our GNN-specific partitioning method for splitting large graphs to fit caching-enabled data parallel training. We implement our partition algorithm based on Linear Deterministic Greedy (LDG) [20], a stream-based partition solution. We only evaluate the train vertices (line 3), assign them to the target partition with index ind (line 6) based on the $scores$ computed (line 4-5), and also include their L-hop in-neighbors (line 7) into a partition. Preparing partitions for GNN model training is an offline job before training. Although this step introduces extra time costs and consumes additional resources, it is acceptable due to the following reasons. First, the partition is a one-time offline job, and thus will not block the training process. Second, recent studies [34], [35] find that a number of machine learning jobs are generated by parameter tuning experiments and they share most configurations include dataset and partitions. Therefore, this one-time job can be beneficial across these parameter tuning experiments.

To reduce the storage burden of partitions, we remove the redundant vertices and edges that don't make contributions during training. For a given L-layer graph neural network, we exam whether a val/test vertex is out of L-hops away from all train vertices. If so, we remove this vertex and its associated edges from the sub-graph. Also, we remove redundant edges to avoid inefficient message flow. Undirected edges will be converted into directed edges if one of the message flow direction is unnecessary. With the refinement of graph structure, graph data associated with redundant vertices and edges is also removed.

Pipelining. As DGL already overlaps the mini-batch sampling with the rest steps on the data loading path, we only consider hiding the data loading cost into the computation. Initially, we introduced a daemon thread to prefetch the next mini-batch while the current one is under computation. Unfortunately, this version is inefficient since the legacy GIL (Global Interpreter Lock) feature of python serializes multiple threads. To avoid this unnecessary serialization, instead, we launch a daemon process to perform the target prefetching task, and make the communication between processes managed through a PyTorch Queue. However, this multiprocess solution also brings extra cost spent on copying the prefetched mini-batch data from the shared Queue to the memory space of the computation process, because of the process isolation. In order to further hide this non-negligible cost, we additionally spawn a daemon thread from the main process to asynchronously copy data from Queue to the main process in the background.

Resource Isolation. GNN libraries are usually implemented on top of deep learning frameworks and add additional graph storage and sampling features to these frameworks. In their implementations, both sampling and data loading are placed in a single process and simultaneously use OpenMP [36] to perform concurrent jobs, such as sampling multiple mini-batches, gathering the mini-batch data, and copying it into a protected pinned memory space, etc. We observed interference between sampling and data loading with a single process, where both of them compete CPU resources. This interference also slows down the frequency of the kernel launching from CPU host to GPU device. To eliminate this resource contention, in our implementation, we make sampling and data loading use separate processes, and tune the OpenMP configurations to balance CPU resources between them.

Local Shuffling. To achieve better empirical performance [37] in data parallel training, data samples need to be shuffled. *Shuffling* can be done across partitions (globally) or within each partition (locally), where the former case may lead to faster convergence, but is more costly than the latter one. However, local shuffling is widely adopted in real-world practice, and recent studies show that it can still perform well with slightly slower convergence [38]. Therefore, PaGraph locks the assigned partition for each Trainer and shuffles the accessing order of data samples in that

TABLE 2
Statistics of Datasets. (K: Thousand. M: Million, B: Billion)

Dataset	vertex#	edge#	feature	label
reddit	232.96 K	114.61 M	602	41
wiki-talk	2.39 M	10.04 M	600	60
livejournal	4.04 M	69.46 M	600	60
lj-link	5.20 M	103.55 M	600	60
lj-large	10.69 M	224.61 M	400	60
enwiki	12.15 M	756.28 M	400	60
friendster	64.19 M	2.15B	400	60

partition before every epoch begins. We further show in Section 5.9 that local shuffling does not impact both the model accuracy and the convergence speed.

Discussions. Currently, PaGraph works on a single multi-GPU server, but the core idea of caching, graph partition and pipelining can be directly applied to distributed GNN training to leverage even more GPUs for handling larger graphs that cannot be fitting into the memory of a single server. The only thing that needs to design in that extension carefully is to efficiently synchronize gradients among servers, so as to avoid the synchronization bottleneck.

5 Evaluation

We intensively evaluate the performance of PaGraph using representative GNN models and real-world datasets. In particular, we explore the single-GPU performance and multi-GPU scaling efficiency, conduct a breakdown analysis on the cache performance under different conditions and quality of our partition algorithms, and understand the joint effects of caching and pipelining when facing large graphs.

5.1 Experimental Setup

Environments. We deploy experiments on a multi-GPU server, which consists of dual Intel Xeon E5-2620v4 CPUs, 512 GB DDR4 main memory and 4 NVIDIA GTX 1080Ti (11 GB memory) GPUs with no NVLink connections. The machine is installed with CentOS 7.6, CUDA library v10.1, DGL [10] v0.4, PyTorch [23] v1.3.

Datasets. We use seven real-world graph datasets listed in Table 2 for evaluation, including reddit [2] social network, wiki-talk [39] page edition history network, three variants of the livejournal [18], [27], [40] communication network (livejournal, lj-link, lj-large), and enwiki [22] wikipedia links network. The "feature" column represents the dimension of vertex features, and the "label" column shows the number of vertex classes. Following the setting of Reddit in [2], we split the vertices in each dataset into train, val, and test vertex categories with a 65:10:25 ratio.

Sampling Methods. We adopt Neighbor Sampling [2] and Layer-wise Sampling [15] combined with Skip Connection [16] to evaluate PaGraph. A recent work [19] already shows neighbor-based sampling training can achieve comparable model performance with full graph training, where only two neighbors are sampled at each neighborhood layer. Thus, in our evaluation, we choose to follow their suggestion that two random neighbors will be sampled for a vertex in each layer to form a mini-batch. We set the training batch size as 6,000 for the whole of evaluation, following

TABLE 3
Model Architectures

Model	Layer#	Agg.	Apply	Hidden-dimension
GCN	2	sum	FC	32
GraphSAGE	2	mean	FC	16

(FC: Fully-Connected Layer. Agg: Type of Aggregating Operation. Hidden-Dimension: Hidden Embedding Dimension of the First Layer Output.)

the suggestion from DGL [26]. We set the layer size as 2,400 for layer-wise sampling as suggested by FastGCN [15]. Due to space limit, we report results corresponding to neighbor sampling at most time, while only showing results of layer-wise sampling in Sections 5.7 and 5.9. We observe similar trends across the two sampling methods, proving that our solution is applicable to various sampling methods.

