PGLBox: Multi-GPU Graph Learning Framework for Web-Scale Recommendation

Xuewu Jiao*
Weibin Li*
Xinxuan Wu*
Wei Hu*
Miao Li
Jiang Bian
Baidu Inc.
Haidian District, Beijing, China

Siming Dai
Xinsheng Luo
Mingqing Hu
Zhengjie Huang
Danlei Feng
Junchao Yang
Shikun Feng
Baidu Inc.
Haidian District, Beijing, China

Haoyi Xiong
Dianhai Yu
Shuanglong Li
Jingzhou He
Yanjun Ma
Lin Liu
Baidu Inc.
Haidian District, Beijing, China

ABSTRACT

While having been used widely for large-scale recommendation and online advertising, the Graph Neural Network (GNN) has demonstrated its representation learning capacity to extract embeddings of nodes and edges through passing, transforming, and aggregating information over the graph. In this work, we propose PGLBox[1] – a multi-GPU graph learning framework based on PaddlePaddle [24], incorporating with optimized storage, computation, and communication strategies, to train deep GNNs based on web-scale graphs for the recommendation. Specifically, PGLBox adopts a hierarchical storage system with three layers to facilitate I/O, where graphs and embeddings are stored in the HBMs and SSDs, respectively, with MEMs as the cache. To fully utilize multi-GPUs and I/O bandwidth, PGLBox proposes an asynchronous pipeline with three stages – it first samples the subgraphs from the input graph, then pulls & updates embeddings and trains GNNs on the subgraph with parameters updating queued at the end of the pipeline. Thanks to the capacity of PGLBox in handling web-scale graphs, it becomes feasible to unify the view of GNN-based recommendation tasks for multiple advertising verticals and fuse all these graphs into a unified yet huge one. We evaluate PGLBox using a bucket of realistic GNN training tasks for the recommendation, and compare the performance of PGLBox on top of a multi-GPU server (Tesla A100×8) and the legacy training system based on a 40-node MPI cluster at Baidu. The overall comparisons show that PGLBox could save up to 55% monetary cost for training GNN models, and achieve up to 14× training speedup with the same accuracy as the legacy trainer. The open-source implementation of PGLBox is available at https://github.com/PaddlePaddle/PGL/tree/main/apps/PGLBox.

*These authors contributed equally to this work.
[1]This work was supported in part by National Key R&D Progam of China (No. 2021ZD0110303).

CCS CONCEPTS

• **Mathematics of computing → Graph algorithms**.

KEYWORDS

Graph learning; GNN; GPU graph engine; Hierarchical storage

ACM Reference Format:
Xuewu Jiao, Weibin Li, Xinxuan Wu, Wei Hu, Miao Li, Jiang Bian, Siming Dai, Xinsheng Luo, Mingqing Hu, Zhengjie Huang, Danlei Feng, Junchao Yang, Shikun Feng, Haoyi Xiong, Dianhai Yu, Shuanglong Li, Jingzhou He, Yanjun Ma, and Lin Liu. 2023. PGLBox: Multi-GPU Graph Learning Framework for Web-Scale Recommendation. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '23), August 6–10, 2023, Long Beach, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3580305.3599885

1 INTRODUCTION

In the last decades, machine learning techniques have been pushing the frontiers of the development of internet services, such as Ads click-through rate prediction[16] and content understanding[5, 38]. Among the wide range of machine learning models, Deep Neural Networks(DNNs) have been widely used to learn users' preferences from their historical Ad impressions and clicks [3, 36]. Particularly, given the online Ad data, including users, items, and records of purchases or clicks, recommender systems might model the data into web-scale graphs with users/items as nodes and purchases/click-throughs as edges [9, 37]. They further incorporate Graph Neural Networks(GNNs) to capture structural information from the graph and recommend items to users by predicting potential edges accordingly.

In general, given a graph with features attributed to every node, the GNN treats the node or edge features as the input of the neural network while incorporating the graph structure as part of neural layers. For every node/edge, the GNN aggregates its neighbors' information to form the embeddings (low-dimensional vectors) of the node/edge. In this way, GNN is capable of performing node/edge prediction tasks, such as node classification[18], graph classification[1], and predicting missing links between nodes, aka link prediction[39]. Along this line of research, algorithms have been studied extensively to facilitate GNNs training at scale. For example, DeepWallk [29] and Graph Convolutional Neural Networks (GCNs) [4, 18] propose to mimic information propagation over graphs through random walks or convolution and extract features accordingly. Graph Attention Networks (GATs)[32] extends GCNs with self-attention mechanisms, while GraphSage[12] leverages subgraph sampling in mini-batch stochastic training for efficacy.

Though ways to improve GNNs have been studied in previous works, training GNNs over a large-scale graph with millions of nodes and billions of edges is still a challenging task for web-scale recommendation. There are several non-trivial technical issues that should be addressed.

- While the graph is huge, the embeddings of its nodes and edges are all ultra high-dimensional vectors, with millions of sparse and dense features mixed, for the web-scale recommendation. Blessed by the data locality of graph learning, it is possible to cache the key data that is high-frequently used in the calculation. Furthermore, during the GNN training, the graph (or subgraphs) is imported as a read-only constant while embeddings of nodes/edges would be updated in an extremely fast manner. Thus, imbalances between both read/write and graphs/embeddings widely exist. In this way, there needs a way to store data, including graphs and embeddings, in storage devices at different speeds, with respect to the frequency of read and write.
- To carry out large-scale graph learning, a traditional way is to map the computation load over a cluster of high-performance computing nodes [40] on top of the Message Passing Interface (MPI) with shared disks. For every single node, it however suffers significant time costs for data synchronization between CPU and GPU. Researchers even proposed a CPU-only strategy that outperforms a GPU-CPU solution [40] for large-scale training. Thus, to fully utilize the computational power of GPU, there needs a way to parallel CPU-GPU data communications and GPU computations in an asynchronous work-stealing fashion [2].

To this end, we propose PGLBox, a large-scale graph learning framework, based on multi-GPUs with hierarchical storage. As shown in Figure 1, PGLBox loads the graph (nodes with features, edges) from disks, and further stores the sparse representations of the graph and embeddings in SSD and HBM (High-Bandwidth Memory) respectively. It leverages a three-stage asynchronous pipeline to train GNNs, where CPU-GPU communications and GPU computations occur in an asynchronous manner for work-stealing. In addition to system design, a novel metapath split-based sampling strategy has been integrated into PGLBox for efficient subgraph-based learning. In summary, the main contributions of this work are as follows.

- We study the problem of large-scale GNN training for web-scale recommendation tasks, where billions of edges connecting millions of nodes are incorporated. To the best of our knowledge, this study is the first work aiming to accelerate GNN-based recommendation systems by addressing technical issues, including (1) the data access imbalance between both the graph/embeddings and read/write, (2) significant communication overhead caused by GPU-CPU data synchronization for huge graphs, and (3) subgraph-based efficient learning over huge graphs.
- We propose PGLBox based on PaddlePaddle [24] deep learning framework, that addresses the technical issues in a systematic way. To accelerate storage and data access, PGLBox incorporates SSD and HBM to store the sparse representations of graph data and embeddings respectively. To reduce the cost of synchronous, asynchronous pipelining has been adopted in the training algorithm design with three stages. Finally, a modified Fisher-Yates subgraph sampling algorithm has been proposed to lower the costs of every training iteration. Compared to GraphSage [12], the proposed sampling method re-indices the execution order of subgraphs during the training process, leverages a metapath split strategy and the compressed features (variable-length slot features) for further cost reduction.
- Extensive experiments on the real-world data from Baidu's online Advertising systems and an open-sourced benchmark dataset are conducted to evaluate PGLBox. The results show that PGLBox outperforms the legacy MPI cluster in terms of efficiency, where it could save up to 55% monetary cost for training GNN models (with equivalent effectiveness in recommendation) and achieve up to 14× training speedup with the same accuracy as the legacy trainer. We also establish the ablation studies to demonstrate the effectiveness of each design and optimization in PGLBox system.

