Optimizing Task Placement and Online Scheduling for Distributed GNN Training Acceleration

Ziyue Luo*, Yixin Bao*, Chuan Wu*
*Department of Computer Science, The University of Hong Kong, Email: {zyluo, yxbao, cwu}@cs.hku.hk

***Abstract*—Training Graph Neural Networks (GNN) on large graphs is resource-intensive and time-consuming, mainly due to the large graph data that cannot be fit into the memory of a single machine, but have to be fetched from distributed graph storage and processed on the go. Unlike distributed deep neural network (DNN) training, the bottleneck in distributed GNN training lies largely in large graph data transmission for constructing mini-batches of training samples. Existing solutions often advocate data-computation colocation, and do not work well with limited resources where the colocation is infeasible. The potentials of strategical task placement and optimal scheduling of data transmission and task execution have not been well explored. This paper designs an efficient algorithm framework for task placement and execution scheduling of distributed GNN training, to better resource utilization, improve execution pipelining, and expediting training completion. Our framework consists of two modules: (i) an online scheduling algorithm that schedules the execution of training tasks, and the data transmission plan; and (ii) an exploratory task placement scheme that decides the placement of each training task. We conduct thorough theoretical analysis, testbed experiments and simulation studies, and observe up to 67% training speed-up with our algorithm as compared to representative baselines.**

I. Introduction

Graph neural networks (GNN) [1][2] generalize deep neural networks (DNN) to learning from graph-structured data and have been exploited in various domains, *e.g.*, computer networking [3], social and biological network analysis [4][5]. GNNs learn high-level graph representations (aka embeddings) by aggregating information from the neighborhood of nodes in a graph, and have shown their superiority in various tasks including node classification [6], graph classification [7] and link prediction [8].

As compared to traditional graph analysis models [9][10], GNNs can capture more complicated features of nodes/edges of large graphs with millions of nodes and billions of edges (*e.g.*, Amazon Product Co-purchasing Network [11], Microsoft Academic Graph [12]). However, training GNNs on large graphs is very resource-intensive and time-consuming. The large graph sizes often exceed the memory and computation capacities of a single device (*e.g.*, GPU) or physical machine, yielding distributed GNN training using multiple devices and machines as the solution. While full-graph training by loading the entire graph into device memory is often infeasible [1], a common practice of distributed GNN training is to do subgraph sampling [13][14] and mini-batch training at each device: samplers select a set of training nodes in the graph, retrieve from graph stores features of (a subset of) several-hop neighbor nodes of each training node to form subgraphs, construct mini-batches with the subgraphs and feed them into workers for training.

A few distributed GNN training frameworks have recently been proposed, *e.g.*, distributed DGL [15], Dorylus [16]. It has been observed that frequent, large graph data transfers exist in distributed GNN training, as mini-batch sampling is carried out in each training iteration, which involves retrieval of subgraphs commonly consisting of hundreds of graph nodes each. Graph data transfer often consumes the majority of time during GNN training (up to 80% of overall training time [15][17]) and renders the performance bottleneck of GNN training, which is different from the common bottlenecks of computation or gradient/parameter communication in DNN training. Careful design to alleviate the graph data transfer overhead is hence the key for distributed GNN training acceleration.

A few efforts have been devoted to minimizing the graph data transfers in distributed GNN training, through static caching [18], min-edge-cut graph partition [19], and data-computation co-location [15]. Even with these schemes, large data transfers between samplers and graph stores may still exist; data-computation co-location may not always be applicable when resource availability varies across machines. On the other hand, strategical task placement, data flow and task execution scheduling to improve resource utilization and execution parallelization, have not been well explored, which can be good complements to the traffic-minimizing schemes for distributed GNN training acceleration.

We focus on optimized planning of distributed GNN training, involving effective placements of training tasks (samplers, workers and parameter servers), near-optimal execution scheduling of the tasks, and data flow transfers. Unique challenges exist in distributed GNN training planning:

First, existing designs largely advocate co-locating a worker with its samplers on the same physical machine, which is only applicable if the computational resources on the machine allow. In a practical machine learning (ML) cluster where resource availability differs across machines, it is non-trivial to plan task placements to minimize data transfer traffic and maximize resource utilization.

Next, optimal scheduling of data transfers and task execution in a distributed GNN training job is complex, falling in the category of strongly NP-hard multi-stage coflow scheduling problems [20]. Further, the data transfer volume between graph

This work was supported in part by grants from Hong Kong RGC under the contracts HKU 17204619, 17208920 and 17207621.

stores and samplers varies according to the graph nodes and their neighbors sampled in each training iteration [13][14] and their storage locations, rendering the scheduling problem an online nature and calling for efficient online algorithm design.

Tackling the challenges, we design an algorithm framework for distributed GNN training planning, comprising two modules: 1) an online scheduling algorithm to strategically set execution time of training tasks and transfer rates of data flows; and 2) an exploratory task placement scheme that decides the placement of each task among available machines. Our goal is to maximize task parallelization while respecting various dependencies, and hence minimize the overall training time of a given GNN model. Our main techniques and contributions are summarized as follows:

▷ Given task placements, we formulate the task and flow scheduling problem for distributed GNN training as an online optimization problem. We design an online scheduling algorithm by effectively overlapping task computation with graph data communication, and adaptively balancing the flow transmission rates among parallel flows into (from) the same machine, to eliminate negative impact of potential communication bottlenecks on the training time. We rigorously analyze the online algorithm and identify a competitive ratio on the training makespan, which is decided by the maximum number of incoming or outgoing flows at any machine in one iteration.