Models. We use two representative GNN models, Graph Convolutional Network (GCN) [3] and GraphSAGE [2] to evaluate PaGraphwith detailed configurations in Table 3. *GCN* generalizes the convolution operation on graphs as follows. Each vertex in a GCN layer aggregates the features of its neighbor vertices using a *sum* operation. Then the aggregated feature goes through a fully-connected layer and a ReLU activation to generate output representations. GCN has shown its excellence in tasks such as classifications [3] and neural machine translation [7]. *GraphSAGE* [2] is an inductive learning model to learn the different aggregation functions on different number of hops. There are four aggregation types in GraphSAGE: GCN(sum), Mean, LSTM and Pooling. We choose to present the experimental results of the GraphSAGE-Mean model since all four aggregators show the similar execution cost [2].

Baselines. We deploy the original DGL as the first baseline. In addition, we run an advanced DGL where the preprocessing [19] optimization is enabled as another baseline, denoted as "DGL+PP". The idea behind preprocessing is to remove the first layer of GNN model by aggregating the corresponding features offline. Even though, the data loading still dominates the performance of preprocessing-enabled training and makes it fail to take most advantage of this optimization. Unless stated otherwise, all the performance numbers are the average of results from 10 epochs.

5.2 Single GPU Performance

We first evaluate the training performance of PaGraph on a single GPU. Fig. 11 shows the training performance of GCN and GraphSAGE on different datasets. Overall, PaGraph achieves training performance speedups from 2.4× (lj-large) to 3.9× (reddit) for GCN and from 1.3× (lj-large) to 4.9× (reddit) for GraphSAGE, compared to DGL. We observe that the combination of preprocessing and DGL (DGL+PP) behaves differently across the two GNN models, i.e., the performance speedup of GCN achieved by preprocessing is better than GraphSAGE. This is due to the different forwarding procedures used in GCN and GraphSAGE. During forwarding in GCN, each vertex aggregates the features from its neighbors and sum-reduces these features into one feature. Unlike GCN, for each vertex in the corresponding mini-batch, GraphSAGE has to save its own feature and the aggregated feature from its neighbors. This leads GraphSAGE to more

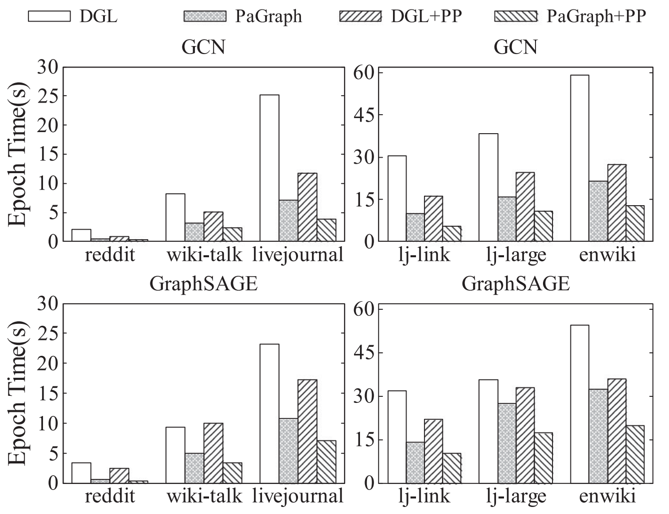


Fig. 11. Single-GPU training performance of GCN (top) and GraphSAGE (bottom). (PP: preprocessing optimization).

CPU-GPU data transfer compared with GCN. In contrast, we further observe that our optimizations in PaGraph can better exploit the potential of the preprocessing optimization, other than the vanilla DGL. For instance, PaGraph+PP improves the performance of DGL+PP by 2.1× to 3.0× and 1.8× to 6.2× for both GCN and GraphSAGE across the first 6 datasets.

Training Time Breakdown. To further explore the data loading overhead reduced by PaGraph, we break down the GCN training time on both DGL and PaGraph into GPU computation time and CPU-GPU data loading time. We collect such system statistics using nvprof [41] and PyTorch Profiler [42]. Fig. 12 shows the breakdown results corresponding to experiments related to GCN in Fig. 11 (We also observe the similar treads for GraphSAGE). In this clustered bar figure, from left to right, each bar cluster presents the result of DGL, DGL+PP, PaGraph and PaGraph+PP, respectively. Consistent with the results presented in Section 2, DGL using the built-in data loader faces a severe loading problem. Although preprocessing saves both computation and data loading, it still suffers from the data loading bottleneck, which occupies up to 61 percent of training time.

In contrast to the baseline data loader, thank to the joint effort of caching and pipelining, PaGraph completely overlaps the data loading phase and the GPU computation phase, thus removing the loading cost from the training pipeline, for the original GNN model and its variant with preprocessing enabled. We find that caching stand-alone is sufficient for training the target model over the first three datasets. This is because these datasets are small and can be fully cached into GPU memory, and the data loading time becomes negligible. Unlike this, although the data loading time has already been reduced from 65.7–78.7 percent for the second three more massive datasets by caching, it still accounts for up to 34.9 percent of the whole training time. In this case, pipelining data loading and computation could completely hide the cost of the former case into the latter one. Interestingly, the computation time is slightly reduced in some cases like "PaGraph+PP" for lj-large dataset, since we carefully avoid CPU contentions among parallel training jobs. However, the computation time is slightly increased in some cases like "PaGraph" for enwiki dataset. This is because the background pre-fetching process spends some CPU cycles, which may influence the GPU kernel launching.

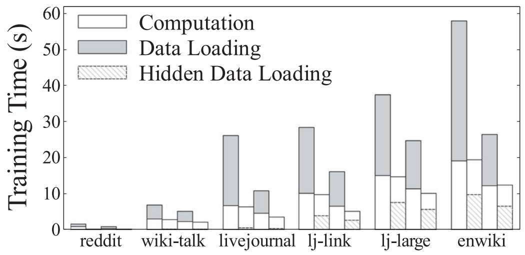


Fig. 12. Breakdown of GCN training time on single GPU. Each bar cluster from left to right represents DGL, PaGraph, DGL+PP and PaGraph+PP. "Hidden Data Loading" corresponds to the part of data loading time that is fully overlapped with computation in PaGraph.