2 FRAMEWORK DESIGN

We present the overview of PGLBox and introduce its main components from macro perspectives. The architecture of PGLBoxis shown in Figure 1, it contains three parts, which are the CPU module, Sparse table module, and GPU module.

- **The CPU module** – is responsible for coordinating the entire training process. It initializes the graph parameters and loads the nodes and edges from the file system into memory. It then uploads this data to the GPUs and creates mini-batches for parallel processing. After each mini-batch is calculated, the CPU module computes the node and feature embeddings and stores them in the Sparse Table module so they can be referenced later. The CPU also periodically saves the current training parameters as checkpoints to the file system for warm-start GNN training.
- **The Sparse Table** – is a key-value storage system that stores the discrete nodes and features on solid-state drives (SSDs). It also has a key hash index feature-to-file mapping to easily look up keys. To reduce the latency of SSD access, we have designed an explicit in-memory cache strategy, which can act as a buffer to avoid excessive writing to the SSDs.
- **The GPU Module** – has the responsibility of receiving the graph data and constructing the GPU-type graph structure with a node list and neighbor list for each GPU. The sampling and walking are based on this graph data. The sparse embeddings from the CPU module are also received by the GPU module and stored in High-Bandwidth Memory (HBM). Then these embeddings are fed into the GNNs training. Different CUDA streams are created for the sampling and GNNs training. At the end of each mini-batch, all of the GPUs in the PGLBox perform collective communications to synchronize parameters using the NVLink high-speed interconnections.

2.1 Hierarchical Storage

PGLBox stores graph structures, nodes, and sparse embeddings hierarchically to support large-scale graphs of up to 10 billion nodes and tens of billions of edges. Graph structure data contains the link relationships of edges and slot features of the nodes. Sparse

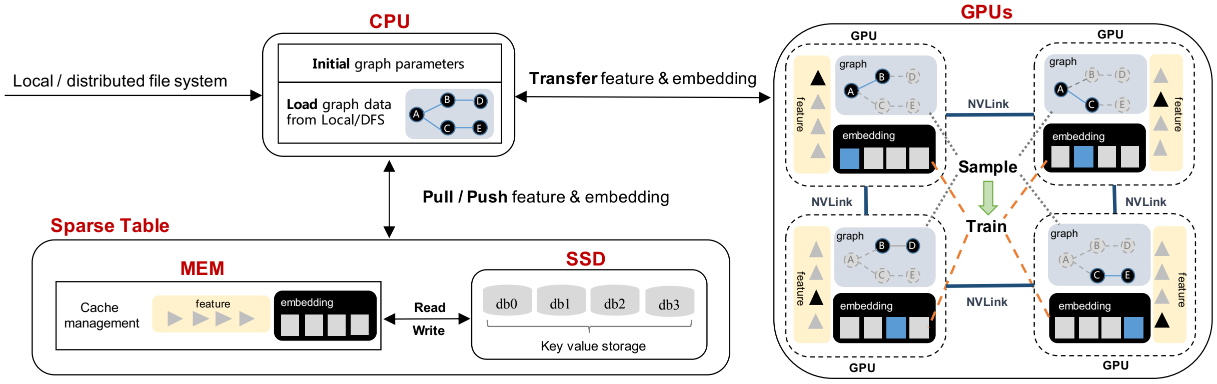


Figure 1: The architecture of PGLBox.

embedding contains node embedding and slot feature embedding. We use four tables to store the graph data and sparse embeddings, including nodeid2neighbor, nodeid2feature, node embedding, and feature embedding. Nodeid2neighbor describes the link relationship of edges, nodeid2feature represents the slot features of nodes, and node embedding and feature embedding describe the nodeID and slot feature embeddings respectively.

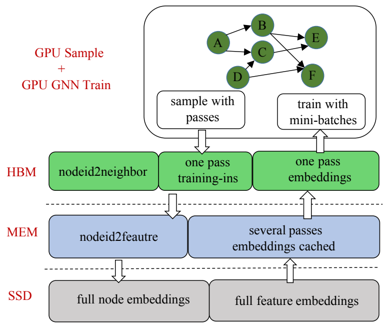


Figure 2: Hierarchical storage.

To make the most efficient use of resources, the storage is divided into High Bandwidth Memory (HBM), Solid State Drives (SSD), and Memory (MEM). As shown in Figure 2, nodeid2neighbor is chosen to be fully stored in HBM to reduce communication overhead between the CPU and GPU. The other three tables are stored in SSD and Memory to accommodate larger-scale graphs. Both node-id embeddings and slot feature embeddings are stored in SSD. Memory acts as a link between the HBM and SSD, caching several passes of sparse embeddings, nodeIDs, and slot features of nodes. Additionally, we introduce the concept of a "pass" in the Graph Neural Network (GNN) training process. The sampling starting points are divided into multiple passes, with a specific number of sampling starting points forming a pass.

HBM is expensive and has limited capacity, so one-pass embedding and generated training instructions are stored in HBM, while embeddings are updated with mini-batches in GNN training. When a pass of nodes has been sampled in GPU, multiple walks are generated starting from these nodes and traverse the walks to form training instructions. As the number of training instructions grows, feed forward and back propagation computations are conducted for the GNNs, and the embeddings in the GPU hash table are updated by the backward gradient. Additionally, the embeddings cached in the CPU are updated when a pass is finished in the GPU. Full embeddings stored in SSDs are updated when several passes of training have finished, which can act as a buffer to avoid frequent writing to SSDs. When the next pass begins, embeddings can be used directly if Memory has cached all the nodes, otherwise, they need to be looked up in SSDs.

Finally, embeddings in SSD, Memory and HBM are pulled and updated based on different periods. The slot feature of nodes is stored with only HBM and memory and is loaded into Memory after loading node files from HDFS. All slot features of current pass nodes are duplicated before the training process begins, and embeddings of nodeID and slot features are obtained together from Memory or SSD. These duplicated embeddings of nodeID and node slot features are then fed to GPU for GNN training. This hierarchical storage makes full use of resources, improving resource utilization, and making it possible for large-scale graphs of up to ten billion nodes and tens of billions of edges to be trained on a single node machine.

2.2 Three-Stage Asynchronous Training Pipeline

The training workflow consists of three time-consuming tasks: sampling neighbors to generate training instances, pulling & updating embeddings from sparse tables, and training with GNNs. These processes require overlapping hardware resources such as GPUs and SSDs. However, the limited GPU resources can greatly affect the training efficiency if they are not used properly. To address this problem, we have developed a three-stage pipeline that includes a worker thread (for the work-stealing) that takes tasks from the prefetch queue and makes use of the corresponding resources in each stage asynchronously. The processed results are then pushed into the next stage. In addition, we use atomic semaphores [19] to control the number of prefetch queues.