▷ Next, we propose an exploratory task placement scheme based on the Markov Chain Monte Carlo (MCMC) framework [21]. We start by efficient construction of an initial feasible placement in polynomial time. We then introduce a resource violation tolerance factor to encourage full exploration among feasible placements in the solution space. A carefully designed cost function of the placements, defined on the expected training makespan and resource feasibility, guides our search process to the best feasible placement of tasks in arbitrary (heterogeneous) environments, to achieve the minimal expected training time in conjunction with our online scheduling algorithm.

▷ We implement our design atop DGL [22], and conduct thorough testbed experiments and trace-driven simulations. Testbed experiments show that our design achieves significantly lower GNN training time as compared to DistDGL [15] (31.75% on ogbn-products dataset [11] and 22.95% on Reddit dataset [2]) with more efficient network bandwidth utilization. Simulation studies further prove that our design accelerates training up to 67% compared to representative baselines under more diversified training settings, by exploiting strategical task placements to minimize the overall data traffic and maximize the utilization of heterogeneous network bandwidths, maximally overlapping communication with computation, and efficiently scheduling data traffic despite the varying data volumes.

II. Background and Related Work

A. GNN Training

GNNs learn effective graph embeddings by iteratively aggregating neighborhood features (Fig. 1) [23][24]. The derived

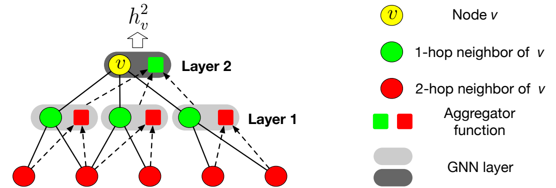


Fig. 1: An example of embedding calculation of node v with a 2-layer GNN: embeddings of v's 1-hop neighbors ($l = 1$) are computed using $h_v^l = \sigma(h_v^{l-1}, \text{AGGREGATE}_{v' \in v\text{'s neighbors}} f(h_v^{l-1}, h_{v'}^{l-1}, e_{v,v'}))$, and then aggregated to derive v's embedding, h_v^2, using the formula with $l = 2$. $\sigma(\cdot)$ and $f(\cdot)$ are trainable parameterized functions, $e_{v,v'}$ is the edge between v and v', and AGGREGATE is an aggregator function (*e.g.*, mean, min, max).

embeddings can be further processed (*e.g.*, using DNN layer, softmax operation), to produce decisions for downstream tasks (*e.g.*, node classification, link prediction).

To construct a mini-batch for GNN training, a set of training nodes are sampled from the input graph, and their L-hop neighbors are used for embedding generation by a L-layer GNN. Using features of all L-hop neighbors of the selected training nodes may lead to GPU/CPU memory overflow or high computation complexity. A common practice is to recursively sample neighbors of each training node with a sampling algorithm (*e.g.*, [13][14]), and a sub-graph is formed among the training node and its sampled L-hop neighbors. Each sub-graph with its features renders one sample in the mini-batch.

Using mini-batches of graph samples, GNN training is similar to DNN training: forward propagation is carried out to compute a loss, and then backward propagation to derive gradients of the GNN model parameters based on the loss, using an optimization algorithm (*e.g.*, stochastic gradient descent); a gradient update operation follows, which involves gradient aggregation among workers in distributed training and application of updated parameters to the GNN model.

B. Distributed GNN Training Systems

Deep Graph Library (DGL) [22] is a package built for easy implementation of GNN models on top of DL frameworks (*e.g.*, PyTorch [25], MXNet [26]). The recent release of DGL supports distributed GNN training on relatively large graphs. It uses random sampling, collocates one worker with one graph store, and does not pipeline GNN training across iterations, leaving a large room for further performance improvement. Euler [27] is integrated with TensorFlow [28] for GNN training, which partitions a large graph in a round-robin manner and splits feature retrieving requests to allow concurrent transmissions; large data transfers still exist due to its locality-oblivious graph partition. AliGraph [29] adopts distributed graph storage, optimized sampling operators and runtime to efficiently support GNNs. PyTorch Geometric [30] is a deep learning library on irregularly structured input data such as graphs, supporting multi-GPU training on a single machine only. Dorylus [16] distributes GNN training over serverless cloud function threads on CPU servers, requiring specialized functions provided by AWS [31]. Large data traffic exists

in these systems, and careful transfer scheduling and task deployment can enhance them for training time minimization.

