Adaptive Parallel Training for Graph Neural Networks

Kaihao Ma[*†]
The Chinese University of Hong Kong
khma@cse.cuhk.edu.hk

Renjie Liu[*†]
Southern University of Science and Technology
liurj2023@mail.sustech.edu.cn

Xiao Yan[‡]
Centre for Perceptual and Interactive Intelligence
yanxiaosunny@gmail.com

Zhenkun Cai
Amazon
zkcai@amazon.com

Xiang Song
Amazon
classicxsong@gmail.com

Minjie Wang
AWS Shanghai AI Lab
minjiw@amazon.com

Yichao Li
The Chinese University of Hong Kong
yichaoli21@cse.cuhk.edu.hk

James Cheng
The Chinese University of Hong Kong
jcheng@cse.cuhk.edu.hk

Abstract

There are several strategies to parallelize graph neural network (GNN) training over multiple GPUs. We observe that there is no consistent winner (i.e., with the shortest running time), and the optimal strategy depends on the graph dataset, GNN model, training algorithm, and hardware configurations. As such, we design the APT system to automatically select efficient parallelization strategies for GNN training tasks. To this end, we analyze the trade-offs of the strategies and design simple yet effective cost models to compare their execution time and facilitate strategy selection. Moreover, we also propose a general abstraction of the strategies, which allows to implement a unified execution engine that can be configured to run different strategies. Our experiments show that APT usually chooses the optimal or a close to optimal strategy, and the training time can be reduced by over 2x compared with always using a single strategy. APT is open-source at https://github.com/kaihaoma/APT.

CCS Concepts: **• Computing methodologies → Machine learning**; **Parallel computing methodologies**.

Keywords: Graph Neural Networks, Distributed and Parallel Training, Network Communication

[*]Both authors contributed equally to this research.
[†]Work done during the authors' internship at AWS Shanghai AI Lab.
[‡]Dr. Xiao Yan is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPoPP '25, March 1–5, 2025, Las Vegas, NV, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1443-6/25/03
https://doi.org/10.1145/3710848.3710883

1 Introduction

Graph data are prevalent in domains such as e-commerce, finance, biology, and social networks. As machine learning models specialized for graph data, *graph neural networks* (GNNs) yield good accuracy for graph tasks such as node classification [27], link prediction [54], and graph clustering [44]. Hence, GNNs are widely used for many applications including recommendation [51], fraud detection [46], and drug discovery [13]. As real graphs can be large (e.g., with millions of nodes and billions of edges), to achieve a short training time for GNNs and host large graph data, parallel execution with multiple GPUs is a common choice [35, 57].

The core problem of parallel GNN training is how to partition the data graph and model computation among the GPUs, and several parallelization strategies (called strategies for conciseness afterwards) are proposed as solutions [6, 24, 25, 41]. In brief, GNN training samples subgraphs from the data graph around some seed nodes and runs model computation on the sampled subgraphs. The classical *graph data parallel* (GDP) [47, 57] strategy assigns each GPU to process some seed nodes along with their sampled subgraphs; *node feature parallel* (NFP) [10] partitions the input node features by dimension among the GPUs and runs each GPU to compute on their local feature dimensions; *source node parallel* (SNP) [39] partitions the sampled subgraphs by source nodes and assigns each GPU to compute the contributions from its local source nodes to their destination nodes; motivated by existing strategies, we propose *destination node parallel* (DNP), which partitions the sampled subgraphs by destination nodes and assigns each GPU to aggregate embeddings for their local destination nodes.[1]

The four strategies are semantically equivalent in that they execute the same algorithm logic and produce identical trained models, and thus the question becomes which one runs the fastest. We observe there is no consistent winner, and the optimal strategy depends on the specifics of

[1]Originally, GDP is called data parallel, NFP is called model parallel, and SNP is called split parallel. We change their names to be more intuitive.

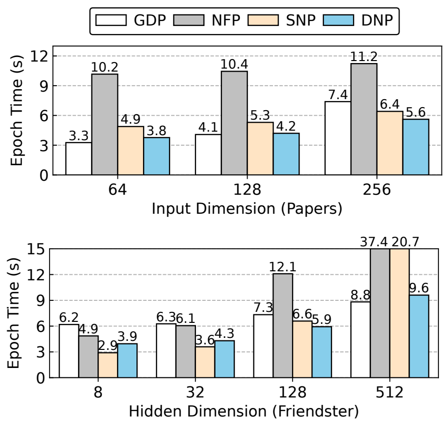


Figure 1. Epoch time for training GraphSAGE model on one machine with 8 GPUs, varying hidden and input dimension.

the GNN training task, which include the data graph, GNN model, training algorithm, and hardware platform. We provide two such examples in Figure 1. In particular, for the Papers graph [15], GDP is the optimal when the input feature dimension is 64, and when the input dimension becomes 256, GDP runs over 30% slower than the new optimal DNP. For the Friendster graph [49], SNP has the shortest running time when the hidden dimension is 8 and 32, but DNP and GDP run the fastest when the hidden dimensions are 128 and 512, respectively. Moreover, Figure 1 also shows that the running time of the strategies varies by a large margin for the same task, and thus the performance penalty can be high if a GNN training task is set to use an improper strategy.

Motivated by these observations, we propose the *strategy selection problem* for parallel GNN training, which chooses and runs an efficient strategy given the specifics of a GNN training task. Strategy selection is difficult because the optimal strategy depends on many factors (cf. Figure 1). We should also make selection fast as it contributes to the end-to-end training time. Moreover, existing GNN systems can only run a single strategy and thus may not be able to execute the optimal strategy even if it can be selected accurately. To avoid the complexity of working with multiple systems, we need a general system that can run all the strategies.

By tacking the challenges above, we design the APT system. Users simply provide APT with the specifics of a GNN training task (e.g., data, model, and platform), and APT automatically selects and executes an efficient parallelization strategy for the training task. The cores of APT are *effective cost models* and a *unified execution engine*. In particular, the cost models estimate the running time of different strategies such that the optimal strategy can be selected. We observe that the costs have common parts for all strategies and unique parts for each strategy, and thus we compare only the unique parts. Moreover, we conduct dry-run on the data graph to collect data-dependent statistics in the cost models and make the dry-run fast by skipping model computation.

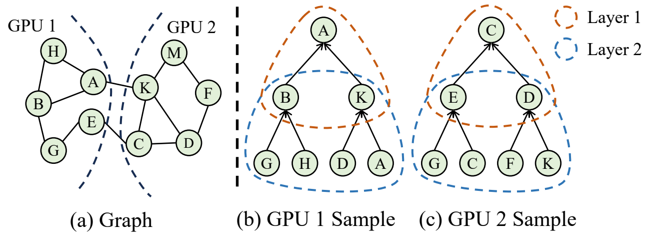


Figure 2. An illustration of graph sampling. A and C are the seed nodes, and node-wise sampling is used.

The unified execution engine can be configured to run different strategies and is implemented using DGL [47] as the single GPU engine. The insight is that different strategies essentially partition the tensors of a GNN layer along different dimensions; and after partitioning, each GPU runs a computation graph with source and destination nodes. To support different strategies, we insert computation and communication operators to prepare inputs for the single-GPU DGL workhorse and post-process its outputs. We also implement a unified feature store such that the GPUs can read input node features cached in a complex memory hierarchy.

We evaluate APT for multi-GPU GNN training on both a single machine and multiple machines. The results show that APT finds the optimal or a near-optimal strategy. Compared to always using a single strategy, the maximum speedup of APT is over 2x in most cases (i.e., a graph plus a model for a case). Micro experiments suggest that our cost models have a low overhead and achieve good estimation accuracy. We also experiment extensively to explore and analyze how different factors affect the trade-offs among the strategies.