Figure 3 shows the pipeline process. In the sampling stage, we have already built graph data with GPU type, and sampled neighbors from a given number of start nodes to form a walk with a preset walk length. This stage can operate depending on the preset atomic semaphore, and once the number of loads exceeds 0, the sample is executed immediately. The Pull&Update Embedding stage consists of two procedures, which are 1) retrieving the referenced parameters in the samples from MEM and SSDs, and 2) updating gradient information (from the training results) to build the model. Thus, the training stage heavily relies on the Pull Embedding stage, as the forward and backward propagation of the GNNs can only be computed after all the embeddings are loaded. Note that we combine the Pull and Update stages in Figure 3 since the Update stage (the parameters of several past passes have been cached in MEM) consumes a significantly shorter time than the Pull stage (the cache rebuilding is required) so that the Update stage with its dependency on the training results could be neglected and hidden in the whole pipeline.

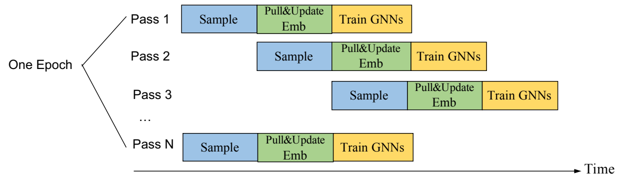


Figure 3: The three-stage pipeline covering sampling, pulling&updating embedding, and training GNNs.

The asynchronous pattern is designed for the purpose that enabling work stealing. Each pass occupies different system resources during various stages, and executing different passes at different time points can maximize heterogeneous resource utilization. Simultaneously, the subsequent pass can pre-execute some stages while the previous pass is being executed. For instance, the second pass can pre-execute sampling while the first pass is pulling parameters, thus reducing the overall task execution time. It is the same as the classic work stealing style, where each processor in a computer system has a queue of work items to perform.

Although sampling and GNN training are all operated in GPUs, PGLBox with multiple high-end GPUs has a comparable computation power to a distributed cluster with dozens of CPU-only nodes. Sampling purely in GPUs is more complicated and thus requires optimization to ensure that the latency of the samples can be overlapped by the pipeline. In the following subsections, we introduce the detailed sampling strategy and the optimization designs.

2.3 Sampling on Multiple GPUs

Sampling Operation. Sampling a subgraph for a set of target nodes and training multi mini-batches in one pass makes the training of GNNs in large-scale graphs possible, as mentioned in Section 1. This process is mainly divided into four steps, as shown in Figure 4: from the input of the entire graph to the generation of subgraphs of target nodes. The first step is to input the entire graph, the second step is to randomly walk through the graph according to the preconfigured metapath to obtain different paths, the third step is to perform multi-stage sampling on the deduplicated nodes to generate the subgraphs, and the fourth step is to generate pairs according to the walk and deduplicate the nodes in the pairs. Here multi- stage is closed to multi-hop neighborhood-based sampling [27], which refers to the number of sampling stages. When performing 2-stage sampling, we first sample the immediate neighbors connected to the central node, and then, use these immediate neighbors as starting points to further sample the second-order neighbors that are directly connected to the first-order neighbors. The second and third steps both involve the sampling of one or more nodes from the candidate neighbors, and two algorithms are used: reservoir sampling[33] and modified Fisher-Yates sampling[7]. The reservoir sampling ensures the neighbor nodes are chosen with equal probabilities when the resource occupation is determined, so it is used to generate random walks. However, it takes a long time to generate the subgraphs due to high complexity. Therefore, the modified Fisher-Yates algorithm is proposed to improve the efficiency in the subgraph generation step.

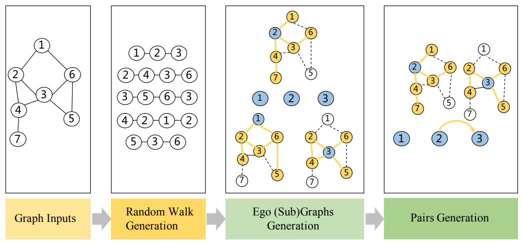


Figure 4: The procedure to generate sub-graphs.

Assuming that the neighbor count of the node is *N*, the maximum sample count of the node is *K*, and the final sampled result is stored in *res[K]*, and the data stores the real node ID of the neighbors. The main steps for the random walk are shown in Algorithm 1. Multiple start nodes can select the next node in parallel according to the reservoir algorithm, and the selected nodes will be used as the next sampling starting nodes. The walk paths are generated after the sampling operation executed *walk_len* steps. Pairs in the walks are generated according to the skip-gram method [11], and the duplicated nodes are used as the target nodes to generate the subgraphs.

Algorithm 1 Random Walk Based Sampling

```
Input: maximum sample count $K$, neighbor count $N$ for each target node.
  Output: sampled result array $res[K]$
 1: if $N \leq K$ then
 2:     for each $i \in [0, N-1]$ do
 3:         $res[i]$ =data[i]
 4:     end for
 5: else
 6:     for each $i \in [0, K-1]$ do
 7:         $res[i]$ =i
 8:     end for
 9:     for each $j \in [K, N]$ do
10:         $num$ =random (0,j)
11:         if $num \leq K$ then
12:             $res[num]$ =j
13:         end if
14:     end for
15:     for each $j \in [0, K-1]$ do
16:         $res[j]$ =data[res[j]]
17:     end for
18: end if
```

For the modified Fisher-Yates algorithm used for the subgraph sampling, the main steps are shown in Algorithm 2. The assumptions of neighbor count and maximum sample num are the same as Algorithm 1. Note that the attached pseudo-code describes the sampling process in a single step, while the whole sampling will not be terminated without achieving the step threshold *walk_len*. The

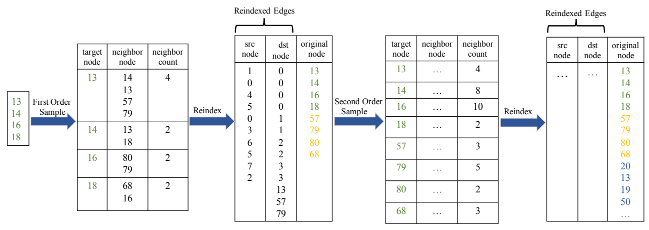


Figure 5: Reindex operation.

modified Fisher-Yates algorithm limits the shuffling range within K according to the different sizes of N and *walk_len*, which is different from the global shuffling operation in the original Fisher-Yates[7]. The modified algorithm can reduce the computational complexity and improve the sampling speed. The promising performance will be verified in the experimental section.

Algorithm 2 Modified Fisher-Yates Sampling

```
Input: maximum sample count $K$, neighbor count $N$ for each target node.
  Output: sampled result array $res[K]$.
 1: if $N \le K$ then
 2:   for each $i \in [0, N-1]$ do
 3:     $res[i] = data[i]$
 4:   end for
 5: else
 6:   if $N \le 2*K$ then
 7:     $begin = 0$
 8:     $split = K$
 9:   else
10:     $begin = N - K$
11:     $split = N - K$
12:   end if
13:   for $idx \in [split, N-1]$ do
14:     $num = random(0, idx)$
15:     swap($data[num], data[idx]$)
16:   end for
17:   for $idx \in [0, K-1]$ do
18:     $res[idx] = data[begin + idx]$
19:   end for
20: end if
```

Reindex Operation. The sequence of the target nodes and the sampled neighbor nodes can be obtained after sampling the deduplicated nodes. On top of the nodes sampled, a reindex operation is proposed to sort all the nodes in ascending order, to obtain the edges connected according to the renumbering, and the new sequence of nodes starting from the source node, the first-order neighbor, the second-order neighbor, etc. As shown in Figure 5, we assume the IDs of the target node are {13,14,16,18}, and the first-order neighbors are {14,13..68, 16}. After the first step of sampling, the reindexed target nodes and first-order neighbors are listed in the third table. The same operation iterates until sorted in order. It is obvious that the source node and neighbor nodes of each order are sorted from top to bottom as shown in the last column.