C. Distributed Training Acceleration

NeuGraph [32] and PaGraph [18], which train GNN models on a single machine, adopt full-graph training by loading entire graphs into GPU memory, and are hence only feasible for training over small graphs. Considering multi-server clusters, ROC [33] splits the input graph over multiple GPUs or machines to achieve workload balance, and adopts a memory management scheme to reduce CPU-GPU data transfer. DistDGL [15] alleviates network transfer in distributed GNN training by co-locating each worker with its samplers on the same server, and partitioning the input graph with a minimum edge cut method. Further, various graph partition, sampling and caching methods have been proposed for enhancing distributed GNN training [2][34][35]. These studies focus on minimizing data transfer volumes across devices/machines. Optimization of task placement and execution scheduling is orthogonal to the existing efforts, and our solution can complement them to fully accelerate distributed GNN training. DGCL [36] is a recently proposed communication library for distributed GNN training, which decides data routing strategy for every graph node to the requiring worker(s), considering the detailed interconnection topology among workers. Its detailed communication plan is re-computed before every training epoch, which may incur substantial overhead for large graphs. Our design performs efficient, polynomial-time online scheduling on both task execution and data flows between tasks, effectively reducing the overall training time.

Task placement, computation and communication scheduling have been studied for DNN training on non-graph data [37][38][39]. The communication scheduling deals with arranging transmission time and order of gradient/parameter tensors for parameter synchronization [38][39]. Placement studies focus on worker placement to minimize interference [40] instead of proximity to data, and DNN operator placement to achieve model parallelism [41]. Computation scheduling deals with fine-grained operator execution ordering, in case of model- or pipeline-parallel DNN training [42][43][44]. Compared to distributed DNN training, GNNs are largely trained with data parallelism, incurring large graph data communication that blocks the computation and occupies a majority of the training time (up to 80% [17]). Instead of operator-level placement and scheduling of a GNN model, we study placement of tasks (samplers, workers and parameter servers), overlap both graph data transfer and tensor communication with computation (the graph data traffic is magnitudes larger than tensor transfers), and pipeline mini-batch training across training iterations, which are all dedicated for GNN training acceleration.

III. Problem Model

A. Distributed GNN Training System

We train a GNN model (with L embedding layers) in a cluster of M physical machines. Partitions of a large graph used for GNN training are stored on the M machines. Each machine $m \in [M]^1$ is equipped with R types of computational resources (*e.g.*, GPU, CPU and memory), with type-r resource available at the amount of C_m^r. Let B_{in}^m (B_{out}^m) represent the available incoming (outgoing) NIC bandwidth on machine m.

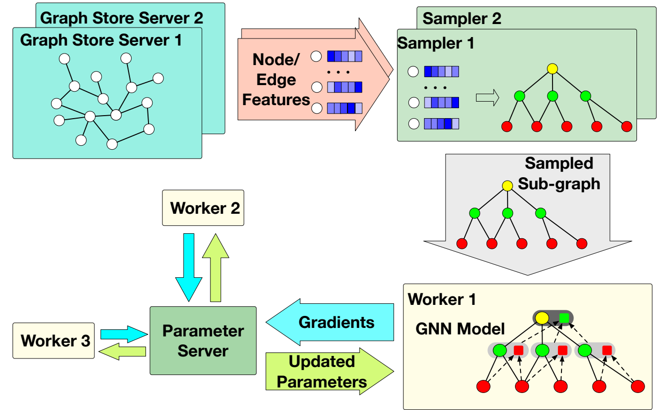


Fig. 2: Distributed GNN training workflow

There are four types of tasks in our GNN training job: (1) *Graph store server:* Each machine hosts a graph store server, to maintain one graph partition (including graph structure and node/edge features). (2) *Sampler:* Each sampler selects training nodes, retrieves sampled node/edge features from graph store servers and forms sub-graphs. (3) *Worker:* Each worker carries out forward and backward computation, pushes gradients to and pulls parameters from parameter servers for parameter synchronization. A worker is typically associated with one or multiple samplers, which supply mini-batches dedicatedly to the worker. (4) *Parameter server (PS)*: PSs aggregate gradients from all workers, update the GNN model parameters and distribute updated parameters to all workers.

We use J_g, J_s, J_w and J_{ps} to represent the sets of graph store servers, samplers, workers and PSs, respectively, in the training job. We suppose the number of each type of tasks is specified by the ML developer: the number of graph stores is M (as each machine hosts exactly one graph partition), the number of workers can be larger or smaller than M (considering a machine may host multiple GPUs and CPUs, and a worker typically consumes one GPU or CPU), the number of samplers to serve each worker is usually fixed (*e.g.*, 2 samplers per worker). Let $J = J_g \cup J_s \cup J_w \cup J_{ps}$ denote the set of all tasks. Each task $j \in J$ occupies a w_j^r amount of type-r resource, $\forall r \in [R]$. For example, graph store servers, samplers and PSs are commonly run on CPUs, while workers can run on GPUs [18] or CPUs [15], and consume the respective memory. Tasks of the same kind (*e.g.*, all samplers) occupy the same amount of resources. Let p_j denote the execution time of task j in each iteration.

In a training iteration, each sampler selects a number of training nodes from the input graph and signals the graph store servers to acquire neighbor information. Upon requests from a sampler, a graph store server samples among L-hop neighbors of the training nodes that it hosts (using a given sampling algorithm), and sends the node/edge features back to the sampler.

[1] $[X]$ denotes set $\{1, 2, \ldots, X\}$

The sampler then sends sub-graph samples to its associated worker, which form a mini-batch from samples supplied by its sampler(s), for forward and backward computation. Computed gradients are sent from workers to the PSs and then updated parameters are dispatched from PSs to workers. The workflow is illustrated in Fig. 2.