To summarize, we make the following contributions:

- We survey existing parallelization strategies for GNN training and analyze their pros and cons in detail.
- We observe that there is no consistent winner among the strategies and propose the strategy selection problem.
- We design the APT system to automatically select and run efficient strategies for GNN training tasks, which features effective cost models and a unified execution engine.
- We conduct extensive experiments to check the influence of various factors (e.g., data, model, and platform) on the choice of the optimal parallelization strategy.

2 Background on GNN Training

GNNs consider a data graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where each node $v \in \mathcal{V}$ has an input *feature vector* h_v^0 (e.g., profiles of each product and user in a user-product graph for e-commerce). A K-layer GNN model computes an output embedding h_v^K for each node v by aggregating the embeddings of v's neighbors. For the kth ($k \geq 1$) layer, the GNN model can be expressed as

$$h_v^k = \sigma\left[AGG^k\left(\{W^k h_u^{k-1}, \forall u \in \mathcal{N}(v)\}\right)\right], \quad (1)$$

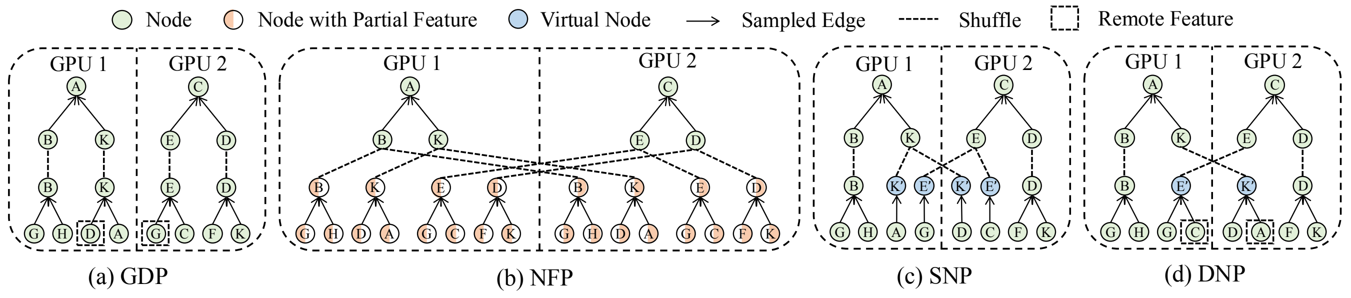


Figure 3. An illustration of the 4 parallelization strategies with 2 GPUs and 2 seed nodes. For SNP and DNP, the nodes managed by the 2 GPUs are shown in Figure 2(a), i.e., GPU 1 manages nodes A, B, E, G, H, and GPU 2 manages nodes C, D, F, M, K.

where set $\mathcal{N}(v)$ contains v's neighbors, h_u^{k-1} is the $(k-1)^{\text{th}}$ layer embedding of node u, and for the 1st layer, h_u^0 is the input feature vector of node u. As the model parameter of the k^{th} layer, W^k is a matrix of size $d' \times d$, where d' is the dimension of h_v^k and d is the dimension of h_v^{k-1}. $AGG^k(\cdot)$ is the aggregation function, which merges the neighbor embeddings into one embedding. $\sigma(\cdot)$ is the activation function.

Unfolding Eq. (1), we observe that computing h_v^K requires all K-hop neighbors of v. As a node has many K-hop neighbors, *graph sampling* is used to sample some neighbors to reduce computation [4, 5, 12, 32, 42, 53]. There are many graph sampling algorithms, and Figure 2 provides an example of the popular node-wise sampling scheme [5, 12, 33, 52, 55]. In particular, node-wise sampling is conducted by layers, with layer 1 sampling the neighbors of the seed node and layer 2 sampling the neighbors of the layer 1 samples. Node-wise sampling is configured by a fanout vector, for instance, Figure 2 uses a fanout of [2,2], which means that both layers sample two neighbors for each node. Considering the sampled subgraph for seed node A, to compute the layer-2 output embedding of node A, we require the layer-1 embeddings of nodes B and K, which in turn are computed using the input features of nodes G, H and D, A, respectively.

Training procedure. GNN training is conducted in mini-batches, and each mini-batch involves three steps, i.e., *sampling*, *loading*, and *training*. In particular, *sampling* constructs subgraphs for the seed nodes in a mini-batch according to the graph sampling algorithm; *loading* gathers the input feature vectors for all nodes in each sampled subgraph, which are required for model computation; *training* conducts the forward pass to compute model outputs for the seed nodes and the backward pass to compute gradients for the model parameter. Training is said to have finished an *epoch* when the mini-batches enumerate all seed nodes.

3 Parallelization Strategies and Cost Models

Here, we first introduce the 4 parallelization strategies, then present our cost models to compare their execution time, and finally discuss their trade-offs based on the cost models.

3.1 Parallelization Strategies

We assume a single machine with multiple GPUs when introducing the strategies. We call the GNN layer furthest from the seeds as the first layer of *computation* (e.g., between nodes B, K and G, H, D, A for seed node A in Figure 2), which is the last layer of *graph sampling*. We also note two common design considerations for parallel GNN training. First, GNN models are usually small, and thus the strategies focus on the communication for the sampled subgraphs, input node features, and hidden node embeddings, which are much larger. Second, the first layer of GNN dominates training cost because it involves many more nodes than the other layers. All strategies target the first layer and use data parallel training for the other layers because these layers are lightweight, and overheads of the strategies outweigh their benefits (mainly reduced communication). Figure 3 provides an illustration of the 4 strategies assuming that 2 GPU workers are used to process the 2 sampled subgraphs in Figure 2.

Graph data parallel (GDP). As shown in Figure 3(a), GDP assigns each GPU to process some seed nodes independently, including conducting graph sampling, loading the input node features, and running model computation. If some input node features required by a GPU are not cached on it, the GPU loads these node features from the CPU or peer GPUs.

Node feature parallel (NFP). NFP partitions the input node features by dimension over the GPUs. If the node feature has dimension d and there are C GPUs, each GPU manages d/C dimensions of all node features. The GPUs still sample subgraphs for their assigned seeds independently; but after that, they broadcast their layer-1 computation graphs to all GPUs. The first layer model W^1 is co-partitioned with the features, and thus each GPU gets some columns of W^1. To conduct computation, each GPU collects the computation graphs of all GPUs, projects its local feature dimensions using the partial model, aggregates the layer-1 outputs targeted for each node, and sends the partial embeddings to their destined GPUs. For instance, in Figure 3(b), GPU 2 aggregates the contributions from the latter-half feature dimensions of nodes G, H to target node B. We say the local aggregation is *partial* because it does not have all inputs for a target node. On the receiver side, each GPU aggregates the hidden embeddings from all GPUs to compute the layer-1 embeddings for the nodes in its graph samples. For instance, node B on GPU 1 aggregates the hidden embeddings from both GPU 1 and GPU 2. In the backward pass, the GPUs broadcast the gradients of their layer-1 embeddings, which are used to compute the gradients for layer-1 model W^1 on each GPU.

Source node parallel (SNP). For layer-1, we call the nodes whose output embeddings are computed *destination nodes* (e.g., nodes B, K on GPU 1), and the nodes whose input features are required as *source nodes* (e.g., nodes G, H, D, A on GPU 1). With C GPUs, SNP uses an edge-cut (e.g., METIS [24]) to partition the graph $\mathcal{G}$ into C partitions with a small number of cross-partition edges, e.g., Figure 2(a). Each GPU manages the source nodes in one partition. As shown in Figure 3(c), if a destination node requires source nodes from another GPU, a virtual node is created for it on the target GPU. For instance, destination node K of GPU 1 requires source node D, which is managed by GPU 2; and thus a virtual node K' is created on GPU 2. Each GPU holds a replica of the layer-1 model W^1 to project and aggregate the contributions of its source nodes to each virtual node (i.e., $W^1 h_u^0$); and the partial embeddings are sent back to the GPU requesting the virtual nodes (e.g., GPU 1 for virtual node K') to compute the complete layer-1 embeddings.