The motivation for reindexing is mainly to reduce intermediate redundant calculations actually. The model trains from high-order neighbors to low-order neighbors on the sampled subgraphs. As shown in Figure 6, {20,13,19,50} are unnecessary to be updated after the first layer training, because the updated gradient at this

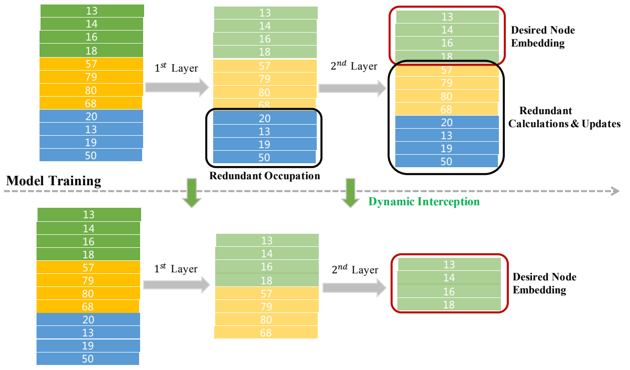


Figure 6: Dynamic interception during the model training.

time is calculated and aggregated without these nodes. Similarly, only target nodes are required to be updated in the second layer training. Thus, the redundant calculations in the middle layer can be intercepted to reduce storage space, which relies on the reindex operation as the prerequisite. The dynamic interception not only reduces memory usage but also boosts the calculation speed of the model training.

2.4 Other Key Optimizations

Variable-Length Slot Features. Compared to the ID of the graph node, the slot features on the node also play an important role in graph model training. In the current web-scale graph, the node usually contains text features and semantics, which are full of user behavior information and could contribute to business growth. However, the text features assigned to specific nodes' slots are extracted from the varying size of the original text data, which could cause a lot of redundant storage usage for traditional feature slots setup, especially for hyper-scale graphs. The traditional way is to use a fixed-length size to preset all the feature slots for convenience, where the size is required to cover the maximum length of the mixed features. It is obvious that such an all-in-one method could consume a fixed large amount of space in HBM to damage the capacity to load larger graphs. Therefore, we are motivated to design the variable-length feature slot, which stores in HBM in a filled-as-needed manner when pushing/pulling the corresponding embeddings, to increase the size of the trainable graphs in turn.

To enable the variable-length feature slot, the CPU in PGLBox is in charge of parsing the features and recording the length of features corresponding to various slots of the node (if the node contains the slot feature). All node IDs can be obtained when a pass has finished sampling. Then, the feature tables of the nodes in the CPU will be queried according to the node IDs, and the queried result is transferred to the GPU. The querying and filling process of variable-length slot features is shown in Figure 7. Specifically, the slot ID lists and feature lists of nodes in GPU are constructed at first. A hash table with nodeID as key and feature size/offset as value is also built into GPU, where feature size indicates the number of features according to the slot, and feature offset denotes the storage offset of the feature. When the allocation of the slot feature and embedding is initiated in the GPU, the hash table is queried to obtain the size and offset information according to the node ID. As drawn in the HBM block (Figure 7), the slot feature of all nodes can be retrieved by the feature size, the feature offset, and the pre-built feature list, where the offset of different slots can be further retrieved by the slot list. Finally, all the features in this pass are deduplicated for further embedding pulling, and filling in the slot loading tensor.

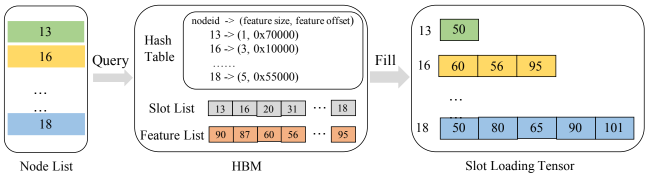


Figure 7: Query and fill of variable-length slot features.

Parsing and recording the lengths of different slot features in advance, allocating feature space and feature embedding space on demand, storing features, and corresponding embeddings in hierarchical levels are all beneficial to the support of higher-dimensional feature embedding and a larger number of features (i.e., larger graphs).

Metapath Split. To quickly sample edges on multi-GPUs, we store all the link relationships of edges in HBM, where nodes in the graph are set as the target nodes to be sampled according to the previously defined metapath. Typically, metapath is for sampling different walks composed of varying types of edges on demand. However, with the growth of the graph scale, the expanded edges cannot be fully stored in limited GPU HBM. To mitigate this issue, we propose the metapath split optimization, assigning the heterogeneous graphs with their loads into metapath-level subgraphs, where the resource limitation of a single machine could be broken through and the trainable scale of graphs could be further enlarged.

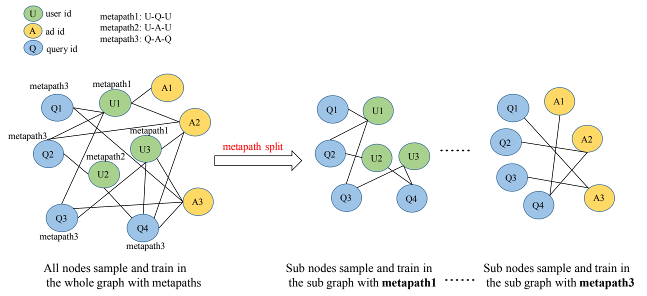

U user id
A ad id
Q query id
metapath1: U-Q-U
metapath2: U-A-U
metapath3: Q-A-Q
metapath3
metapath1
metapath1
metapath2
metapath3
metapath3
metapath3
Q1 U1 A1 A2 Q2 U3 U2 Q3 Q4 A3
metapath split
Q1 U1 Q2 U2 U3 Q3 Q4
……
Q1 A1 Q2 A2 Q3 Q4 A3
All nodes sample and train in the whole graph with metapaths
Sub nodes sample and train in the sub graph with metapath1
……
Sub nodes sample and train in the sub graph with metapath3

Figure 8: Metapath split optimization.

The key operation of metapath split is to split the processing of the entire graph into 1) the loading, 2) the sampling, and 3) the training of subgraphs at metapath level. As shown in Figure 8, only the edges and nodes assigned to the corresponding metapath are loaded in the CPU and GPU. The splitting operation first divides all metapaths with the same starting nodes into a specific class and set indexes to metapaths within the same class. Then, the set of starting nodes for sampling can be calculated according to the total length of the metapaths and the index of the current metapath, while the duplicated nodes are collected simultaneously for pulling embeddings. Once the trainable embeddings are ready, GNN training is initiated and organized in batches. With the metapath split, the peak storage space for edges in HBM decreases since only metapath-related edges are in charge of once splitting. The capacity of loaded edges could be doubled in real applications (introduced in the experimental section). The performance gains are visible but not that significant compared to DeepWalk under the GraphSAGE mode, which is necessary to upload all relevant edge types for the specified metapath starting points in the GPU memory, so the memory savings and graph scalability of the GraphSAGE model are significantly lower compared to DeepWalk.