B. Problem Formulation

We target overall training time minimization in our distributed GNN training job. Our design space includes two subproblems.

1) **Task Placement:** We decide placements of all tasks in the GNN training job on the machines, to maximize task parallelization and minimize communication traffic. We use binary variable y_j^m to indicate task placement: y_j^m equals 1 if task j is deployed on machine m, and 0, otherwise. The placement constraints are:

$$\sum_{m\in[M]} y_j^m = 1, \forall j \in J \tag{1}$$

$$\sum_{j\in J} w_j^r y_j^m \leq C_m^r, \forall m \in [M], r \in [R] \tag{2}$$

$$y_j^m = 1, \forall j \in J_g, j \text{ is placed on machine } m \tag{3}$$

$$y_j^m \in \{0, 1\}, \forall j \in J, m \in [M] \tag{4}$$

Constraints in (1) ensure that every task is placed on one and only one machine. (2) are resource capacity constraints on the machines. (3) specifies the given placements of graph store servers on machines. Fig. 3(a) shows an example task placement of a GNN training job on two machines.

2) **Online Execution and Flow Scheduling:** Suppose it takes N iterations for the GNN model training to converge. Given task placements, we decide the start time of each task and transmission schedules of sampled data and tensor flows, in each training iteration. Let binary variable $x_{j,n}^t$ indicate the start time of task j in iteration n: $x_{j,n}^t$ is 1 if task j in iteration n starts at time t, and 0, otherwise. We use $k_{(j,n)\to(j',n')}^t$ to denote the amount of traffic sent from task j of iteration n to task j' of iteration n' at time t, including the following cases: sampled graph data from a graph store server to a sampler or from a sampler to a worker in the same iteration, gradients from a worker to a PS, or parameters updated at a PS (j) in iteration-n training to a worker (j') for iteration-$(n+1)$ training ($n' = n + 1$).

The execution schedule should respect execution dependencies among tasks and flows, as follows:

$$x_{j,1}^1 = 1, \forall j \in J_g \tag{5}$$

$$\sum_{t\in[T]} x_{j,n}^t = 1, \forall j \in J, n \in [N] \tag{6}$$

$$\min\{t | k_{(j,n)\to(j',n')}^t > 0, t \in [T]\} \geq \sum_{t\in[T]} t x_{j,n}^t + p_j, \forall j \in J,$$
$$n \in [N], (j', n') \in succ(j, n), j \text{ and } j' \text{ are on different servers} \tag{7}$$

$$\max\{t | k_{(j,n)\to(j',n')}^t > 0, t \in [T]\} < \sum_{t\in[T]} t x_{j',n'}^t, \forall j \in J,$$
$$n \in [N], (j', n') \in succ(j, n), j \text{ and } j' \text{ are on different servers} \tag{8}$$

$$\sum_{t\in[T]} t x_{j,n}^t + p_j \leq \sum_{t\in[T]} t x_{j',n'}^t, \forall j \in J, n \in [N],$$
$$(j', n') \in succ(j, n), j \text{ and } j' \text{ are on the same server} \tag{9}$$

We ignore the training node selection time at a sampler, and message passing from a sampler to a graph store server for graph data requests, as the traffic volume is negligible. Constraint (5) indicates that graph store servers run first to sample neighbors. (6) ensures that each task in each training iteration is scheduled once. Here T is a potentially large time span in which our GNN training converges.

Among tasks and flows, there are the following execution dependencies: (i) a sampler can start after receiving data from all graph store servers in each iteration; (ii) in iteration n, a worker can start after receiving a mini-batch of graph data from its samplers and model parameters updated in iteration $n - 1$; (iii) a PS can start after receiving gradients from all workers, computed in this iteration. We call (j', n') a successor of (j, n) if task j' in iteration n' can only start after receiving data from task j in iteration n, and $succ(j, n)$ denotes the set of all successors of (j, n). Constraint (7) specifies that transmission from (j, n) to its successor (j', n') starts after (j, n) is done. (8) ensures that task j' in iteration n' does not start before the transfer from (j, n) to (j', n') is completed, if tasks j and j' do not reside on the same machine. We ignore data passing time between tasks on the same machine, but specify execution dependency among those tasks in (9).

Across training iterations, we require that task j in iteration $n+1$ can only start after task j's execution in iteration n has been done (*e.g.*, a sampler prepares training data for iteration n before those for iteration $n + 1$), and data transfer $(j, n + 1) \to (j', n' + 1)$ cannot start before transmission $(j, n) \to (j', n')$ has been completed. These inter-iteration dependencies are formulated as in (10) and (11):

$$\sum_{t\in[T]} t x_{j,n}^t + p_j \leq \sum_{t\in[T]} t x_{j,n+1}^t, \forall j \in J, n \in [N-1] \tag{10}$$

$$\max\{t | k_{(j,n)\to(j',n')}^t > 0, t \in [T]\} <$$
$$\min\{t | k_{(j,n+1)\to(j',n'+1)}^t > 0, t \in [T]\}, \forall j \in J, n \in [N-1],$$
$$(j', n') \in succ(j, n), j \text{ and } j' \text{ are placed on different servers} \tag{11}$$