Destination node parallel (DNP). Inspired by SNP, we propose a new parallelization strategy called DNP. Like SNP, DNP uses an edge-cut to partition the data graph to the GPUs and assigns each GPU to manage the nodes in a graph partition. However, DNP focuses on destination nodes and sends each destination node to its managing GPU. For instance, in Figure 3(d), source node K on GPU 1 is sent to GPU 2, which manages nodes C, D, F, M, K. For each destination node, the managing GPU is responsible for loading the features of its source nodes and computing layer-1 embedding. These layer-1 embeddings are transferred back to the destined GPUs to compute the subsequent layers.

3.2 Cost Models for Strategy Selection

We observe that the running time of all strategies can be decomposed into four parts as follows

$$T = T_{build} + T_{load} + T_{shuffle} + T_{train}. \tag{2}$$

- T_{build} is the time to construct the computation graph, which includes both graph sampling and the GPUs sending part of their computation graphs to the other GPUs following the strategy (i.e., for NFP, SNP, and DNP).
- T_{load} is the time to load the input node features.
- $T_{shuffle}$ is the time to shuffle the hidden embeddings and their gradients across GPUs (i.e., for NFP, SNP, and DNP).
- T_{train} is the time to conduct the training computation.

T_{train} is the same for all strategies since they conduct the same training computation. Thus, we ignore T_{train} when comparing the running time of the strategies. T_{build} is obtained by dry-run, where we assign the GPUs to conduct graph sampling and send the computation graphs following each strategy. The dry-run is cheap for three reasons. First, an epoch of dry-run is sufficient for high quality estimations while actual training requires hundreds of epochs. We conduct dry-run on the PS graph and observe that the top-1% popular graph nodes for two different epochs have an overlap of 94.77%, with an estimation error of the running time only around 5%. Second, the same graph samples are reused during dry-run for different strategies. Third and most importantly, the dry-run only obtains the computation graphs, while the actual feature loading, hidden embedding shuffling, and training computation are not conducted.

It would be expensive to measure T_{load} and $T_{shuffle}$ by actually running the strategies. As they require communication, we collect the communication volume of different operations in the strategies (without actually conducting the communication) during dry-run. We also profile the speed of different communication operators (e.g., GPU to CPU UVA and GPU all-to-all). T_{load} and $T_{shuffle}$ are estimated as the data volume divides the speed of the corresponding communication operator. Denote the input feature dimension as d and hidden dimension as d', we collect the communication volume of T_{load} and $T_{shuffle}$ for each strategy as follows.

- In GDP, the GPUs read input node features from local CPU memory or remote CPU memory (when using multiple machines) but conduct no hidden embedding shuffling. As such, $T_{shuffle}$ is 0, and we estimate the communication volume for T_{load} by recording the numbers of input features the GPUs read from local and remote CPU memory. Note that local GPU-CPU read and cross-machine read are different communication operators and treated separately. NFP, SNP, and DNP use the same method to estimate the communication volume for T_{load}.
- In NFP, all layer-1 destination nodes (denote the count as N_d) require partial hidden embeddings from all C GPUs. The GPUs use an allreduce operator to exchange the hidden embeddings, and each GPU pays $2d'$ communication for each destination node, which includes both embedding in the forward pass and gradient in the backward pass. Thus, the communication volume of $T_{shuffle}$ is $2d'CN_d$.
- In SNP, a layer-1 destination node creates a virtual node on one remote GPU if the GPU holds (one or more) source nodes of the destination node. The GPUs use a sparse all-to-all[2] operator to exchange the hidden embeddings of the virtual nodes, and each virtual node requires $2d'$ communication. Denote the number of virtual nodes for SNP as N_{vs}, the communication volume of $T_{shuffle}$ is $2d'N_{vs}$.

[2]Implemented as a all-to-all operator and a local aggregation operator.

Table 1. Comparing the four parallelization strategies. *G, F, H* refer to sampled subgraph, input node feature, and hidden node embedding. We use ■■■ to □□□ to indicate a level from cost to low, ✔ means require and ✘ means do not require.

Parallel Strategy	Shuffle G	Shuffle F	Shuffle H	Cache Locality	Excess Cache	Partial Aggr.	Graph Partition
GDP	□□□	■■■	□□□	□□□	■■■	✘	✘
NFP	■■■	■□□	■■■	■■■	□□□	✔	✘
SNP	■■□	■□□	■■□	■■■	□□□	✔	✔
DNP	■□□	■■□	■□□	■■□	■■□	✘	✔

- In DNP, a layer-1 destination node creates a virtual node on a remote GPU if the node is managed by a GPU that is different from the GPU holding its seed node. The GPUs use a sparse all-to-all operator to exchange the hidden embeddings, and each virtual node requires $2d'$ communication. Denote the number of virtual nodes for DNP as N_{vd}, the communication volume for $T_{shuffle}$ is $2d'N_{vd}$.

Cache configuration. During dry-run, we collect the access frequency of the graph nodes, i.e., how many times they appear in the sampled subgraphs. If the GPU memory cannot hold all input node features, GDP and NFP store the most popular nodes in the GPU; with SNP, each GPU stores the most popular nodes in its assigned graph partition; with DNP, each GPU considers the nodes in its partition and their 1-hop neighbors to select the most popular nodes. The rationale of the cache rules is to minimize the GPU-CPU communication for feature read. Note that for both SNP and DNP, the seed nodes are assigned to the GPUs according to the graph partition, i.e., each GPU processes the seed nodes in its managing partition to improve locality.

3.3 Trade-offs of the Parallelization Strategies

We compare the four strategies in seven aspects in Table 1 and discuss their trade-offs as follows.

GDP only needs to synchronize the model among the GPUs and thus has the smallest inter-GPU communication. This is advantageous when inter-GPU communication is slow (e.g., using multiple machines). However, the GPU cache locality of GDP is poor, which can lead to a high feature loading cost. This is because (the sampled subgraphs on) a GPU can access K-hop neighbor of its seed nodes, which encompass many nodes but the GPU may cache only a small portion of the node features for large graphs. If each GPU can hold all node features (implying a small graph), GDP has no GPU-CPU communication for feature loading and is the most efficient among all four strategies.

NFP depends on the relation between input feature dimension d and layer-1 hidden dimension d' because it transfers more hidden embeddings to read fewer input features. In particular, if each GPU can cache $1/C$ of the input features, NFP has no GPU-CPU communication to load input feature. If d' is much smaller than d, NFP has a low cost to communicate the layer-1 embeddings and their gradients. NFP does not suit GNN models that use attention (e.g., GAT [45], HGT [16], and CAT [14]). This is because attention requires a source node to have a complete view of all its source nodes.

SNP resembles NFP in that it eliminates GPU-CPU communication for loading input feature when each GPU can host $1/C$ of the input features. SNP also needs to pay extra communication for attention-based GNN models because each GPU may not have all the source nodes for a virtual node. Moreover, both SNP and NFP cannot use excess GPU memory when it is larger than $1/C$ of the input features. An advantage of NFP over SNP is that it does not require graph partitioning, which can be expensive. However, SNP transfers fewer hidden embeddings than NFP.

DNP usually has smaller inter-GPU communication for hidden embeddings than both NFP and SNP because each destination node requires to shuffle at most one hidden embedding. Moreover, DNP can utilize extra memory to reduce the GPU-CPU communication for feature loading since each GPU can cache the nodes in its assigned partition along with their 1-hop neighbors. DNP is also friendly to attention-based models because each GPU has all source nodes for the destination nodes it manages. However, when the GPU cache is small, DNP reads more input features from CPU than SNP because DNP has a larger input node set for each GPU (i.e., the nodes in its partition and their 1-hop neighbors) than SNP (only the nodes in its partition).

4 The APT System

In this part, we first provide an overview of APT and then introduce its unified execution engine.