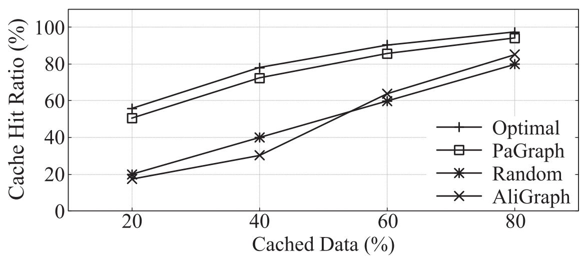


Fig. 13. Cache policy comparison with different cache capacity. "Optimal" represents the ideal cache hit ratio. (Dataset: livejournal).

5.3 Effectiveness of Caching Policy

Next, we compare our static cache policy with the policy presented in AliGraph [43], which supports GNN training across multiple CPU machines. It reduces communication costs between training tasks and the remote storage system by caching a vertex at local if its in-degree to out-degree ratio exceeds a threshold. We also compare to the random strategy, which randomly keeps vertices inside Loader. To help understand how different cache policies perform, we derive the best cache hit ratio which can be obtained if all subsequent vertex visits can be absorbed in cache. We denote this best cache hit ratio as "optimal". The derivation is done by analyzing the visiting trace of training. We did not directly compare PaGraph to AliGraph, since the open-source version of AliGraph did not include the cache code, and it was built atop of TensorFlow other than PyTorch and designed for CPU machines. To make a fair comparison, we follow the description in its paper to implement the AliGraph caching policy in PaGraph.

Fig. 13 shows the cache hit ratio under different cached ratio using a single GPU, respectively. We observe that when only 20 percent of graph are cached, we can achieve more than 50 percent hit ratio, which is more than 200 percent of the performance of other polices. More interestingly, it shows that our caching policy is not complicated, but is incredibly effective, very close to the optimal case. We also achieve the similar close-optimal cache hit ratio on other datasets. Furthermore, we explore the performance implications of the different cache hit ratios achieved by both PaGraph and AliGraph. As shown in Fig. 14, as the proportion of cached graph data increases, the per-epoch training time achieved by both DGL and PaGraph keeps declining

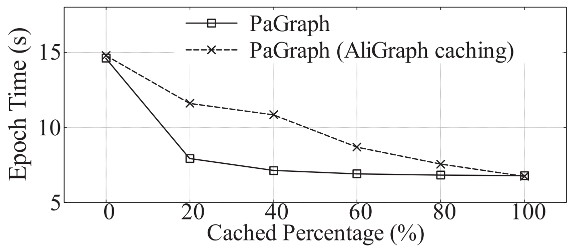


Fig. 14. Training performance of PaGraph at different cached percentage. (Dataset: livejournal).

and converges to 6.7 seconds when all required data is cached. However, when the cache size is constrained, PaGraph's caching policy achieves up to 1.5× performance speedup than the AliGraph's. More interestingly, when the cached percentage reaches 40 percent, the training performance becomes stable, and no further improvements are observed when more cache space is used. This is because at that critical point, the data loading time is reduced to be shorter than the GPU computation time, and our pipeline mechanism could completely overlap the two phases. This further reveals an additional benefit of combining both caching and pipelining, which makes our solution applicable and deliver good performance under GPU memory constraints.

With our caching policy, PaGraph achieve massive data loading reductions on large graphs, compared to DGL. Table 4 shows the average reduced data loading volume of each epoch in each Trainer during GCN training. Overall, we achieve 91.8, 80.9 and 81.0 percent loading reductions for lj-link, lj-large and enwiki datasets, respectively.

5.4 Multi-GPU Performance

We evaluate the scalability with different number of GPUs. Fig. 15 shows the experimental results of training GCN and GraphSAGE against a real-world dataset enwiki with a different number of GPUs. Both DGL and PaGraph achieves higher throughput numbers when more GPUs are in-use. However, in general, PaGraph out-performs DGL and shows better scalability w.r.t different hardware configurations, e.g., PaGraph outperforms DGL by up to 2.4× (4 GPUs for GCN). We further evaluate the performance of PaGraph on multiple GPUs to show its effectiveness on different algorithm optimizations. For this purpose, we zoom in on the performance numbers when using 2 GPUs and summarize the results in Fig. 16. PaGraph achieves the performance of 2.1× to 5.6× over DGL across all datasets. With preprocessing optimization, PaGraph can achieve speedups from 2.8× to 8.5× over DGL. The reasons for the notable speedups are two-fold. First, multiple GPUs provide more available memory for caching, thus can achieve a higher cache hit ratio and lower data loading cost.

TABLE 4
Average Loaded Data Volume in One Epoch of DGL and PaGraph on One GPU

Dataset	DGL(GB)	PaGraph(GB)	PaGraph Cached
lj-link	46.17	3.80 (91.8%↓)	81%
lj-large	46.82	8.94 (80.9%↓)	58%
enwiki	65.38	12.42 (81.0%↓)	54%

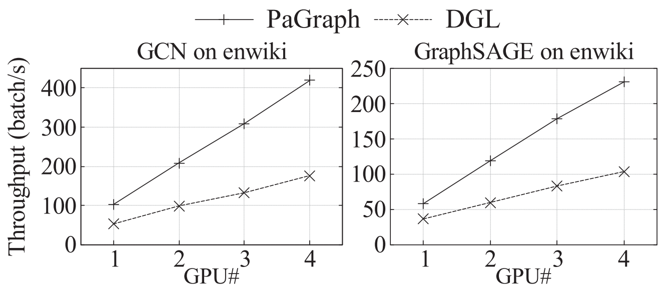


Fig. 15. GCN (left) and GraphSAGE (right) scalability of PaGraph and DGL on enwiki dataset. (Preprocessing enabled).

To confirm this, we also test the performance of GCN on enwiki with a total cache size fixed 6 GB on four GPUs. It shows that the speedup on 4-GPU is only 3.7×, 23 percent lower than the speedup achieved without the limitation of cache size. Second, in all tested cases, data loading runs faster than computation, therefore, our pipeline mechanism completely hides it into the computation.