Although the metapath split can drastically enlarge the scale of the graph loaded, the training of metapath-level subgraphs could affect the performance of the prediction accuracy (i.e., recall in our settings). The original starting nodes use width-first traversal to choose different metapaths cyclically, while the same starting nodes are evenly divided and each metapath is traversed in a depth-first manner. This could change the update order for embeddings compared to the original way so that the final recall is possibly influenced. To address the disorder, we sort different metapaths according to the number of starting nodes and edges. Specifically, metapaths are sorted in ascending order based on the class occurrence of their starting nodes, and then for the matapaths with the same class of starting node, the sorting will follow ascending order based on the number of edges in the metapath. We also evaluate the sorted metapath adjustment in the experimental section to demonstrate its usefulness.

2.5 Multi-Node Support

To further increase the capacity to load larger graphs, PGLBox supports distributed storage of node embeddings on multiple machines. Splitting node embeddings based on nodeID to different machines could take advantage of the remaining capacity of GPU resources. Specifically, embeddings can be prefetched from SSD into the memory of this pass, while different nodes communicate via nodesIDs exchanged in the form of RPC communication [26], which is used to build the HBM table for subsequent training. Then, the system leverages NCCL's all2all [28] strategy to query node embedding and communicate gradient data in the training process. Ahead of the multi-machine update, it is necessary to merge the gradients between multiple GPUs to ensure the consistency of the gradients.

3 DEPLOYMENT AND EVALUATION

In this section, we showcase the detailed deployments of our proposed PGLBox in a real-world industrial case and conduct comprehensive experiments to evaluate the performance of the overall system as well as the effect of each optimization strategy. Specifically, we are eager to answer the following questions based on the experimental results:

- How does PGLBox system perform compared with the MPI cluster in terms of efficiency and effectiveness?
- Does the three-stage pipeline in PGLBox achieve the effect of "work stealing" in multithreaded computation?
- What are the effects of the proposed optimization strategies on runtime and resource usage?

Setups. As shown in Table 1, we listed the detailed configurations for deploying PGLBox and MPI cluster in a real-world industrial case, where these two systems are launched to predict the neighbor information (user – ads click pairs) from a given graph built upon the data collected from Baidu's search engine. Based on the basic deployment, we conduct several experiments to measure the efficiency and effectiveness of deployed systems as well as the ablation test for each optimization strategy in PGLBox. Note that, the CPU in both systems keeps the same type for all the experiments.

Table 1: Overview of the system deployment.

System	MPI Cluster	PGLBox
Processor	Intel(R) Xeon(R) Gold 6271C CPU@2.60GHz (×40)	Tesla A100 (×8)
Memory	100T	320G
Storage	24T	24T
Remarks	+High-speed Ethernet switch +Single data center	+NVMe SSDs RAID 0 +NVLink +1.5T MEM CPU

Datasets. Two datasets are adopted in the experiments, which are (1) a large-scale dataset (denoted as **AdLogs**) created from Baidu's recommender system, and (2) an open graph benchmark dataset **MAG240M** [13]. To construct **AdLogs**, we collect the historical Ad logs for the past 180 days from Baidu's search engine, including the information (7 in total) about user ID (*uid*), clicks (*clk*), conversion (*conv*), Ad entity (*entt*), Video completion rate (*eplay*), click-through of e-commerce apps (*eclk*), and other related browsing records (*rid*). Then, we define 9 categories for the edges based on the collected Ad logs, which are *uid2clk*, *uid2conv*, *conv2entt*, *clk2entt*, *uid2pnclk*[2], *uid2pnconv*, *uid2eplay*, *uid2eclk*, and *uid2rid*. The final graph dataset consists of 1 billion nodes and 20 billion edges, where each node carries on 8 slot features, such as age, sex, etc. to enhance the node representations. The dataset split for training and testing (validation is skipped assuming the graph model's hyper-parameters are fitted already) follows a 180:1 ratio, where the past 180 days' logs are used to train the graph model, and the next day's logs are used for testing. For the second dataset, **MAG240M** is a heterogeneous academic graph extracted from the Microsoft Academic Graph (MAG) [34], typically used to predict the primary subject areas of the given arXiv papers. **MAG240M** dataset owns about 0.24 billion nodes and 1.7 billion edges respectively. We follow the default data splitting strategy in [13].

Graph Models. We adopt three popular graph representation learning models, i.e., DeepWalk, GraphSage, and UniMP [30] to evaluate PGLBox. Specifically, DeepWalk and GraphSage are commonly used in the actual business lines (applications) from Baidu, where we apply these two graph models in the real case with **AdLogs** dataset. For UniMP, we follow the original evaluation settings in [13] which uses the benchmark dataset MAG240M, DeepWalk is used to generate embeddings on the MAG240 dataset for the UniMP model. DeepWalk and GraphSage were originally proposed for homogeneous graph learning. while they have been also developed into heterogeneous graph learning [17, 21, 31]. Our current framework supports both homogeneous and heterogeneous graphs, and the baseline models can accommodate both types of graphs depending on the input data. When the input data is a heterogeneous graph, the model will automatically extend to a heterogeneous model.

Metrics. To evaluate the performance of the proposed PGLBox, we consider two perspectives, which are effectiveness and efficiency.

[2] *pnclk* and *pnconv* indicate the clicks and conversion brought by the Baidu's Phoenix PPC platform [8], which is different from the regular native Ads system.

For the effectiveness, we obtain the recall of Ads based on the well-trained GNN, where the value could be calculated by comparing the predicted Ad preferences with the real Ad clicks from users, defined as $recall = \frac{num\_of\_clicked\_Ads}{num\_of\_predicted\_Ads}$. For efficiency, we measure the system-wise cost including the training time, the memory usage, the training capacity (the size of the subgraphs loaded), and the utilization of computing units (i.e., CPU and GPU). Note that, the above efficiency indicators are observed when the baselines achieve the same training effect. Since the main comparison is between PGLBox and the MPI cluster, we also report the ratio of improvement or degradation (relative recall) based on the performance of the legacy trainer.

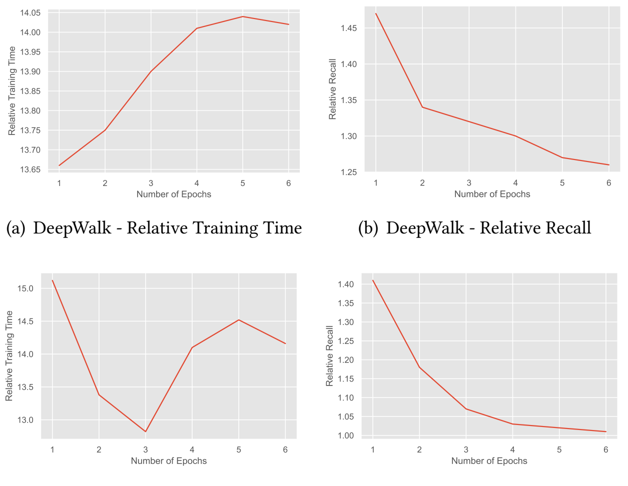
3.1 Case Study

In this subsection, we study a real case coming from Baidu's recommendation service, where we train a GNN model using the AdLogs dataset to predict the Ad clicks (edges) given users' information (nodes). For a fair comparison, the legacy MPI cluster and PGLBox use the same training and testing datasets in an A/B test manner. To compare the effectiveness and efficiency of these two systems, we report the training time, the speedup ratio, and the recall of Ads generated by DeepWalk and GraphSage.