4.1 System Overview and Workflow

Data layout. APT supports parallel GNN training with both a single and multiple machines. APT assumes the graph topology is replicated on the CPU of every machine such that the GPU workers can use unified memory addressing (UVA) to access it for graph sampling. The node features are usually larger than topology, and APT partitions them to the machines. To train with a single machine, the CPU memory

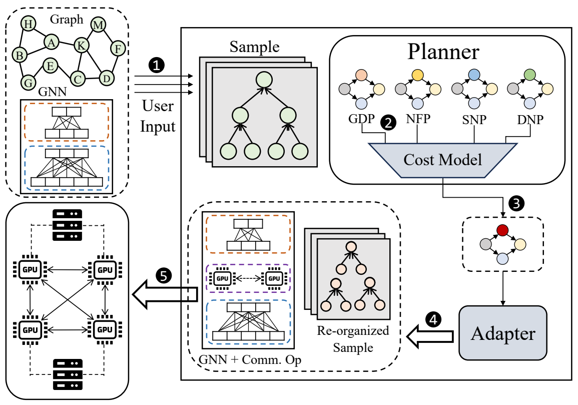


Figure 4. The architecture and workflow of the APT system.

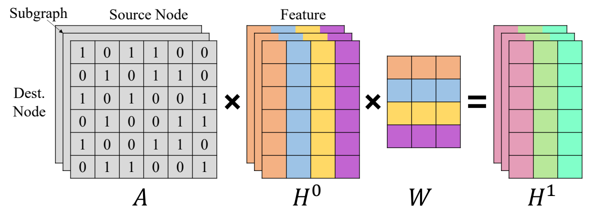


Figure 5. The tensor abstraction of a GNN layer. The four strategies partition the tensors along different dimensions.

hosts all node features; and for M machines, each machine keeps at least $1/M$ of the node features. These requirements ensure that GPU workers can load node features from either local or remote CPU memory. Each GPU worker caches node features in its device memory according to the chosen parallelization strategy to reduce GPU-CPU communication.

Workflow. As shown in Figure 4, to run a GNN training task, APT takes the following steps.

- **Prepare**: The user provides the specifics of the training task to APT, which include the GNN model, data graph, graph sampling algorithm, and hardware configurations. The user can implement the GNN model using DGL [47] or PyG [9]. We provide tools (similar to those in DGL [8]) for users to partition the graph according to the number of machines M and the number of GPUs C. APT also conducts trails to measure the bandwidth of different communication operators (e.g., *alltoall* and *allreduce*) to prepare for the cost models in the next step.
- **Plan**: APT runs the cost models to select the optimal strategy. In particular, each GPU conducts graph sampling for an epoch and sends the communication and computation tasks to the other GPUs according to the sampled subgraphs. This is done for each of the 4 strategies (i.e., GDP, NFP, SNP, and DNP) as they use different task dispatch schemes, and allows the planner of APT to collect the communication volume of different communication operators. Using these dry-run statistics, the planner adopts the cost models to compare the running time of different strategies such that the most efficient strategy can be selected.
- **Adapt**: Given the selected strategy, the adapter configures the execution engine by inserting communication and computation operators into the GNN model at appropriate places, which results in an adapted GNN model that can be executed in parallel over the GPUs. APT also configures data layout for the chosen strategy, which includes the node features to keep on each CPU and GPU. A global feature map is built such that the GPUs know where to fetch their required node features.
- **Run**: APT uses the Pytorch distributed data parallel (DDP) package [29] to launch workers on the GPUs. Then, APT runs the chosen training strategy for the user-specified number of epochs.

We note that APT is general for different graph sampling algorithms and GNN models. As will be seen in Section 4.2, APT treats them as black-boxes in that it only requires graph sampling to produce sampled subgraphs and each GNN layer to conduct computation on a bipartite graphs that consists of source and destination nodes.

4.2 Unified Execution Engine

APT needs an execution engine that can run different strategies but existing GNN systems only support a single strategy (mostly GDP). To implement the execution engine, we observe that different strategies essentially partitions the tensors of a GNN layer along different dimension, and after such partitioning, the workload of each GPU still resembles a GNN layer. As shown in Figure 5, a GNN layer contains dense tensor multiplication (i.e., between H^0 and W) and sparse matrix aggregation for neighbors (i.e., A), where A contains the sampled subgraphs, H^0 is the input node feature, W is the model parameter, and H^1 is the layer-1 output. Different strategies partition the tensors in Figure 5 along different dimensions to the GPUs. In particular, GDP partitions the subgraph dimension of A, NFP partitions the feature dimension of H^0, SNP and DNP partitions the source node and destination node dimensions of A, respectively.

The general abstraction of GNN computation enables us to use DGL (one of the most famous graph learning libraries) as the single GPU execution engine to enjoy its GPU kernel optimizations. However, the partitioning breaks the input format for DGL kernels that run on each GPU. To implement the strategies, APT injects computation and communication operators at DGL kernel barriers to conduct pre- and post-processing to ensure the correct data format. Although the strategies have different logic, they can all be decomposed into four stages, i.e., *Permute*, *Shuffle*, *Execute*, and *Reshuffle*, which we describe as follows.

- **Permute**: APT reorganizes the sampled subgraphs and prepares to shuffle them among the GPUs. In particular, each GPU worker clusters, permutes, and further encodes the sampled subgraphs into a chunk of continuous GPU memory, which is to be sent or broadcast in *Shuffle* stage.
- **Shuffle**: APT communicates the computation graph prepared by *Permute* among the GPUs. Specifically, GDP does nothing in this stage. For SNP and DNP, APT uses a customized Alltoall operator for each GPU to send the source and destination nodes to their managing GPUs for subsequent computation, respectively. SNP also sends the virtual nodes in the computation graph. For NFP, an AllBroadcast operator is called to broadcast the sampled subgraphs such that every GPU receives all layer-1 computation graphs for subsequent execution.
- **Execute**: The GPUs fetches the required node features from a unified feature store and conducts GNN computation with the shuffled subgraphs. GDP is handled by the original DGL kernels. For DNP and SNP, they still follow the logic of DGL's layer computation (i.e., aggregating along the edges between all source and destination nodes). We convert the shuffled subgraphs to match the input format of DGL and utilize DGL kernels. As NFP broadcasts all subgraphs to all GPUs, we conduct both the forward and backward pass jointly for all subgraphs in one batch, using customized SegmentedSpMM and SegmentedSDDMM kernels to ensure correctness.
- **Reshuffle**: The GPUs reshuffle the generated hidden embeddings back to the requesting GPUs and prepare the required data format for the subsequent GNN layer. As GDP does not shuffle subgraphs, it does not perform any operations in this stage. For DNP, we use an Alltoall operator for each GPU to send back its output embeddings. For SNP, a GroupReduce operator is inserted to aggregate the partial hidden embeddings of the virtual nodes from remote GPUs. For NFP, a SparseAllreduce operator is invoked for each destination node to aggregate the partial hidden embeddings from all GPUs.

Unified feature store. The training platform may have a complex memory hierarchy, which includes GPU memory, local and remote CPU memory. APT provides a unified feature store to configure the node features to store in each place and allows the GPUs to find the features to read. As we have discussed in Section 3, each GPU collects its node access frequencies during dry-run and stores the node features that are most likely to be accessed by it. This also applies to the machines, i.e., each machine keeps th node features its GPUs are likely to access in its CPU memory.[3] Note that the cache configuration differ by strategies as they have different access patterns. After configuring the CPU and GPU caches, APT produces a feature map for each GPU, which indicates where the CPU can read a node feature given the node ID. The positions of features are recorded using the rules below.

- The position is the GPU itself if it caches the node. If there are fast inter-GPU links (e.g., NVLinks) and a peer GPU caches the node, the position is the peer GPU.
- If the first check fails and the local CPU stores the node feature, the position is local CPU.
- For the other features, the position is the remote CPU that contains the node feature.

[3]To be rigorous, each machine must store the $1/M$ of node features assigned to it such that all node features are available. Hotness-based caching is conducted using excess CPU memory.

Table 2. Statistics of the graph datasets used for experiments.