5.5 Implications of Graph Partition

To validate the benefit of data parallel training enabled by graph partition, we first adapt DGL to use our caching mechanism, denoted as "DGL+Cache," and then compare PaGraph with DGL+Cache, where the only difference is that only the former uses partitioned graph data while the latter uses non-partitioned full graph. Fig. 17a illustrates the cache hit ratio under different number of GPUs. Since Trainers in DGL+Cache shares the same global Graph Store, which maintains the full graph structure, the cached part inside the GPU cache accessed by each Trainer is the same. However, within PaGraph, as the number of Trainer goes up, each Trainer consumes vertices sampled from a smaller set of train vertices, and exhibits better data locality. Therefore, the cache hit ratio achieved by PaGraph keeps increasing when more GPUs are in-use, while DGL+Cache observes a decreasing cache hit ratio. This difference can be directly translated into performance gaps shown in Fig. 18 err (4 GPUs used). By combining caching with partitioning,

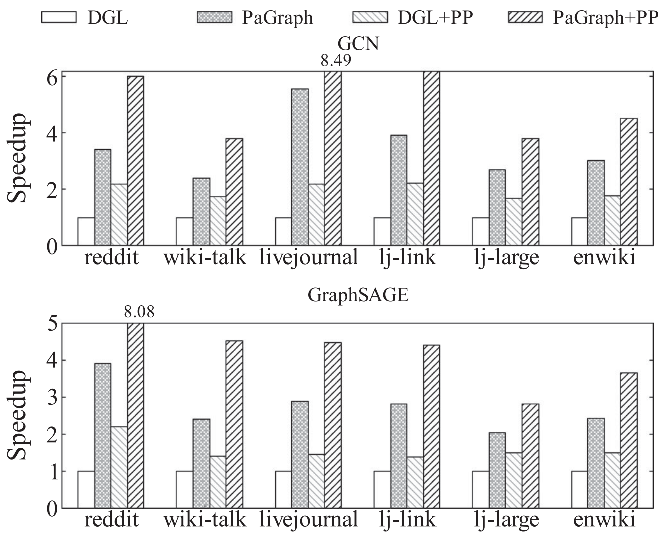
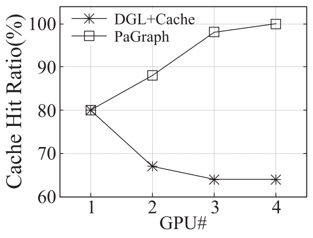
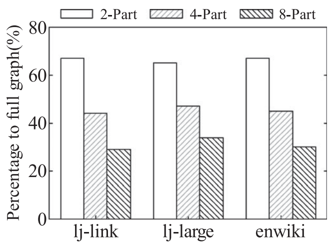


Fig. 16. Training speedup with different optimizations of GCN and GraphSAGE on different datasets (2 GPUs used).



(a) Cache hit comparison.



(b) Vertex proportion.

Fig. 17. (a) Implications of graph partition in terms of cache hit ratio on dataset enwiki, and (b) the proportion of vertices in each partition on average, compared to the full graph.

PaGraph outperforms DGL+Cache and DGL by 25 and 52 percent, respectively. The pipeline mechanism could further improve training speed by 27 percent for GCN.

However, the improvement of graph partitioning comes at prices, namely, redundancy across partitions and offline time cost. Next, we quantitatively analyze their impacts. Fig. 17b shows the proportion of vertices in a single partition to the whole graph, when the number of partitions varies. The y-axis presents the proportion occupied by each partition relative to the full graph. As the partition number increases, the vertex number of each partition decreases. With our algorithm, we bring a small proportion of redundant vertices to each partition in an 8-partition configuration, accounting for 2.5 to 21.5 percent of full graph size. This cost is still affordable, given the training speedups achieved using partitions. Second, Table 5 shows the overall partition cost of four larger datasets out of the total 6, ranging from 6.0 to 32.2 minutes. We believe this is acceptable, since partition is a one-time task performed offline and its cost could be amortized among multiple epochs and repeated training for hyper-parameter tuning [44].

5.6 Effects of Pipelining

Next, we shift our attention to understanding the effects of pipelining data loading and computation. Here, we choose the *friendster* dataset, which is much larger than the other six datasets in Table 2. This large dataset consumes 358 GB CPU memory and only 25 percent of its vertex features can be cached, even when 4 GPUs are in-use. Thus, with this dataset, we can isolate the performance benefits of pipelining from caching. To this end, we train GCN and GraphSAGE by incrementally adding each optimization presented in Section 3, and report the training speed numbers in Fig. 18. Here, we use the neighbor sampling method and enable preprocessing.

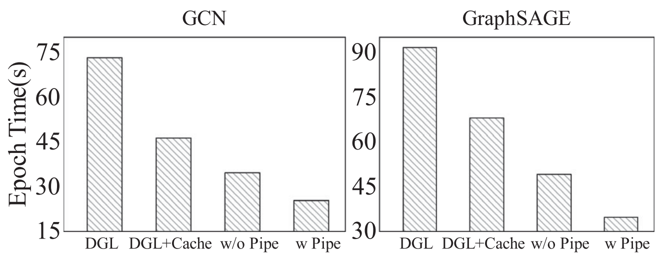


Fig. 18. Training performance improvement when incrementally enabling different optimizations for GCN and GraphSAGE with Neighbor Sampling and the friendster dataset.

TABLE 5
Partition Time Cost

Dataset	livejournal	lj-link	lj-large	enwiki
Time (min)	6.0	8.1	17.1	32.2

Compared to the vanilla DGL, "DGL+Cache" achieves a speedup of 1.4× and 1.3× for GCN and GraphSAGE, due to the caching mechanism. In addition to caching, enabling graph partition improves the performance by 25.1 and 28.1 percent for the two GNN models, respectively. However, in this case, combining caching and partitioning together is not sufficient to fully explore the training acceleration opportunities, since the reduced but not eliminated data loading cost still places negative impacts on the overall performance. Therefore, finally, switching on pipelining could further reduce the per-epoch training time by 27.3 and 29.3 percent for GCN and GraphSAGE, respectively.