Figure 9(a) shows the total training speedup ratio of PGLBox compared to the MPI cluster with DeepWalk. The comparison involves 6 epochs and over one million passes. The training time of a single epoch with the MPI cluster is a multiple of 13.66 with PGLBox. It is reasonable that the two most time-consuming stages – sampling and training are both implemented on GPU in PGLBox, which is faster than processing on CPU in the MPI cluster, with the help of GPU's powerful parallel computing. We can also observe that relative training time grows when the epoch number increases, it can achieve 14.04 maximally in the fifth epoch. It is mainly due to the failure of inter-node communication and the retry of processing failures increases rapidly when need training multiple epochs in the MPI cluster. While the training time of each epoch of PGLBox is basically the same for the relatively stable process success rate. This also results in the stability of PGLBox being higher than the MPI cluster. Moreover, the hardware and maintenance cost of PGLBoxis less than 55% of the MPI cluster's cost.

To evaluate the effectiveness, we use the recall of Ads to measure the quality of the node embedding. Figure 9(b) shows the recall relative to the MPI cluster of trained embedding by DeepWalk. We train the input graph data with multiple epochs to obtain the node embedding and calculate the similarity to get the top 50 Ads (predicted), then we use these top 50 Ads and the real clicked Ads in the past 3 days to calculate the recall. Since the recall is highly related to the revenue of recommendation, we have to guarantee acceptable recall generated by PGLBox. The results show that PGLBox has an even higher recall than the MPI cluster. Moreover, the relative recall decreases as the number of epochs increases, which indicates that the best model parameters can be obtained by both PGLBox and the MPI cluster, while PGLBox consumes a shorter period of time. Similar results could be observed from GraphSage. As shown in Figure 9(c)(d), the comparison of recall shows consistent trends with the DeepWalk, while the speedup ratio is around 14 on average.

Summary. The improvement in training speed (Figure 9) primarily stems from the distributed GPU graph engine, which includes three main aspects: 1) Full GPU implementation of graph storage, random walks, sampling, and training, eliminating the inefficiencies of CPU processing and extensive data interaction between the CPU and GPU. 2) Employing a multi-GPU distributed architecture to partition the graph, attributes, and model, enabling parallel acceleration and scalability. 3) Addressing the issues of non-fully connected NVLink and network card topologies by implementing intelligent relay communication. The enhancement in convergence speed is mainly due to the more frequent parameter synchronization compared to single-node GPU and distributed MPI. This results in more updated parameters being pulled by nodes during training.



(a) DeepWalk - Relative Training Time
(b) DeepWalk - Relative Recall
(c) GraphSage - Relative Training Time
(d) GraphSage - Relative Recall

Figure 9: Performance improvements on top of the legacy trainer (the MPI cluster).

In general, in such a real case with web-scale graphs, we could conclude that the overall performance of PGLBox surpasses the traditional MPI cluster in significant gaps. Other than this real case, we also evaluate PGLBox on an open-sourced benchmark dataset MAG240M to showcase its efficiency on general graph training, the results in Table 2 depict that PGLBox still outperforms the legacy MPI cluster with around 10 times training speedups.

graph model	MPI	PGLBox	speedup
DeepWalk	147 min	16 min	9.19
GraphSage	192 min	19 min	10.11

Table 2: Training speedup in one epoch on MAG240M (with the same training loss).

3.2 Abalation Test

In this subsection, we investigate the effects of the proposed framework designs and optimization strategies: three-stage pipeline, variable-length slot feature, and metapath split.

Three-Stage Pipeline. We first evaluate the proposed three-stage pipeline, where the relative execution time of sampling, pulling & updating embedding, and training GNNs are shown in Table 3. The total execution time of training one epoch corresponds to 100%. The stage Training GNN takes 79% time of an epoch, which dominates the entire pipeline runtime. Note that 14% of the remaining 21% is model saving time, and the rest 7% time is the communication and processing overhead. The latency of stage sampling and pulling & updating embedding are hidden in the pipeline, which demonstrates the effectiveness of asynchronous work stealing.

Stage	Avg time	STD DEV
Sample	65.38%	15.70%
Pull Emb	5.31%	15.39%
Train	79.53%	5.35%
Save Model	14.18%	9.2%

Table 3: Relative training time of each stage in the pipeline, 100% is the average time of training one epoch.

slot num	5	50	100	125
gpu_mem (F)	17.31	28.9	36.6	38.5
gpu_mem (V)	17.06	16.8	17.07	17.5
mem_save	1.38%	41.86%	53.36%	54.54%
train_time (F)	199	241	254	269
train_time (V)	192	197	203	199
time_save	3.52%	18.25%	20.08%	26.02%

Table 4: GPU memory usage and training time of variable-length design compared with fixed-length design.

Variable-length Slot Features. We further evaluate the design of variable-length slot features. Table 4 shows the GPU memory usage and training time with different numbers of slot features in the DeepWalk training, where we use (F) to represent fixed-length storage, and (V) to represent variable-length slot feature. It can be observed that GPU memory usage and execution time are reduced significantly from fixed-length design to variable-length design, while the gap is enlarging with the increase of slot feature numbers. It is common sense that more GPU memory is occupied when the slot feature size increases. The legacy fixed-length design allocates storage according to the maximum space, while the variable-length design only allocates the needed GPU memory, which helps reduce the redundancy occupation. Moreover, there are a lot of 0 keys in fixed length allocation to occupy storage, and these 0 keys also participate in pulling parameters and pushing gradients in GNN training, which results in training time growing rapidly. The variable-length slot features could mitigate this issue as a bonus.

Metapath Split. We finally evaluate the design of metapath split, by running a graph learning task to compare the memory usage and the graph scale with and without the optimization. Table 5 shows the GPU memory usage and graph scale in nodes with slot features and nodes without slot features. We can observe that the released GPU memory (memory saving) is significant no matter whether the node has a slot feature or not when applying the metapath split strategy. Furthermore, the edge capacity of the graph can reach twice the original size benefiting from the metapath splitting in the 0 slot setting, while it provides 3.8 times expansion for the node in the 8 slot setting. Thus, splitting graphs into sub-graphs in the way of metapath could save a lot of memory, which can increase the capacity for larger graphs in turn.

slot num	0 slot	8 slots
gpu_mem (w/o)	38.4	40
gpu_mem (w/)	22	26
mem_save	42.7%	33.25%
node_scale (w/o)	1 billion	1 billion
node_scale (w/)	1 billion	3.8 billion
edge_scale (w/o)	20 billion	20 billion
edge_scale (w/)	40 billion	20 billion

Table 5: GPU memory usage and graph capacity w/ Metapath Split and w/o Metapath Split.

3.3 Effect of Miscellaneous Designs

The effects of two additional designs are reported, which are beneficial to PGLBox in terms of efficiency and effectiveness.

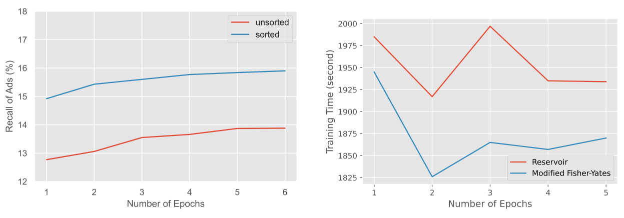
High-speed Interconnections via NVlink. We compare the training time of PGLBox w/ and w/o high-speed interconnections (HIN) with DeepWalk and GraphSage. Specifically, we alter the number of slot features to observe the time difference on one epoch training. As shown in Table 6, the speedup ratio is significant with HIN. Note that, the speedup ratio for the setting of 8 slot features is relatively smaller than the 0 slot feature due to the frequent communication with the CPU for the retrieval and query of features.

slot num	0 slot	8 slots
DeepWalk		
w/ HIN	156 min	237 min
w/o HIN	49 min	104 min
speedup	3.18	2.28
GraphSage		
w/ HIN	168 min	209 min
w/o HIN	64 min	104 min
speedup	2.63	2.01

Table 6: Effect of HIN.