Attributes	Papers	Friendster	IGB260M
Abbr.	PS	FS	IM
Vertices	111M	66M	269M
Edges	3.2B	3.6B	3.9B
Feature dimension	128	256	128
Topology size (GB)	24.9	27.4	29.8
Feature size (GB)	52.9	62.6	128

5 Experimental Evaluation

We experiment to answer the following questions:

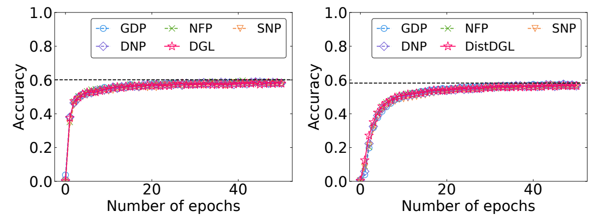
- *How do the specifics of a GNN training task (e.g., dataset, platform, model) affect the performance of the strategies?*
- *What decides the optimal strategy for a GNN training task?*
- *How effective is our APT in selecting the optimal strategy?*

Beside checking APT, our experiments also serve as a comprehensive evaluation of the 4 strategies (i.e., GDP, NFP, SNP, and DNP) under diversified configurations. To our knowledge, this is the first time such experiments are conducted.

5.1 Experiment Settings

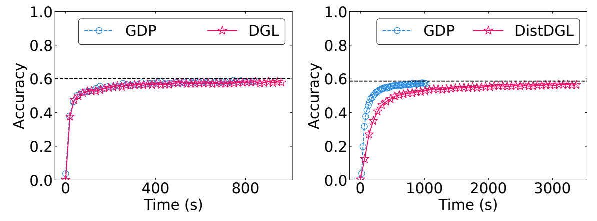
Graphs and models. We use 3 popular and public graph datasets for the experiments, i.e., OGBN-Papers100M [15], Friendster [49] and IGB260M [26]. Their statistics are reported in Table 2. We refer to them by their abbreviations in subsequent descriptions. For GNN models, we use GraphSAGE [12] and GAT [45], which arguably are the most popular. GraphSAGE uses the mean aggregation function, and GAT uses attention for neighbor aggregation. By default, we use 3 layers for the GNN models and conduct node-wise sampling with a fanout of [10, 10, 10] following [47]. GrahSAGE adopts a hidden embedding dimension of 32, and GAT uses a hidden embedding dimension of 8 and 4 attention heads. The mini-batch size of is set as 1024 for each GPU.

Platform. We conduct parallel GNN training using both a single and multiple machines. The single machine has a 96 core Intel Xeon Platinum 8259CL CPU, 378GB main memory, and 8 NVIDIA T4 GPUs (each with 16GB GPU memory).



(a) Single machine **(b)** Four machines

Figure 6. Test accuracy v.s. epoch for GraphSAGE on PS.



(a) Single machine **(b)** Four machines

Figure 7. Test accuracy v.s. time for GraphSAGE on PS.

For multiple machines, we use the same types of CPUs and GPUs as a single machine but run 4 GPUs on each machine. The GPUs on the same machine are connected via PCIe 3.0 to the CPU, and the machines are connected via 100 Gbps Ethernet. The GPU cache size is set to 4GB by default. All the machines run 64 bit Ubuntu 18.04, CUDA v11.8 [37], DGL v1.1.2 [7], and Pytorch v2.0.1 [40]. We are interested in the efficiency of the strategies and thus use average epoch time as the main performance metric.

Sanity check. To check the correctness of our implementation, we compare the 4 strategies of APT with DGL [47] for single-machine training and DistDGL [57] for multi-machine training. Both DGL and DistDGL use GDP. The results in Figure 6 show that all strategies of APT produce the same epoch-accuracy curve as DGL and DistDGL. This is because the strategies are semantically equivalent (i.e., produce an identical model when training for the same number of epochs) and suggest that our implementation is correct. To validate the efficiency of our implementation, we compare the GDP of APT with them in Figure 7. For a single machine, we follow DGL to disable the GPU cache, and the results show that our GDP is as efficient as DGL. For distributed training, our GDP runs faster than DistDGL because we use GPU-based graph sampling, which is faster then the CPU-based graph sampling of DistDGL. Note that the dry-run overhead of APT for strategy selection is low. For a single machine, dry-run takes about 25s while GDP requires 449s to train GraphSAGE to convergence on the PS graph. We cannot compare with NFP (i.e., P3 [10]) and SNP (i.e., Split [39]) implementations because the two systems are not open-source.

Table 3. Node access skewness. We rank the nodes by access frequency and compute the ratio of top nodes in all accesses.

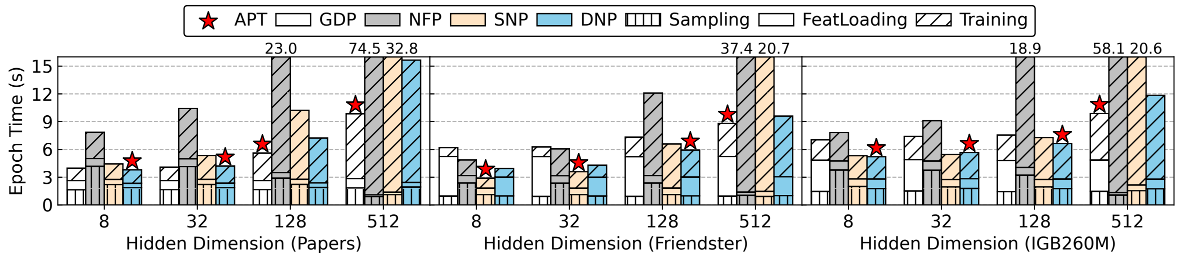
Node Rank	Access Ratio		
	Papers(PS)	Friendster(FS)	IGB260M(IM)
<1%	50.1%	17.7%	31.1%
1%∼5%	34.8%	29.4%	39.0%
5%∼10%	8.8%	19.1%	19.7%
10%∼20%	4.7%	18.8%	9.3%
20%∼50%	1.7%	13.5%	0.9%
50%∼100%	0.0%	1.6%	0.0%

5.2 Main Results

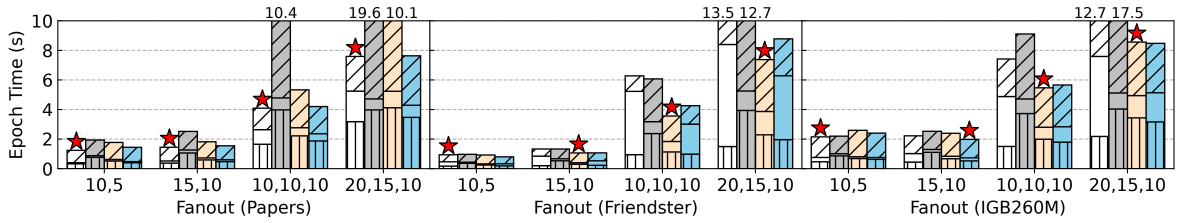
Figure 8 reports the epoch time of the four strategies when training GraphSAGE with 8 GPUs on a single machine. Figure 9 reports the results of distributed training with 16 GPUs on 4 machines. In both figures, we decompose the running time into three parts, i.e., sampling, feature loading, and training, to better understand the influence of different factors. Note that the sampling time includes shuffling the sampled subgraphs among the GPUs, and the training time includes transferring the hidden embeddings. In each case, we mark the strategy selected by APT with a red star.

Hidden dimension. Figure 8a shows that although the epoch time of all strategies increases with hidden dimension, GDP becomes the optimal for all graphs when the hidden dimension is large enough (e.g., 512). This is because GDP is the only strategy that does not shuffle hidden embeddings among the GPUs, and thus its communication cost does not increase with hidden dimension. Among NFP, SNP, and DNP, the epoch time of NFP grows the fastest with hidden dimension because NFP requires every GPU to shuffle a hidden embedding for every layer-1 destination node while SNP and DNP shuffle fewer hidden embeddings. A disadvantage of GDP is that it usually has higher cost for loading the input features than other strategies. When the hidden dimension is small compared to the input feature, other strategies may become the optimal. For instance, with a hidden dimension of 8 and 32, SNP runs the fastest for the FS graph.