5.7 Impact of Data Loading Volumes

To verify the robustness of PaGraph, we evaluate the performance speedup w.r.t the amount of graph data required by each training iteration. Fig. 19 shows training performance with different number of sampled neighbors on DGL, DGL+PP, PaGraph and PaGraph+PP. As the number of sampled neighbors increases, GNN computation will consume more GPU memory, e.g., raising from 1 GB to 5 GB when varying the neighbor size from 2 to 16 in GCN, leading to lower cache capacity. This might limit the performance improvements brought by PaGraph due to fewer cached graph features. However, as shown in Fig. 19, PaGraph constantly outperforms DGL, and its variant combined with preprocessing can achieve even magnitude speedups (e.g., 27.6× of livejournal and 29.2× of lj-link at 16 sampled neighbors). There are three reasons for the substantial performance gains of PaGraph when facing larger neighbor sizes. First, for the DGL baseline, the number of sampled neighbors larger, the imbalance between data loading and computation more severe. Second, despite of its reduced space, the on-GPU

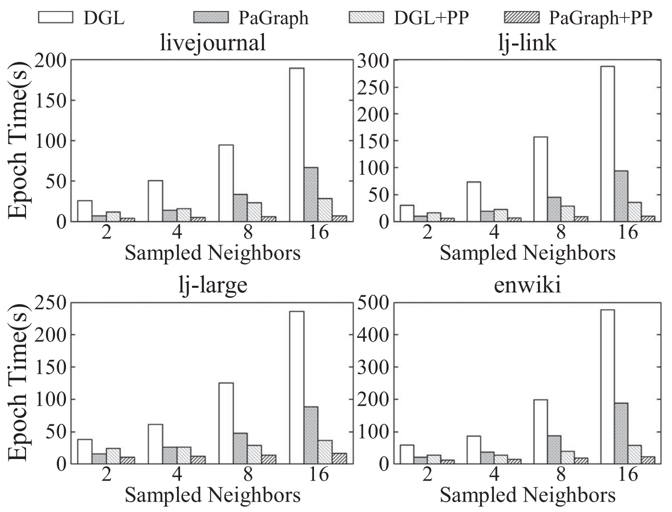
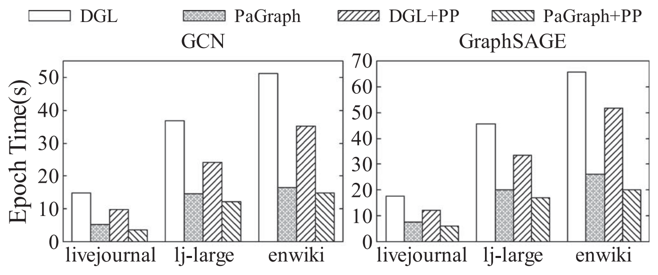
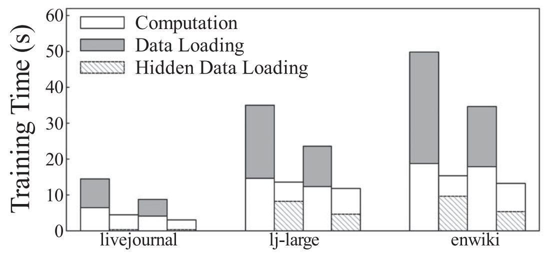


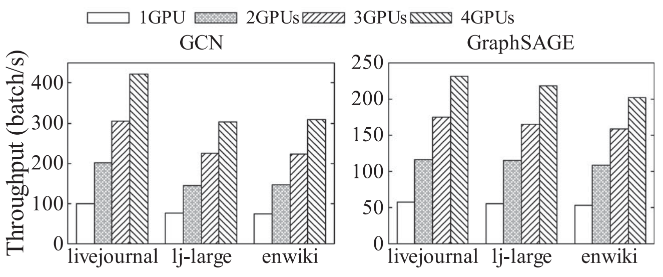
Fig. 19. Training performance of GCN with different numbers of sampled neighbors.



(a) Training performance of GCN and GraphSAGE on single GPU. (PP: preprocessing optimization)

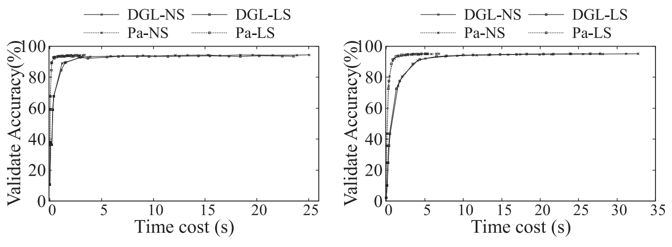


(b) Breakdown of GCN training time on single GPU. Each bar cluster from left to right represents DGL, PaGraph, DGL+PP, PaGraph+PP, respectively. **"Hidden Data Loading" corresponds to the data loading time cost of PaGraph that is fully overlapped with the GNN computation.**



(c) Scalability of PaGraph w.r.t different number of GPUs

Fig. 20. Performance evaluation with layer-wise sampling.



(a) GCN accuracy. (b) GraphSAGE accuracy.

Fig. 21. Validation accuracy and convergence time of PaGraph and DGL during 4-GPU training. (Dataset: Reddit).

cache still significantly shrinks the data loading volumes, as the cache policy prefers to keep vertices with higher out-degrees. Third, pipelining could compensate the performance improvement loss of having smaller cache, as demonstrated by the evaluation of the cache effectiveness (See Fig. 14 and Section 5.3).

5.8 Supporting Other Sampling Method

To validate the effectiveness of PaGraph's designs for other sampling methods, we choose the layer-wise sampling method as another case study, which differs from the neighbor sampling by fixing the size of each layer to avoid exponentially growing of receptive field. Fig. 20a shows the training performance of GCN and GraphSAGE on three datasets using a single GPU. Without preprocessing, PaGraph achieves training speedups from 2.5×(lj-large) to 3.1×(enwiki) for GCN and from 2.3×(lj-large) to 2.5×(enwiki) for GraphSAGE. When the preprocessing is enabled, PaGraph+PP improves the training speed of DGL +PP by 1.9× to 2.7× and 1.9× to 2.6× for GCN and GraphSAGE, respectively. These results imply that PaGraph is also applicable to other sampling method and bring similar performance speedups. We further understand the contributions to the training acceleration by projecting the breakdown time cost in Fig. 20b. We observe that PaGraph reduces the data loading time by up to 93.5 percent over DGL and PaGraph+PP reduces by up to 91.3 percent over DGL+PP. Meanwhile, the data loading time can be hidden by computation, which will further reduce the training time by up to 34.1 percent. Finally, we look into the scalability of GNN training with PaGraph. Fig. 20c shows that similar to the neighbor sampling method evaluated previously, PaGraph can also achieve linear scaling for training GCN and GraphSAGE on three datasets across up to 4 GPUs, with the layer-wise sampling.