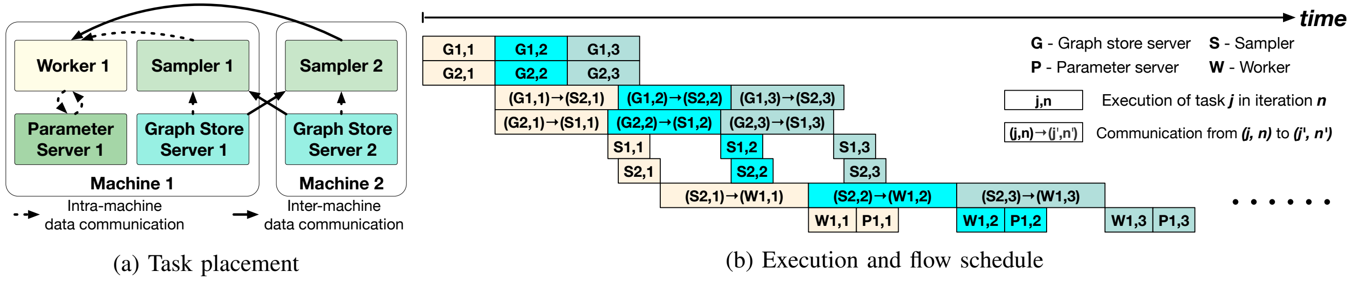
Further, the following constraint specifies the total traffic transmitted from task j in iteration n to task j' in iteration n', as denoted by $d_{(j,n)\to(j',n')}$. The traffic volume is decided according to whether it is graph data transfer from a graph store server to a sampler or from a sampler to a worker (decided by the graph sampling algorithm in use), or gradient/parameter tensor transfer between a worker and a PS (depending on the GNN model size).

$$\sum_{t\in[T]} k_{(j,n)\to(j',n')}^t = d_{(j,n)\to(j',n')}, \forall j \in J, n \in [N],$$
$$(j', n') \in succ(j, n), j \text{ and } j' \text{ are placed on different servers} \tag{12}$$

The total incoming (outgoing) traffic at machine m should not exceed its available bandwidth at each time t:

$$\sum_{n\in[N]} \sum_{j\in J: y_j^m = 1} \sum_{(j',n')\in succ(j,n): y_{j'}^m = 0} k_{(j,n)\to(j',n')}^t \leq B_{out}^m,$$
$$\forall m \in [M], t \in [T] \tag{13}$$

$$\sum_{n\in[N]} \sum_{j\in J: y_j^m = 0} \sum_{(j',n')\in succ(j,n): y_{j'}^m = 1} k_{(j,n)\to(j',n')}^t \leq B_{in}^m,$$
$$\forall m \in [M], t \in [T] \tag{14}$$



(a) Task placement

(b) Execution and flow schedule

Fig. 3: A distributed GNN training planning example: a job with 2 graph store servers, 1 worker with 2 samplers, and 1 PS.

We aim at minimizing the makespan of all N iterations of GNN training, which is computed as $\max_{t\in T, j\in J}\{tx^t_{j,N}+p_j\}$. Given task placements $\{y^m_j\}$, the execution and flow scheduling problem is formulated as:

$$\min \max_{t\in T, j\in J}\{tx^t_{j,N}+p_j\} \tag{15}$$

subject to:

$$(5)\text{–}(14)$$

$$x^t_{j,n}\in\{0,1\}, \forall j\in J, n\in[N], t\in[T] \tag{16}$$

$$k^t_{(j,n)\to(j',n')}\geq 0, \forall j\in J, n\in[N], t\in[T],$$
$$(j',n')\in succ(j,n), j \text{ and } j' \text{ are placed on different servers} \tag{17}$$

Problem (15) is a generalization of the strongly NP-hard multi-stage coflow scheduling problem (MSCSP) [20], by grouping transmission between the same types of tasks in one iteration as one coflow (*e.g.*, data transmission from all graph store servers to all samplers). In addition, the key challenge with our problem lies in the unknown graph data volume transferred between graph store servers and samplers: graph sampling is typically a random algorithm [13], the training nodes and their neighbors selected vary from one training iteration to the next, and hence the sizes of node/edge features to transfer change and are unknown beforehand. Consequently, our execution scheduling is an online problem.

In the following, we first design an online algorithm for task execution and flow transmission schedule, assuming task placements are given; next, we devise the task placement scheme that minimizes the total training time in conjunction with scheduling. An example task and flow schedule is given in Fig. 3(b), where we depict task execution and flow communication for the first three training iterations, based on the task placement in Fig. 3(a). Each training iteration is denoted using a different color.

Key notation is summarized in Table I for ease of reference.