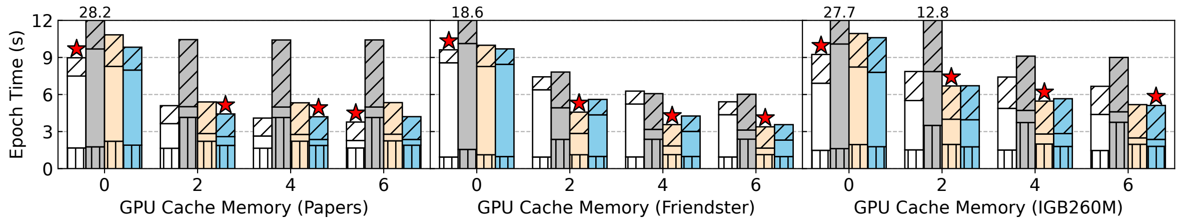
Fanout. Figure 8b reports the results of four fanout configurations for graph sampling, i.e., [10, 5] and [15, 10] for training 2-layer GNN model, and [10, 10, 10] and [20, 15, 10] for training 3-layer GNN model. Recall that fanout controls how many neighbors are sampled for each seed node. The results show that when a small number of neighbors are sampled (e.g., [10, 5]), GDP is usually the optimal. This is because sampling and training are lightweight in this case, which makes the costs of shuffling the sampled subgraphs and hidden embeddings significant for NFP, SNP, and DNP. When many neighbors are sampled (e.g., [10, 10, 10]), different graphs have different optimal strategies, e.g., the PS



(a) Varying hidden dimension



(b) Varying graph sampling fanout



(c) Varying GPU cache size

Figure 8. Epoch time for single machine training with 8 GPUs (best viewed in color).

graph favors GDP while the FS graph likes SNP. This is because the graphs have different *access skewness* for input node features, which we report in Table 3. In particular, we conduct graph sampling using [10, 10, 10], collect the appearance frequencies of the nodes, and compute the frequencies of the top-ranking nodes over all nodes.

For PS, a small portion of nodes dominate the accesses, which means that most input features can be fetched from the GPU cache; in this case, GDP excels because it has a small cost for input feature loading and does not pay the overheads to shuffle hidden embeddings. By contrast, FS has the most scattered node accesses among the three graphs, suggesting that many input features will not appear in the GPU cache; as such, GDP has a long feature loading time, and SNP is the most efficient because it is the most effective in using the GPU cache (i.e., by pushing computation to the input features and shuffling the hidden embeddings). We also notice that DNP generally performs well for the three graphs. This is because DNP serves as a middle ground between GDP and SNP, i.e., compared with GDP, DNP loads fewer input features from CPU but shuffles hidden embeddings among GPUs; compared with SNP, DNP shuffles fewer hidden embeddings but loads more input features. Thus, the DNP proposed by us is a valuable addition to existing strategies.

Cache size. Figure 8c adjusts the cache size on each GPU. The results show that when GPU cache is disabled, GDP is the optimal. This is because all strategies load input features entirely from CPU memory in this case but GDP does not have the overheads of shuffling the sampled subgraphs and hidden embeddings. When GPU cache is enabled, GDP is the optimal for PS and SNP is the optimal for FS due to the node access patterns of the two graphs, which we explained before. For all strategies, increasing the cache size has a diminishing return in reducing the epoch time. This is because larger cache is used to host less popular nodes, and thus the effect of reducing feature loading becomes less significant.

Multiple machines. For Figure 9, we partition the input node features to four machines without overlapping and set the cache size as 4GB on each GPU. Due to the page limit, we only vary the hidden dimension for distributed training because exchanging the hidden embeddings requires inter-machine communication. The results show that GDP and DNP generally perform well. This is because GDP does not shuffle hidden embeddings among the GPUs on different machines, and DNP shuffles fewer hidden embeddings than SNP and NFP. SNP performs much worse than on a single machine because it shuffles many hidden embeddings, and

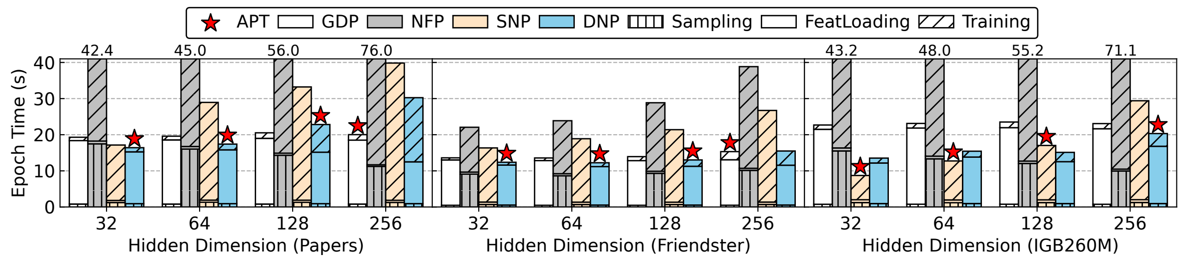


Figure 9. Epoch time for training GraphSAGE with 16 GPUs on 4 machines (best viewed in color).

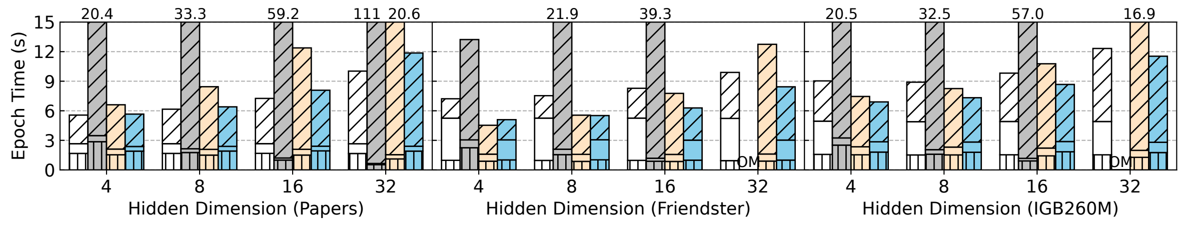


Figure 10. Epoch time for training GAT on a single machine with different hidden dimensions.

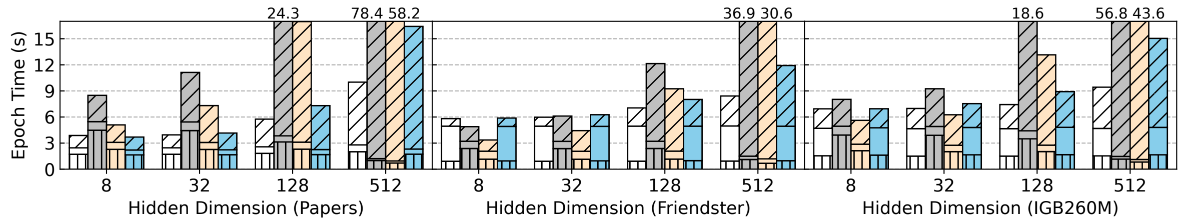


Figure 11. Epoch time for training GraphSAGE on a single machine with random graph partitions.

Table 4. The maximum speedup of using APT for strategy selection compared with always using a single strategy.

Dataset	GDP	NFP	SNP	DNP
Papers(PS)	1.18	7.57	3.33	1.59
Friendster(FS)	2.13	4.25	2.35	1.36
IGB260M(IM)	2.60	5.88	2.09	1.55

inter-machine communication is more expensive than intra-machine. The results suggest that reducing inter-machine communication is important for distributed training, and we conjecture that *hybrid strategies* may run faster. For instance, it is possible to use GDP to coordinate different machines in order to avoid shuffling hidden embeddings among machines, and SNP for the GPUs on each machine to effectively utilize the GPU cache for graphs like FS. We leave analyzing and supporting these hybrid strategies as future work.