5.9 Training Convergence

To confirm the correctness of our implementation as well as the shuffling impact, we evaluate the validation accuracy of training the two exercised GNN models with Neighbor Sampling and Layer-wise Sampling (NS) atop PaGraph (preprocessing enabled) and the DGL library over the reddit dataset on 4 GPUs. DGL adopts global shuffling while PaGraph adopts local shuffling. As shown in Fig. 21a, no matter which sampling method is used, on the GCN model, PaGraph converges to approximately the same accuracy as the original DGL but outperforms DGL by 7.4× (NS) and 8.5× (LS) on convergence speed. This is because comparing with DGL, PaGraph takes the same number of iterations to achieve the same accuracy, but with faster iteration speed. We can get a similar conclusion for GraphSAGE model according to Fig. 21b. We did not conduct similar experiments for datasets other than Reddit because features of vertices in those datasets are randomly initialized.

6 Related Work

Frameworks for GNN Training. Deep learning frameworks like PyTorch [23], MXNet [25] and TensorFlow [9] have been widely adopted in both academia and industry. However, these frameworks don't provide enough graph operations required for GNN. Thus, in recent years, it drives the birth of a few specialized frameworks [10], [11], [13], [43], [45], [46], [47], [48] designed for handling graph neural networks. They all borrow and extend traditional graph processing primitives and vertex-programming abstractions into current deep learning frameworks. For instance, NeuGraph [13] develops graph operation primitives by Scatter-ApplyEdge-Gather-ApplyVertex (SAGA) to provide efficient and convenient programming models.

Full Graph versus Sampling Training. The full graph training trains graph neural network model with whole graph data in every forward-backward propagation, while the sampling training only trains part of vertices and edges in every forward-backward propagation. To deal with the large graphs that can not be fully filled into GPU, PBG [46] and NeuGraph [13] split the full graph into chunks and iteratively loads each chunk and its vertex data into CPU and GPU for full graph computation, respectively. While for sampling training, DGL [10] locates the full graph and its data in the CPU shared memory and only loads the required vertex and edge data into GPU memory at the beginning of every forward-backward propagation. For large graphs, recent work [19] already shows the sampling training can achieve faster convergence several times than full graph training with similar final model performance. AliGraph is a platform supporting sampling-based GNN training on CPU platforms, rather than exploring the potentials of GPUs [43].

Caching. Caching graph data benefits many graph processing tasks [43], [48], [49]. Pre-select [49] applies a static caching policy to speed up BFS-like computations. AliGraph [43] caches vertices and their data for distributed computing scenarios to avoid communication costs between trainer and remote storage. However, as we have verified in Section 5.3, its caching policy is not suitable for neighbor-based sampling characteristics. ROC [48] targets full graph training and explores an efficient strategy for swapping the intermediate results of GNN model between CPU and GPU by leveraging partition and caching. We differ from ROC as handling intermediate results is not the primary concern under sampling-based GNN training.

Graph Partition. Partitioning graphs [8], [20], [21], [50], [51], [52] is widely adopted in distributed computations. PowerGraph [8] designs a vertex-cut partition algorithm for power-law graphs. NeuGraph [13] leverages Kernighan-Lin [21] to partition graphs into chunks with different sparsity level. [47] further proposes a GNN-specific graph partition algorithm to reduce the communication overhead among multiple machines. All the partition algorithms mentioned above are designed for non-GNN tasks or full graph GNN training. Our partition algorithm is designed for sampling training, and cooperated with caching.

Other Optimizations. Gorder [53] designs a general graph re-ordering method to accelerate graph processing tasks. Norder [49] proposes a tailored re-ordering technique for BFS-like tasks. We have implemented these techniques in PaGraph but found there is no significant improvement in either partition quality or training performance.

7 Discussion and Future Work

Though we believe PaGraph can benefit a wide range of GNN models and systems, there are still some open challenges need to address before making it more general and applicable to even wider context.

Data Loading and Computation Ratio. The caching strategy, as the core of PaGraph, works well when the data loading phase dominates the whole training process. However, its relative ratio is model specific. In addition to GCN and GraphSAGE, the two representative and successful GNN models widely adopted, we have examined other models such as GAT [4], GIN [54] and diffpool [55]. The preliminary conclusion here is that comparing with GAT and GIN, which look similar to GCN and GraphSAGE, diffpool is more computational-intensive. Therefore, it is expected that PaGraph benefits less diffpool than the other four models. Nevertheless, the witnessing emergence of GNN computation optimizations [56] can further stress the imbalance between data loading and computation, and make PaGraph remain a valid design for future GNN models.

Different Sampling Patterns and Evolving Graphs. Clearly, the graph traversal patterns carried out by sampling methods play a key role in determining good cache efficiencies. The performance improvements of the current PaGraph implementation would be limited if the corresponding sampling method did not assume that graph vertices with high out-degrees are more likely to be selected. Furthermore, PaGraph adopts static caching and partitioning strategies, and consequently cannot handle sampling methods assuming features that are trainable or changing across iterations, e.g., control-variate [19]. Finally, although most of existing work focus on static graphs, training GNN models over dynamic graphs is attracting increasing interests [57], [58]. As the vertex out-degrees and the graph skewness rapidly change alongside the graph evolution, it is desirable to design a dynamic caching and partitioning strategy with sustained caching efficiencies and moderate cost, as well as tunable pipeline for complementing caching, whose performance is bounded by data locality and the cached graph portion.

GPU-Centric Methods. The data loading efficiency problem that PaGraph targets is highly related to the current CPU-GPU co-processing architecture, widely used in the DNN community. Most recently, there have been some initial efforts to explore GPU-centric optimizations for eliminating such problem, for instance, on-GPU sampling [59]. These methods could deliver the highest performance acceleration for small graphs, whose features can be fitting into GPU memory entirely. In this case, not only the cumbersome CPU-to-GPU data loading phase can be eliminated, but also the high processing density and memory bandwidth of GPU can be leveraged for fast sampling. However, it remains challenging for larger graphs, for which the CPU-involving out-of-core data access for GPU and limited PCIe bandwidth may offset the benefits of on-GPU sampling. Fortunately, the new GPUDirect Storage technology [60] offers a direct data path between fast NVMe storage and GPU memory, and experimentally demonstrates the increased data loading performance by bypassing CPU. Therefore, it is worthy considering the design points of PaGraph in the context of hardware innovations.