TABLE I: Notation

Symbol	Description
T	total time span
J	set of all tasks
$J_g/J_s/J_w/J_{ps}$	set of graph store servers/samplers/workers/PSs
M	# of machines
N	# of training iterations
R	# of resource types
C^r_m	available amount of type-r resource on machine m
B^m_{in} (B^m_{out})	avail. incoming (outgoing) bandwidth of machine m
p_j	execution time of task j in one iteration
$d_{(j,n)\to(j',n')}$	amount of traffic transmitted from task j of iteration n to its successor task j' of iteration n'
w^r_j	type-r resource demand of task j
y^m_j	task j is placed on machine m (1) or not (0)
$x^t_{j,n}$	task j of iteration n starts at t (1) or not (0)
$k^t_{(j,n)\to(j',n')}$	amount of traffic transmitted from task j of iteration n to its successor j' of iteration n' at t

IV. Online Execution and Flow Scheduling

A. Scheduling Algorithm

Given placements $\{y^m_j\}$, we design an online algorithm that decides start time of each task ($x^t_{j,n}$) and flow transmission ($k^t_{(j,n)\to(j',n')}$) over time.

We maintain two flow sets: (i) F_{act}, that stores every active flow $(j,n)\to(j',n')$ which currently has started but not finished transmission yet; (ii) F_{pend}, to store every pending flow $(j,n)\to(j',n')$ whose predecessor task (j,n) has been done, and that has not started because its predecessor flow $(j,n-1)\to(j',n'-1)$ in the previous iteration has not completed transmission yet. For each task (j,n), we use $\mathcal{F}(j,n)$ to represent the set of flows that originate from (j,n) to tasks that reside on other machines (than where j is).

Our online scheduling algorithm is in Alg. 1. We start by running graph store server processing for the first training iteration at $t=1$ (line 2). Then at each time t, we run every task that has received all required data and hence is available to execute (line 7). For each task (j,n) completed at $t-1$, consider every flow $(j,n)\to(j',n')\in\mathcal{F}(j,n)$ in t: if the flow's predecessor flow $(j,n-1)\to(j',n'-1)$ is in F_{act} or F_{pend} (indicating it not done yet), we add $(j,n)\to(j',n')$ to F_{pend}; otherwise, it is scheduled to transmit in t and added to F_{act} (lines 8-13). In addition, for every flow $(j,n)\to(j',n')$ ended at $t-1$, we check if its successor flow $(j,n+1)\to(j',n'+1)$ is in F_{pend}: if so, we move it from F_{pend} to F_{act} and start the flow transmission (lines 14-17). For every flow $(j,n)\to(j',n')$ which transfers in t, supposing j placed on m and j' on m', we set its traffic volume $k^t_{(j,n)\to(j',n')}$ at t to $\min\{B^{m'}_{in}/\Delta^{m'}_{in}, B^m_{out}/\Delta^m_{out}\}$ (lines 18-21). Δ^m_{in} (Δ^m_{out}) is the *ingress flow degree* (*egress flow degree*) on machine m, counting the number of active flows entering and exiting from m, respectively:

$$\Delta^m_{in}=|\{(j',n')\to(j,n)|(j',n')\to(j,n)\in F_{act}, y^m_j=1\}| \tag{18}$$

$$\Delta^m_{out}=|\{(j,n)\to(j',n')|(j,n)\to(j',n')\in F_{act}, y^m_j=1\}| \tag{19}$$

In this way, we balance flow rates among flows going into and out of each machine, ensuring no individual flow becoming the

Algorithm 1: Online Execution Scheduling - **OES**

Input: $T, J, M, N, \{y_j^m\}$
Output: $\{x_{j,n}^t\}, \{k_{(j,n)\rightarrow(j',n')}^t\}, T_{OES}$

1 Initialize F_{act} and F_{pend} to $\emptyset$
2 $x_{j,1}^1 \leftarrow 1, \forall j \in J_g$
3 **for** $t \in [T]$ **do**
4   **if** *every* $(j, N), j \in J$ *is done (aka training has converged)* **then**
5     $T_{OES} \leftarrow t - 1$ **break**
6   **for** $(j, n) \in \{(j, n) | j \in J, n \in [N]\}$ **do**
7     $x_{j,n}^t \leftarrow 1$ if (j, n) is available
8   **if** (j, n) *finished at* $t - 1$ **then**
9   **for** $(j, n) \rightarrow (j', n') \in \mathcal{F}(j, n)$ **do**
10   **if** $(j, n-1) \rightarrow (j', n'-1) \in F_{act} \cup F_{pend}$ **then**
11   add $(j, n) \rightarrow (j', n')$ to F_{pend}
12   **else**
13   add $(j, n) \rightarrow (j', n')$ to F_{act}
14   **for** *every flow* $(j, n) \rightarrow (j', n')$ *finished at* $t - 1$ **do**
15   **if** $(j, n+1) \rightarrow (j', n'+1) \in F_{pend}$ **then**
16   remove $(j, n+1) \rightarrow (j', n'+1)$ from F_{pend}
17   add $(j, n+1) \rightarrow (j', n'+1)$ to F_{act}
18   **for** $m \in [M]$ **do**
19   calculate Δ_{in}^m and Δ_{out}^m according to (18) (19)
20   **for** $(j, n) \rightarrow (j', n') \in F_{act}$ **do**
21     $k_{(j,n)\rightarrow(j',n')}^t \leftarrow \min\{B_{in}^{m'}/\Delta_{in}^{m'}, B_{out}^m/\Delta_{out}^m\}$, where $y_j^m = 1$ and $y_{j'}^{m'} = 1$
22 **return** $\{x_{j,n}^t\}, \{k_{(j,n)\rightarrow(j',n')}^t\}, T_{OES}$

bottleneck. The algorithm terminates when the whole training process is done, *i.e.*, all tasks of the last training iteration are completed (lines 4-5).