Sorted Metapath. We compare the recall of Ads for PGLBox before and after applying the sorted metapath strategy with DeepWalk. Specifically, the differences in recalls are recorded from the first epoch to the sixth epoch. As shown in Figure 10(a), the performance gain is significant after applying the sorted metapath strategy.

Sampling Optimization. As aforementioned in Section 2.3, we proposed a modified Fisher-Yates algorithm to further boost the speed of sampling in GraphSage model. Specifically, we compare the modified Fisher-Yates algorithm with the baseline reservoir sampling algorithm in terms of training time consumption on the AdLogs dataset. The result in Figure 10(b) demonstrates that the proposed algorithm can save considerable time during the sampling process in each training epoch.



(a) Sorted Metapath (b) Sampling Strategy in GraphSage

Figure 10: Training speedup from the sorted metapath and modified Fisher-Yates sampling.

3.4 Summary and Discussion

Based on the above results, we can answer the questions driving the experimental evaluation with clear evidence. For both GraphSage and DeepWalk graph models, PGLBox reduced the training time of a 40-node MPI cluster by at least tenfold, and cost less than 55%. Additionally, node embedding trained by PGLBox yielded a better recall than the distributed parameter server in the cluster solution with multiple epochs. It is likely because of the more frequent parameter synchronizations in the single-node architecture of PGLBox. The graph neural network training dominates the execution time, meaning the latency of sampling and pulling embedding are largely hidden (work stealing). Furthermore, the variable length slot features can reduce GPU memory usage and training time when the slot feature number is high. Lastly, splitting the graph in metapath way reduces the redundant edges loaded to GPU memory, increasing the scale of graph data.

4 RELATED WORK

Due to the rising interest in GNNs, frameworks designed specifically for expressing and accelerating GNNs are developed. Euler[41] and PGL[22] focus on sampling-based mini-batch training on the large graph but lacks GPU support. ROC[15] is a distributed multi-GPU framework for GNN training on full-graphs without using sampling techniques, so it is not eligible for graph inductive representation learning, and graph size is also limited. NextDoor[14] was designed to perform graph sampling on a single GPU. PyTorch-Geometric[10] is an extension for geometric deep learning to the PyTorch framework. PyG's programming model is centered around sparse tensor abstraction. During message passing, it first gathers node features to edges, applies a user-defined message function and then scatters them to the target node for aggregation. These scatter-gather patterns are inefficient due to generating large intermediate message tensors. DGL[35] is also a popular GNN framework using a message-passing interface but fails to make the most of GPU resources due to low GPU utilization. NeuGraph[23] accelerates GNN training by partitioning graphs to multiple GPUs, while intermediate GNN data is stored in the host CPU, and training performance is limited by the communication between CPU and GPU. A specific GPU-oriented data communication architecture is provided in[25] for multi-GPU GNN training. While PGLBox is a huge and deep graph learning framework, based on multi-GPUs with hierarchical storage. It fits well for inductive representation learning on graphs for combining graph operations and neural network training.

There are several existing graph learning frameworks, such as GraphVite, PyTorch-BigGraph[20], and Graph4Rec[22]. GraphVite only performs walk-based models on a single machine with multi-GPUs. Although PBG supports distributed training, it cannot deal with heterogeneous GNN models, lacking the capability of modeling complex structural data for recommender systems. Graph4Rec store graph structures and features on CPU and also training network on CPU in MPI cluster, high cluster cost, and low throughput bound the wide board usage.

5 CONCLUSION

GNNs are popular in recent years due to their capability to learn information from graph-structured data. Since the size of graphs is growing rapidly, the demand for processing large-scale graph datasets is urgent. In this work, we propose a novel efficient GNN training framework-PGLBox, based on a multi-GPU system with hierarchical storage consisting of SSD, MEM, and HBM. A pure GPU-based graph processing engine is built for sampling, walking, and training, releasing the burden caused by the communication between CPU and GPU. Three stage pipeline is designed to overlap the execution of sampling, pulling & updating sparse tables, and training. Moreover, multiple optimization operations are proposed to further improve the overall performance. The comprehensive experiments show that PGLBox outperforms the legacy MPI cluster in terms of efficiency and effectiveness and is fitting well with the web-scale graph data in the industrial recommender systems.

REFERENCES

[1] Davide Bacciu, Federico Errica, and Alessio Micheli. 2018. Contextual graph markov model: A deep and generative approach to graph processing. In *International Conference on Machine Learning*. PMLR, 294–303.

[2] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46, 5 (1999), 720–748.

[3] Junxuan Chen, Baigui Sun, Hao Li, Hongtao Lu, and Xian-Sheng Hua. 2016. Deep ctr prediction in display advertising. In *Proceedings of the 24th ACM international conference on Multimedia*. 811–820.

[4] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems* 29 (2016).

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[6] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. 2017. metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 135–144.

[7] Richard Durstenfeld. 1964. Algorithm 235: random permutation. *Commun. ACM* 7, 7 (1964), 420.

[8] Miao Fan, Jiacheng Guo, Shuai Zhu, Shuo Miao, Mingming Sun, and Ping Li. 2019. MOBIUS: towards the next generation of query-ad matching in baidu's sponsored search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2509–2517.

[9] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph neural networks for social recommendation. In *The world wide web conference*. 417–426.

[10] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[11] David Guthrie, Ben Allison, Wei Liu, Louise Guthrie, and Yorick Wilks. 2006. A closer look at skip-gram modelling.. In *LREC*, Vol. 6. 1222–1225.

[12] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

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

[14] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating graph sampling for graph machine learning using GPUs. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 311–326.

[15] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.

[16] Zilong Jiang. 2016. Research on ctr prediction for contextual advertising based on deep architecture model. *Journal of Control Engineering and Applied Informatics* 18, 1 (2016), 11–19.

[17] Andrea Stevens Karnyoto, Chengjie Sun, Bingquan Liu, and Xiaolong Wang. 2022. Augmentation and heterogeneous graph neural network for AAAI2021-COVID-19 fake news detection. *International journal of machine learning and cybernetics* 13, 7 (2022), 2033–2043.

[18] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[19] Jaejin Lee, Changhee Jung, Daeseob Lim, and Yan Solihin. 2008. Prefetching with helper threads for loosely coupled multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems* 20, 9 (2008), 1309–1324.

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

[21] Qiuyan Li, Yanlei Shang, Xiuquan Qiao, and Wei Dai. 2020. Heterogeneous dynamic graph attention network. In *2020 IEEE International Conference on Knowledge Graph (ICKG)*. IEEE, 404–411.

[22] Weibin Li, Mingkai He, Zhengjie Huang, Xianming Wang, Shikun Feng, Weiyue Su, and Yu Sun. 2021. Graph4Rec: A Universal Toolkit with Graph Neural Networks for Recommender Systems. *arXiv preprint arXiv:2112.01035* (2021).