8 Conclusion

To accelerate GNN training performance on large graphs, we present PaGraph, a general sampling-based training scheme that leverages the combination of GNN computation-aware caching and graph partition. We implement PaGraph on DGL and PyTorch, and evaluate it with 2 representative GNN models over 7 graph datasets, using two widely used sampling methods. Experimental results show that PaGraph completely hides data transfer costs and reaches up to 4.8× performance improvements, comparing to DGL. With preprocessing, PaGraph can even achieve a 16.0× performance speedup.

Acknowledgments

The authors sincerely thank all anonymous reviewers for their insightful feedback. This work was supported in part by the National Key R&D Program of China under Grant No. 2018YFB1003204, in part by the National Nature Science Foundation of China under Grant 61 802 358 and Grant 61 772 484, in part by the "USTC Research Funds of the Double First-Class Initiative" under Grant YD2150002006, and in part by the Huawei and Microsoft.

References

[1] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun, "Graph neural networks: A review of methods and applications," 2018, *arXiv:1812.08434.*

[2] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. Advances Neural Inf. Process. Syst.*, 2017, pp. 1024–1034.

[3] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. Int. Conf. Learn. Representations*, 2017.

[4] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *Int. Conf. Learn. Representations*, 2018.

[5] V. Garcia and J. Bruna, "Few-shot learning with graph neural networks," in *Int. Conf. Learn. Representations*, 2018.

[6] Y. Zhang, P. Qi, and C. D. Manning, "Graph convolution over pruned dependency trees improves relation extraction," in *Proc. Conf. Empir. Methods Nat. Lang. Process.*, 2018, pp. 2205–2215.

[7] J. Bastings, I. Titov, W. Aziz, D. Marcheggiani, and K. Simaan, "Graph convolutional encoders for syntax-aware neural machine translation," in *Proc. Conf. Empir. Methods Nat. Lang. Process.*, 2017, pp. 1957–1967.

[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 17–30.

[9] M. Abadi *et al.*, "TensorFlow: A. system for large-scale machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[10] M. Wang *et al.*, "Deep graph library: Towards efficient and scalable deep learning on graphs," in *Proc. ICLR Workshop Representation Learn. Graphs Manifolds*, 2019.

[11] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch geometric," in *Proc. ICLR Workshop Representation Learn. Graphs Manifolds*, 2019.

[12] MindSporeTeam, "Welcome to the model zoo for MindSpore," Accessed: Sep.2020. [Online]. Available: https://github.com/mindspore-ai/mindspore/tree/master/model_zoo/official, 2020

[13] L. Ma *et al.*, "NeuGraph: Parallel deep neural network computation on large graphs," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 443–458.

[14] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. 13th Int. World Wide Web Conf.*, 2004, pp. 595–601.

[15] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast learning with graph convolutional networks via importance sampling," in *Proc. Int. Conf. Learn. Representations*, 2018.

[16] W. Huang, T. Zhang, Y. Rong, and J. Huang, "Adaptive sampling towards fast graph representation learning," in *Proc. Advances Neural Inf. Process. Syst.*, 2018, pp. 4558–4567.

[17] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, "Collective classification in network data," *AI Mag.*, vol. 29, no. 3, pp. 93–93, 2008.

[18] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowl. Inf. Syst.*, vol. 42, no. 1, pp. 181–213, 2015.

[19] J. Chen, J. Zhu, and L. Song, "Stochastic training of graph convolutional networks with variance reduction," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 941–949.

[20] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov, "Streaming graph partitioning: An experimental study," *Proc. VLDB Endowment*, vol. 11, 2018, pp. 1590–1603.

[21] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, no. 2, pp. 291–307, 1970.

[22] KONECT, "Wikipedia links, English network dataset - KONECT," Apr. 2017. [Online]. Available: http://konect.uni-koblenz.de/networks/wikipedia_link_en

[23] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Advances Neural Inf. Process. Syst.*, 2019, pp. 8024–8035.

[24] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. data*, 2010, pp. 135–146.

[25] T. Chen *et al.*, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," in *Proc. Neural Inf. Process. Syst., Workshop Mach. Learn. Syst.*, 2015.

[26] DGLTeam, "DGL large-scale trainingtutorial," 2019, Accessed: Jan. 2020. [Online]. Available: https://docs.dgl.ai/tutorials/models/5_giant_graph/2_giant.html,

[27] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proc. 7th ACM SIGCOMM Internet Meas. Conf.*, 2007, pp. 29–42.

[28] M. Chrobak and J. Noga, "LRU is better than FIFO," *Algorithmica*, vol. 23, no. 2, pp. 180–185, 1999.

[29] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *Proc. Conf. Appl. Technol. Architectures, Protocols Comput. Commun.*, 1999, pp. 251–262.

[30] M. E. Newman, "Power laws, pareto distributions and zipf's law," *Contemp. Phys.*, vol. 46, no. 5, pp. 323–351, 2005.

[31] J. M. Pujol *et al.*, "The little engine (s) that could: Scaling online social networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 375–386, 2011.

[32] NVIDIA Corporation, "NVIDIA collective communications library (NCCL)," 2019, Accessed: Dec. 20, 2019. [Online]. Available: https://developer.nvidia.com/nccl

[33] SciPy Team, "SciPy: Sparse matrix CSC format," 2008, Accessed: May 2020. [Online]. Available: http://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csc_matrix.html

[34] A. Kakaraparthy, A. Venkatesh, A. Phanishayee, and S. Venkataraman, "The case for unifying data loading in machine learning clusters," in *Proc. 11th USENIX Conf. Hot Top. Cloud Comput.*, 2019, Art. no. 12.

[35] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Efficient hyperparameter optimization and infinitely many armed bandits," *CoRR*, 2016.

[36] OpenMP Team, "The OpenMP API specification for parallel programming," 2012, Accessed: May 2020. [Online]. Available: https://www.openmp.org

[37] B. Recht and C. Ré, "Toward a noncommutative arithmetic-geometric mean inequality: Conjectures, case-studies, and consequences," in *Proc. 25th Annu. Conf. Learn. Theory*, 2012, pp. 11.1–11.24.

[38] Q. Meng, W. Chen, Y. Wang, Z.-M. Ma, and T.-Y. Liu, "Convergence analysis of distributed stochastic gradient descent with shuffling," in *Proc. Advances Neural Inf. Process. Syst.*, 2017.

[39] J. Leskovec, D. Huttenlocher, and J. Kleinberg, "Predicting positive and negative links in online social networks," in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 641–650.