B. Theoretical Analysis

Let F_{one_iter} denote the set of all inter-machine flows in one training iteration, including the transfer of updated parameters computed in this iteration from PS to workers. We define the *one-iteration ingress flow degree* $\widehat{\Delta_{in}^m}$ and *one-iteration egress flow degree* $\widehat{\Delta_{out}^m}$:

$$\widehat{\Delta_{in}^m} = |\{(j', n') \rightarrow (j, n) | (j', n') \rightarrow (j, n) \in F_{one_iter}, y_j^m = 1\}|$$
$$\widehat{\Delta_{out}^m} = |\{(j, n) \rightarrow (j', n') | (j, n) \rightarrow (j', n') \in F_{one_iter}, y_j^m = 1\}|$$

and the *maximum degree* Δ:

$$\Delta = \max_{m \in [M]} \{\max\{\widehat{\Delta_{in}^m}, \widehat{\Delta_{out}^m}\}\} \quad (20)$$

which represents the maximum number of incoming or outgoing flows at any machine in one training iteration.

Theorem 1. *The overall training makespan achieved by Alg. 1,* T_{OES}*, is no larger than* Δ *times the optimal objective value* T^* *of the offline execution scheduling problem (15), i.e., the competitive ratio of the online algorithm in Alg. 1 is* Δ*.*

Proof sketch. Following our schedule, we can identify a chain, O, consisting of dependent tasks and flows that are executed sequentially without waiting in-between, respecting the execution dependencies. We ensure the execution of chain O covers the whole training makespan. Further, our online schedule guarantees that the minimum flow rate of any flow in O is no less than dividing the maximum flow rate of any flow in O in the offline optimal solution by Δ. Detailed proof of Theorem 1 and all missing proofs are in our technical report.

V. Exploratory Task Placement

We adopt the Markov Chain Monte Carlo (MCMC) search framework [21] to identify a good placement solution to minimize the training makespan with our online scheduling Alg. 1. We start by constructing a feasible initial placement solution, $\mathcal{Y}_0 = \{y_j^m\}_0$, followed by generating a sequence of placements $\mathcal{Y}_1, \mathcal{Y}_2, \ldots$, until a time budget I is exhausted.

A. Constructing Initial Feasible Placement

A feasible task placement solution should respect resource capacity constraints in (2). We first randomly order the M machines into $\{m_1, m_2, \ldots, m_M\}$. Note that placements of graph store servers are given (one on a machine). Let $[q_s, q_w, q_{ps}, i]$ indicate that we can pack q_s samplers, q_w workers and q_{ps} PSs within the first i machine (m_1 to m_i) without violating resource capacities, and (q_s, q_w, q_{ps}, i) be a particular partial placement of putting q_s samplers, q_w workers and q_{ps} PSs on machine m_i. We use $\mathcal{A}_{q_s,q_w,q_{ps},i}$ to denote an exact placement associated with $[q_s, q_w, q_{ps}, i]$, specifying how many samplers, workers and PSs are placed in each of the i machines, to make up for the total numbers of q_s, q_w and q_{ps}. Let $\Omega(i)$ be the set of all $[q_s, q_w, q_{ps}, i]$'s with i fixed and $q_s \in [|J_s|], q_w \in [|J_w|], q_{ps} \in [|J_{ps}|]$.

We use dynamic programming to construct a feasible placement solution. We first consider all feasible placements $(q_s, q_w, q_{ps}, 1)$ on m_1. Let η_s denote the maximal number of samplers that can be hosted by any machine, *i.e.* $\eta_s = \max_{m \in [M]} \min_{r \in [R]: w_j^r > 0} \lfloor C_m'^r / w_j^r \rfloor$, any $j \in J_s$ ($C_m'^r$ is available type-r resource on m excluding that occupied by the graph store server). Similarly, we can define an upper bound on the number of workers and PSs per machine, η_w and η_{ps}. For every possible combination of $q_s \in \{0\} \cup [\min\{|J_s|, \eta_s\}]$, $q_w \in \{0\} \cup [\min\{|J_w|, \eta_w\}]$ and $q_{ps} \in \{0\} \cup [\min\{|J_{ps}|, \eta_{ps}\}]$, we check if the capacity constraints on m_1 are satisfied. For every feasible solution found, we add $[q_s, q_w, q_{ps}, 1]$ to $\Omega(1)$, and set $\mathcal{A}_{q_s,q_w,q_{ps},1} = \{(q_s, q_w, q_{ps}, 1)\}$.