[23] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. {NeuGraph}: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 443–458.

[24] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. 2019. PaddlePaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing* 1, 1 (2019), 105–115.

[25] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021. Large graph convolutional network training with gpu-oriented data communication architecture. *arXiv preprint arXiv:2103.03330* (2021).

[26] Bruce Jay Nelson. 1981. *Remote procedure call*. Carnegie Mellon University.

[27] Minh Tuan Nguyen and Keith A Teague. 2014. Neighborhood based data collection in wireless sensor networks employing compressive sensing. In *2014 International Conference on Advanced Technologies for Communications (ATC 2014)*. IEEE, 198–203.

[28] Yuechao Pan. 2019. *Multi-GPU Graph Processing*. Ph.D. Dissertation. University of California, Davis.

[29] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 701–710.

[30] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjin Wang, and Yu Sun. 2020. Masked label prediction: Unified message passing model for semi-supervised classification. *arXiv preprint arXiv:2009.03509* (2020).

[31] Yunsheng Shi, PGL Team, Zhengjie Huang, Weibin Li, Weiyue Su, Shikun Feng, et al. 2021. R-UNIMP: Solution for KDD Cup 2021 MAG240M-LSC. *Open Graph Benchmark-Large-Scale Challenge@ KDD Cup 2021* (2021).

[32] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[33] Jeffrey S Vitter. 1985. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)* 11, 1 (1985), 37–57.

[34] Kuansan Wang, Zhihong Shen, Chiyuan Huang, Chieh-Han Wu, Yuxiao Dong, and Anshul Kanakia. 2020. Microsoft academic graph: When experts are not enough. *Quantitative Science Studies* 1, 1 (2020), 396–413.

[35] Minjie Yu Wang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*.

[36] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & cross network for ad click predictions. In *Proceedings of the ADKDD'17*. 1–7.

[37] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 974–983.

[38] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. 2018. Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine* 13, 3 (2018), 55–75.

[39] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. *Advances in neural information processing systems* 31 (2018).

[40] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. 2019. Aibox: Ctr prediction model training on a single node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 319–328.

[41] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. Aligraph: a comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730* (2019).

A SUMMARY OF TERMINOLOGIES

We summarize the full definition of the domain-specific terminology in this section with references.

- **Slot**: the slot is equal to a feature slot, used to represent a broad feature category, such as user ID, gender, age, etc..
- **Metapath**: Metapath [6] is a path connecting a series of edge types in a heterogeneous graph network. We define multiple paths during sampling to mine various structural relationships in the heterogeneous graph. For example, we can connect different user IDs through the "userid2clk-clk2userid" path to explore the potential relationships between users.
- **Pass**: the sampling starting points are divided into multiple passes, with a specific number of sampling starting points forming one pass. After sampling, pairs are formed based on the sampling results to obtain training instances. The sampling results of one pass are split into multiple mini-batches. Thus, sampling is performed at the pass level, while GNN training is at the mini-batch level. The primary purpose of this approach is to construct hierarchical storage, prefetch pass-level embedding, and enable full GPU training.
- **Work Stealing**: Work Stealing is a scheduling strategy for multi-threaded computer programs. It solves the problem of executing a dynamically multithreaded computation, one that can "spawn" new threads of execution, on a statically multithreaded computer, with a fixed number of processors.

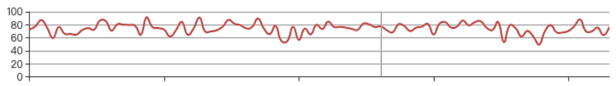
B SYSTEM-WISE UTILIZATION

The system-wise utilization of computing units from PGLBox and the MPI cluster is further presented in this section. Figure 11(a)(b) presents the utilization of GPUs on PGLBox and CPUs on the MPI cluster during the training of the DeepWalk model. We can observe that PGLBox has a higher utilization (pure GPUs) than the MPI cluster (pure CPUs) and the curve of PGLBox is more stable than the MPI cluster with DeepWalk. Note that the size of the graph generated by DeepWalk and GraphSage differs due to the size of sub-datasets, where the sub-dataset trained with DeepWalk (7.8 hundred million nodes, 110 hundred million edges) is nearly twice as large as the one with GraphSage (13 hundred million nodes, 200 hundred million edges). In this case, the GPU utilization with GraphSage on PGLBox is relatively lower than Deepwalk, while the CPU utilization with GraphSage on the MPI cluster still keeps at a high level.

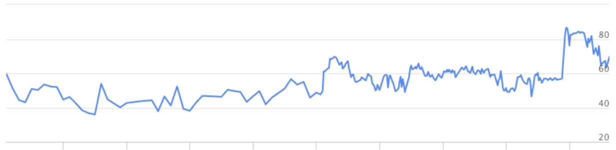
C FURTHER DISCUSSION ON EXPERIMENTS

C.1 GNN vs LM as the feature encoder

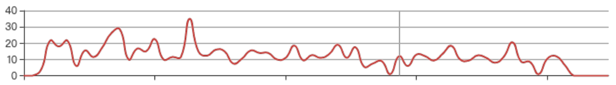
We also consider using LM to fit the problem, while by now it is still a challenging task. Language modeling (LM) involves the use of various statistical and probability techniques to determine the likelihood of a given word sequence appearing in a sentence. LM analyze the main body of text data, providing a foundation for their word predictions. In our graph training, the input text is determined by user clicks, and we process it as feature input for the graph training, which differs from traditional language model probability predictions.



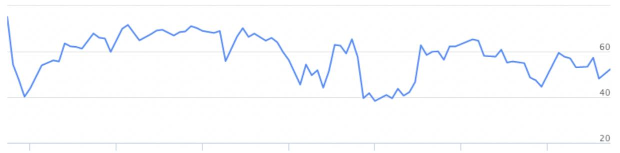
(a) DeepWalk - GPU utilization on PGLBox



(b) DeepWalk - CPU utilization on the MPI cluster



(c) GraphSage - GPU utilization on PGLBox



(d) GraphSage - CPU utilization on the MPI cluster

Figure 11: System-wise utilization comparison between PGLBox and the legacy trainer. The vertical axis denotes the percentage (%) of usage, and the horizontal axis indicates the training time elapses in a single epoch.

C.2 Updating Embeddings

Currently, our embedding update optimizer supports Adagrad and Adam, with each optimizer storing different variables. For example, under the Adagrad optimizer, each node embedding stores the gradient square sum (g2sum) and the actual embedding used in the network. Under the Adam optimizer, each node embedding stores beta1, beta2, gradient square sum (g2sum), and the actual embedding used in the network. Node embeddings are updated within each batch during the backward computation process. The push_sparse operator is called to compute the gradient for each node embedding, and the embeddings are updated in place on the GPU card directly based on the optimizer algorithm being used.

C.3 Training with Multiple GPUs

We do employ data parallelism during the sampling phase by dividing different sampling starting points across multiple GPU cards. Each card is allocated separate storage space for sampling results, and the sampling is conducted independently for different starting points without synchronization between cards. During the training phase, each card accesses the sampled result space to form pairs for training instances. The 8 GPU cards share GNN parameters, and each card pulls GNN parameters before mini-batch training to perform forward and backward calculations, obtaining gradients for updating GNN parameters. The 8 cards synchronize GNN parameters after each mini-batch ends and pull the same GNN parameters before the next mini-batch begins. Note that, our sampling approach is a combination of data parallelism and pipeline parallelism.