Performance of APT. Figure 8 and Figure 9 show that APT selects the optimal or a near-optimal strategy. In Table 4, we report the maximum speedup of using APT over always using a single strategy, and the maximization is conducted over all the configurations we used in Figure 8 and Figure 9 for each dataset. The results show that supporting a single strategy (as in existing systems) is insufficient, and adaptive strategy selection yields significant performance gain.

5.3 Impacts of Other Factors

Attention-based model. Figure 10 shows that GDP and DNP generally perform well for attention-based model (i.e., GAT) while SNP and NFP does not. This is because, with GDP and DNP, each destination node has a complete view of its source nodes and thus does not require extra communication to compute attention. In contrast, SNP and NFP need extra communication because each destination node cannot see all information of its source nodes. We also observe that the intermediate tensors of NFP exceeds GPU memory when the hidden dimension is large. This is because all GPUs store a hidden embedding for every layer-1 destination node.

Graph partition. Following DGL [47], we use METIS to partition the graphs over the GPUs such that each GPU caches the input features it is likely to access. This works if a graph is partitioned on a cheap CPU machine and transferred to expensive GPU machines for training. Figure 11 assumes that high quality graph partitioning is not available and uses random partitioning. The results show that GDP and NFP are not affected but SNP and DNP perform much worse than using METIS partitioning. This is because SNP and DNP rely on a quality graph partitioning for good cache locality.

Cost model accuracy. APT can select the optimal or a near-optimal strategy because our cost models can accurately

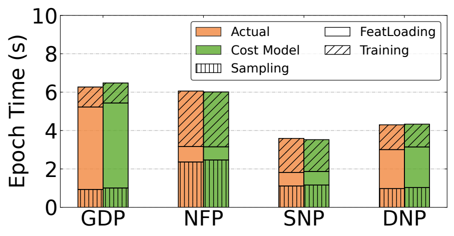


Figure 12. The actual and cost model estimated epoch time for training GraphSAGE on FS with a single machine.

compare the strategies. We illustrate this point in Figure 12, which reports the estimated epoch time produced by the cost models and the actual epoch time. Note that our cost models estimate the hidden embedding shuffling time during training instead of the training time, but we find that it is tricky to isolate the shuffling time. Thus, we run GDP (which does not shuffle during training) to obtain the computation time of training and add our estimated shuffling time to the computation time for Figure 12. The results show that our cost models yield accurate epoch time estimations, and the maximum error is only 5.5% (observed on GDP).

6 Related Work

GNN training systems. DGL [47] and PyG [9] are two predominant systems for GNN training, which support many GNN models [12, 23, 28, 45] and graph sampling algorithms [4, 5, 17, 33, 53, 58]. They mainly focus on kernel optimizations for a single GPU. To break the memory and computation limitations of a single GPU, many systems explore parallel GNN training with multiple GPUs. Early systems like NeuGraph [34], ROC [20], and DGCL [2] adopt full-graph training, which uses all seed nodes in each pass. However, full-graph training suffers from heavy computation and communication for each pass, and many epochs to converge. As such, sampling-based GNN training becomes the mainstream [4, 12, 58], which uses only some seed nodes in each mini-batch and samples neighbors for the seed nodes to reduce computation. Systems like C-SAW [38], NextDoor [19] and gSampler [11] optimize on-GPU sampling, eliminating the bottleneck in complex sampling algorithms on a single GPU. Recent systems that parallelize sampling-based GNN training include DistDGL [57], DSP [3], Quiver [31], and ByteGNN [56], and they usually adopt the graph data parallel (GDP) strategy. Some systems also propose other strategies. For instance, P3 [10] employs node feature parallel (NFP) to reduce the communication for input feature read and eliminate the need of graph partition. GSplit [39] uses source node parallel (SNP) to reduce computation and communication redundancies.

Various strategies are proposed to cache node features on GPU or CPU to reduce communication for GNN training. For instance, PaGraph [30] and Quiver [43] cache nodes with high in-degrees. DSP [3] and Quiver [43] build a unified feature cache for all GPUs and allow the GPUs to read features from peer GPUs via fast interconnections such as NVLink and NVSwitch. GNNLab [50] conducts graph sampling on the data graph to determine the popular nodes to cache.

Existing GNN training systems cover only some of the strategies in APT. We not only propose a new parallel strategy called destination node parallel (DNP), which is usually efficient, but also design a general framework for selecting and executing an efficient strategy for the given GNN training task. Moreover, our APT is orthogonal to existing caching strategies and can easily incorporate them.

Auto-parallelism for DNN. Automatically selecting an efficient parallelization strategy for deep neural network (DNN) training has been extensively explored. The core problem is how to partition the operators of a dense model (e.g., for computer vision, speech, or language) among the devices to achieve a short training time. For instance, OptCNN [21] and Tofu [48] employ dynamic programming to determine the optimal parallel configuration for the operators in a model. FlexFlow [22] formally defines the search space to parallelize a DNN operator and uses a randomized algorithm to reduce strategy search time. TensorOpt [1] improves OptCNN and FlexFlow by considering both training time and memory consumption. PipeDream [36] and GPipe [18] adopt pipeline parallelism to fill up the bubbles in operator parallelism. Our APT is akin to these works as (arguably the first) an auto-parallelism system for GNN training, but GNNs are fundamentally different from DNNs. In particular, the computation and model synchronization is lightweight for GNNs, and the costs for feature loading and hidden embedding shuffling are important. Moreover, the computation graph of GNNs are dynamic with graph sampling.

7 Conclusions

We present the APT system to automatically select and execute efficient parallelization strategies for GNN training tasks. We observe that there is no consist winner among the strategies, and the optimal strategy depends on the specifics of the GNN training task. APT features effective cost models to estimate the running time of the strategies and an unified execution engine that can be configured to run different strategies. Experiment results show that APT usually finds the optimal strategy and significantly outperforms any single fixed strategy. A promising future direction is to develop hybrid strategies that use different schemes for inter-machine and intra-machine communication.

Acknowledgments

We thank the anonymous reviewers for their comments and helpful feedback. This work is supported by Research Grants 8601116, 8601594, and 8601625 from the UGC of Hong Kong. This work is also supported by AWS Shanghai AI Lab.

References

[1] Zhenkun Cai, Xiao Yan, Kaihao Ma, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. 2021. Tensoropt: Exploring the tradeoffs in distributed dnn training with auto-parallelism. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2021), 1967–1981.

[2] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: an efficient communication library for distributed GNN training. In *Proceedings of the Sixteenth European Conference on Computer Systems (Eurosys)*. 130–144.

[3] Zhenkun Cai, Qihui Zhou, Xiao Yan, Da Zheng, Xiang Song, Chenguang Zheng, James Cheng, and George Karypis. 2023. DSP: Efficient GNN training with multiple GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 392–404.

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

[5] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*. 941–949.

[6] Rong Chen, Jiaxin Shi, Haibo Chen, and Binyu Zang. 2015. Bipartite-Oriented Distributed Graph Partitioning for Big Learning. *J. Comput. Sci. Technol.* (2015), 20–29.

[7] DGL. 2023. Deep Graph library. https://www.dgl.ai. [Online; accessed December-2023].

[8] DGL. 2023. DGL Graph Partitioning Tool. https://docs.dgl.ai/guide/distributed-partition.html. [Online; accessed December-2023].

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

[10] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed deep graph learning at scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 551–568.

[11] Ping Gong, Renjie Liu, Zunyao Mao, Zhenkun Cai, Xiao Yan, Cheng Li, Minjie Wang, and Zhuozhao Li. 2023. GSampler: General and Efficient GPU-Based Graph Sampling for Graph Learning. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. 562–578.

[12] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*. 1024–1034.