[40] KONECT, "LiveJournal links network dataset – KONECT," Apr. 2017. [Online]. Available: http://konect.uni-koblenz.de/networks/livejournal-links

[41] NVIDIA corporation, "NVIDIA NVPROF," 2007, Accessed: Jan. 2020. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[42] PyTorch team, "PyTorch profiler," 2017, Accessed: May 2020. [Online]. Available: https://pytorch.org/tutorials/recipes/recipes/profiler.html

[43] R. Zhu *et al.*, "AliGraph: A comprehensive graph neural network platform," *Proc. VLDB Endowment*, vol. 12, pp. 2094–2105, Aug. 2019. [Online]. Available: https://doi.org/10.14778/3352063.3352127

[44] W. Xiao *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 595–610.

[45] P. W. Battaglia *et al.*, "Relational inductive biases, deep learning, and graph networks," 2018, *arXiv:1806.01261*.

[46] A. Lerer *et al.*, "PyTorch-BigGraph: A large-scale graph embedding system," in *Proc. Mach. Learn. Syst.*, 2019, pp. 120–131.

[47] A. Tripathy, K. Yelick, and A. Buluc, “Reducing communication in graph neural network training,” 2020, *arXiv:2005.03300*.

[48] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, “Improving the accuracy, scalability, and performance of graph neural networks with ROC,” in *Proc. Mach. Learn. Syst.*, 2020, pp. 187–198.

[49] E. Lee, J. Kim, K. Lim, S. H. Noh, and J. Seo, “Pre-select static caching and neighborhood ordering for BFS-like algorithms on disk-based graph engines,” in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 459–474.

[50] C. Mayer, M. A. Tariq, R. Mayer, and K. Rothermel, “GrapH: Traffic-aware graph processing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 6, pp. 1289–1302, Jun. 2018.

[51] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998.

[52] S. Verma, L. M. Leslie, Y. Shin, and I. Gupta, “An experimental comparison of partitioning strategies in distributed graph processing,” *Proc. VLDB Endowment*, vol. 10, pp. 493–504, 2017.

[53] H. Wei, J. X. Yu, C. Lu, and X. Lin, “Speedup graph processing by graph ordering,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 1813–1828.

[54] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, “How powerful are graph neural networks?” 2018, *arXiv:1810.00826*.

[55] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec, “Hierarchical graph representation learning with differentiable pooling,” 2018, *arXiv:1806.08804*.

[56] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen, “Understanding and bridging the gaps in current GNN performance optimizations,” in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 119–132.

[57] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, “Dynamic graph CNN for learning on point clouds,” *ACM Trans. Graph.*, vol. 38, no. 5, pp. 1–12, 2019.

[58] F. Manessi, A. Rozza, and M. Manzo, “Dynamic graph convolutional networks,” *Pattern Recognit.*, vol. 97, 2020, Art. no. 107000.

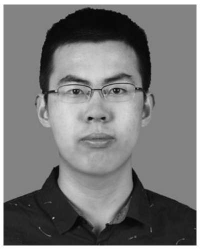
[59] S. Pandey, L. Li, A. Hoisie, X. S. Li, and H. Liu, “C-SAW: A Framework for graph sampling and random walk on GPUs,” 2020, *arXiv:2009.09103*.

[60] NVIDIA, “GPUDirect storage: A direct path between storage and GPU memory,” 2019, Accessed: Feb. 2021. [Online]. Available: https://developer.nvidia.com/blog/gpudirect-storage/

Youhui Bai received the BS degree from the Department of Computer Science, University of Science and Technology of China (USTC), Hefei, China, in 2016. He is currently working toward the PhD degree with Advanced Data Systems Laboratory, University of Science and Technology of China, China. His research interests include distributed AI systems, graph processing and storage systems.

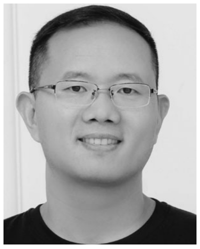
Cheng Li (Member, IEEE) received the PhD degree from the Saarland University/Max Planck Institute for Software Systems, Germany, in 2016. He has been a pretenure professor with the Department of Computer Science and Technology, University of Science and Technology of China, China since Fall 2017. His research interests include various topics related to improving performance, consistency, fault tolerance, and availability of distributed systems. He is a member of ACM, USENIX, and CCF.

Zhiqi Lin received the BS degree from the Department of Computer Science, University of Science and Technology of China, China, in 2019. He is currently working toward the PhD degree with Computer Science Department, University of Science and Technology of China, China, and joined the joint-PhD program in Microsoft Research Asia. His current interest includes distributed AI system.



Yufei Wu is currently working toward the undergraduate degree with the Department of Computer Science, University of Science and Technology of China (USTC), Hefei, China. He is working at Advanced Data System Laboratory, University of Science and Technology of China, China. His research interest includes distributed AI systems.

Youshan Miao received the BS and PhD degrees from the University of Science and Technology of China (USTC), Hefei, China, in 2009 and 2015, respectively. His research interests include distributed systems, AI systems, graph storage, and computing. He is currently a researcher with System Research Group, Microsoft Research Asia (MSRA).



Yunxin Liu (Senior Member, IEEE) received the BS, MS, and PhD degrees from the University of Science and Technology of China, Tsinghua University, China, and Shanghai Jiao Tong University, China, respectively. He is a principal researcher with Microsoft Research, Asia. His current research interests are focused on mobile and edge computing. He received MobiCom 2015 Best Demo Award, PhoneSense 2011 Best Paper Award, and SenSys 2018 Best Paper Runner-up Award.

Yinlong Xu (Member, IEEE) received the BS degree in mathematics from Peking University, Beijing, China, in 1983, and the MS and PhD degrees in computer science from the University of Science and Technology of China (USTC), Hefei, China, in 1989 and 2004, respectively. He is currently a professor with the School of Computer Science and Technology, University of Science and Technology of China, China. He served the Department of Computer Science and Technology, University of Science and Technology of China, China as an assistant professor, a lecturer, and an associate professor. He is currently leading a group in doing some networking, storage, and high performance computing research. His current research interests include storage system, file system, social network, and high performance I/O. He was a recipient of the Excellent PhD Advisor Award of the Chinese Academy of Sciences in 2006 and Baosteel Excellent Teacher Award, in 2014.