[13] Kehang Han, Balaji Lakshminarayanan, and Jeremiah Zhe Liu. 2021. Reliable Graph Neural Networks for Drug Discovery Under Distributional Shift. In *NeurIPS 2021 Workshop on Distribution Shifts: Connecting Methods and Applications*.

[14] Tiantian He, Yew Soon Ong, and Lu Bai. 2021. Learning Conjoint Attentions for Graph Neural Nets. In *Advances in Neural Information Processing Systems (NeurIPS)*. 2641–2653.

[15] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: datasets for machine learning on graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[16] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. 2020. Heterogeneous Graph Transformer. In *Proceedings of The Web Conference (WWW)*. 2704–2710.

[17] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive Sampling towards Fast Graph Representation Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*. 4563–4572.

[18] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[19] 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 (Eurosys)*. 311–326.

[20] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of Machine Learning and Systems (MLsys)*. 187–198.

[21] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks.. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*. 2279–2288.

[22] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks.. In *Proceedings of Machine Learning and Systems (MLsys)*. 1–13.

[23] Haitian Jiang, Renjie Liu, Xiao Yan, Zhenkun Cai, Minjie Wang, and David Wipf. 2023. MuseGNN: Interpretable and Convergent Graph Neural Network Layers at Scale. *arXiv preprint arXiv:2310.12457* (2023).

[24] George Karypis. 2011. METIS and ParMETIS. In *Encyclopedia of Parallel Computing*. 1117–1124.

[25] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* (1998), 359–392.

[26] Arpandeep Khatua, Vikram Sharma Mailthody, Bhagyashree Taleka, Tengfei Ma, Xiang Song, and Wen-mei Hwu. 2023. IGB: Addressing The Gaps In Labeling, Features, Heterogeneity, and Size of Public Graph Datasets for Deep Learning Research. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. 4284–4295.

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

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

[29] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proceedings of the VLDB Endowment (VLDB)* (2020), 3005–3018.

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

[31] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. 2023. BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 103–118.

[32] Xin Liu, Mingyu Yan, Lei Deng, Guoqi Li, Xiaochun Ye, and Dongrui Fan. 2022. Sampling Methods for Efficient Training of Graph Convolutional Networks: A Survey. *IEEE CAA J. Autom. Sinica* (2022), 205–234.

[33] Ziqi Liu, Zhengwei Wu, Zhiqiang Zhang, Jun Zhou, Shuang Yang, Le Song, and Yuan Qi. 2020. Bandit Samplers for Training Graph Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[34] 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 (ATC)*. 443–458.

[35] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj D. Kalamkar, Nesreen K. Ahmed, and Sasikanth Avancha. 2021. DistGNN: scalable

distributed training for large-scale graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 76.

[36] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 1–15.

[37] NVIDIA. 2023. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit. [Online; accessed December-2023].

[38] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S. Li, and Hang Liu. 2020. C-SAW: a framework for graph sampling and random walk on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 56.

[39] Sandeep Polisetty, Juelin Liu, Kobi Falus, Yi Ren Fung, Seung-Hwan Lim, Hui Guan, and Marco Serafini. 2023. GSplit: Scaling Graph Neural Network Training on Large Graphs via Split-Parallelism. *CoRR* (2023).

[40] PyTorch. 2023. PyTroch. https://pytorch.org. [Online; accessed December-2023].

[41] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*. 472–488.

[42] Marco Serafini. 2021. Scalable Graph Neural Network Training: The Case for Sampling. *ACM SIGOPS Operating Systems Review* (2021), 68–76.

[43] Zeyuan Tan, Xiulong Yuan, Congjie He, Man-Kit Sit, Guo Li, Xiaoze Liu, Baole Ai, Kai Zeng, Peter Pietzuch, and Luo Mai. 2023. Quiver: Supporting GPUs for Low-Latency, High-Throughput GNN Serving with Workload Awareness. *arXiv preprint arXiv:2305.10863* (2023).

[44] Anton Tsitsulin, John Palowitch, Bryan Perozzi, and Emmanuel Müller. 2024. Graph clustering with graph neural networks. *The Journal of Machine Learning Research* (2024).

[45] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations (ICLR)*.

[46] Daixin Wang, Yuan Qi, Jianbin Lin, Peng Cui, Quanhui Jia, Zhen Wang, Yanming Fang, Quan Yu, Jun Zhou, and Shuang Yang. 2019. A Semi-Supervised Graph Attentive Network for Financial Fraud Detection. In *2019 IEEE International Conference on Data Mining (ICDM)*. 598–607.

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

[48] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference (Eurosys)*. 1–17.

[49] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *12th IEEE International Conference on Data Mining (ICDM)*. IEEE Computer Society, 745–754.

[50] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: a factored system for sample-based GNN training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems (Eurosys)*. 417–434.

[51] 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 (KDD)*. 974–983.

[52] 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 (KDD)*. 974–983.

[53] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2019. Accurate, Efficient and Scalable Graph Embedding. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 462–471.

[54] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*. 5171–5181.

[55] Qingru Zhang, David Wipf, Quan Gan, and Le Song. 2021. A Biased Graph Neural Network Sampler with Near-Optimal Regret. In *Advances in Neural Information Processing Systems (NeurIPS)*. 8833–8844.

[56] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. 2022. ByteGNN: efficient graph neural network training at large scale. *Proceedings of the VLDB Endowment (VLDB)* (2022), 1228–1242.

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

[58] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. 2019. Layer-dependent importance sampling for training deep and large graph convolutional networks. *Advances in neural information processing systems (NeurIPS)* (2019).

A Artifact Information

A.1 Description

This artifact provides the source code of APT (for the paper Adaptive Parallel Training for Graph Neural Networks), including the mentioned framework APIs and an underlying unified execution engine, as well as a comprehensive experimental demonstration for the pros and cons of different parallel strategies. Specifically, all experimental results in this paper are obtained on AWS EC2 g4dn.metal cloud instances, each of which has 96-core Intel Xeon Platinum 8259CL CPU, 378GB main memory, and 8 NVIDIA T4 GPUs (each with 16GB GPU memory).

A.2 Scope

We include the source codes, benchmark scripts and instructions necessary for automatically reproducing Figure 1, Figure 8, Figure 9, Figure 10, and Figure 11.

A.3 Contents

This artifact consists of one repository: the **APT Repository**. This repository includes the source codes of APT and corresponding examples to illustrate its usage.

The implemtation of APT is organized as follows:

- **examples:** This directory contains the example scripts using APT's API, showcasing the usage of the APT's features and functionalities. Moreover, it is the core training module that will be invoked in each benchmark.
- **python:** This python directory includes all the Python source code files. These files include the main contributions of APT's unified framework and model/sampler adapter. It also serves as a wrapper for underlying C++ implementations of APT's unified execution engine.
- **src:** This src directory contains APT's C++ and CUDA codes for distributed computation and communication.
- **tests:** This test directory is for maintaining code quality and ensuring correct functionality through unit tests.
- **third_party:** The third_party directory includes external libraries used in APT, such as nccl, thrust and googletest.

The reproduction for the experiments is organized as follows:

- **figure1:** It contains epoch time comparisons for training GraphSAGE with 8 GPUs on a single machine varying input dim on the Papers dataset and hidden dim on the Friendster dataset.
- **figure8:** It contains epoch time comparisons for training GraphSAGE with 8 GPUs on a single machine varying hidden dim, fanout and GPU cache size.
- **figure9:** It contains epoch time comparisons for training GraphSAGE with 16 GPUs on 4 machines in a distributed environments varying hidden dim.
- **figure10:** It contains epoch time comparisons for training GAT with 8 GPUs on a single machine varying hidden dim.
- **figure11:** It contains epoch time comparisons for training GraphSAGE with 8 GPUs on a single machine using random graph partitions.

A.4 Hosting

APT: https://doi.org/10.5281/zenodo.14567737.

A.5 Running the benchmarks

See the repository for detailed instructions to run the experiments and obtain the result figures.