Next, we iteratively construct $\Omega(i)$ based on $\Omega(i-1)$ until finding a complete feasible solution of placing all $|J_s|$ samplers, $|J_w|$ workers and $|J_{ps}|$ PSs onto the machines. For each $[q_s, q_w, q_{ps}, i-1] \in \Omega(i-1)$, we examine whether $|J_s| - q_s$ samplers, $|J_w| - q_w$ workers and $|J_{ps}| - q_{ps}$ PSs can be fit into machine m_i. If so, we have identified a complete feasible placement solution that packs all tasks within the first i machines: $\mathcal{A}_{solution} = \mathcal{A}(q_s, q_w, q_{ps}, i-1) \cup \{(|J_s| - q_s, |J_w| - q_w, |J_{ps}| - q_{ps}, i)\}$. Otherwise, we find every feasible placement (q_s', q_w', q_{ps}', i) with $q_s' \in \{0\} \cup [|J_s| - q_s]$,

Algorithm 2: Exploratory Task Placement - **ETP**

```
   Input: T, J, M, R, {C_m^r}, {w_j^r}
   Output: Y_min
1  Y_0 <- IFS(J, M, R, {C_m^r}, {w_j^r}); Y_min <- Y_0
2  _, _, min_makespan = OES(T, J, M, N, Y_0)
3  for z ∈ {0, 1, ..., I − 1} do
4  |  randomly select a task j from J \ J_g
5  |  construct M_avail of j
6  |  randomly select a machine m from M_avail
7  |  construct new placement solution Y'
8  |  _, _, T'_Y' = OES(T, J, M, N, Y')
9  |  π(Y_z → Y') <-
   |    min{1, exp(βcost(Y_z) − βcost(Y'))}
10 |  if rand() ≤ π(Y_z → Y') then
11 |  |  Y_{z+1} <- Y'
12 |  |  if no resource violation with Y_{z+1} and
   |  |    T'_Y' < min_makespan then
13 |  |  |  min_makespan <- T'_Y'
14 |  |  |  Y_min <- Y_{z+1}
15 |  else
16 |  |  Y_{z+1} <- Y_z
17 return Y_min
```

$q'_w \in \{0\} \cup [|J_w| - q_w]$, and $q'_{ps} \in \{0\} \cup [|J_{ps}| - q_{ps}]$ that satisfies capacity constraint on machine m_i; and if $[q_s + q'_s, q_w + q'_w, q_{ps} + q'_{ps}, i]$ is not in $\Omega(i)$ yet, we add it into $\Omega(i)$, and set $A(q_s + q'_s, q_w + q'_w, q_{ps} + q'_{ps}, i)$ to be the union of $A(q_s, q_w, q_{ps}, i-1)$ and $\{(q'_s, q'_w, q'_{ps}, i)\}$. We build from $\Omega(2)$ to $\Omega(M)$ and return the first complete feasible placement solution. The above algorithm for finding the initial feasible placement is referred to as **IFS**.

Theorem 2. ***IFS** identifies a feasible placement solution within polynomial time.*

Proof sketch. Every feasible placement corresponds to exact one $[q_s, q_w, q_{ps}, i]$ tuple. The construction of each $\Omega(i)$ takes up to $O(|J_s||J_w||J_{ps}|\eta_s\eta_w\eta_{ps}R)$ time, and through the construction of $\Omega(1)$ to $\Omega(M)$, **IFS** explores all possible tuples in polynomial time and identifies one feasible solution.

B. Searching for Better Placements

Starting from the initial feasible placement, we iteratively search for better placement solutions, according to a $cost(\mathcal{Y})$ defined on the overall training makespan of placement $\mathcal{Y}$.

Practically, task placements should be decided before training starts and remain fixed during training (to avoid substantial overhead of VM/container migration and flow redirection). The online nature of execution scheduling is due to size variation of sampled graph data; we should identify a placement that works best in expectation of the traffic variation. To this end, we profile task execution time and inter-task traffic volumes by running the GNN training for some iterations (50 as in our evaluation), and produce their distributions. We simulate the training process under placement $\mathcal{Y}$ using Alg. 1, with time and traffic volume drawn from the distributions, and derive the training makespan $T'_{\mathcal{Y}}$. The cost of placement $\mathcal{Y}$ is:

$$cost(\mathcal{Y}) = T'_{\mathcal{Y}}(1 + \sum_{m\in[M], r\in[R]} \max\{\frac{\sum_{j\in J} w_j^r y_j^m - C_m^r}{C_m^r}, 0\}) \quad (21)$$

where $T'_{\mathcal{Y}}$ is multiplied by 1 plus a penalty term for resource violation (computed as the sum of capacity violation percentages over all types of resources and all machines).

Our search explores the solution space by transferring from one placement $\mathcal{Y}_z$ to another $\mathcal{Y}_{z+1}$, for a total of I transfers (the time budget). Give $\mathcal{Y}_z$, we uniformly randomly sample a task $j \in J \setminus J_g$. Let M_{avail} denote the set of machines other than the one where j is placed in $\mathcal{Y}_z$, which can host j adhering to relaxed resource capacity constraints:

$$\sum_{j\in J} w_j^r y_j^m \le (1+\mu)C_m^r, \forall m \in [M], r \in [R] \quad (22)$$

Here, the capacity constraints are relaxed by a μ factor to allow full exploration in the placement space. For example, when the violation factor μ is set to 100% (default in our evaluation), every feasible solution can be identified if an infinite time budget I is allowed: Setting μ to 100% is equivalent to allowing a duplicate set of machines (*i.e.*, each machine has doubled its resource capacities). Therefore, we can transit from any feasible placement $\mathcal{Y}$ to any other feasible $\hat{\mathcal{Y}}$ by moving each task from its placement in $\mathcal{Y}$ to the duplicate of the machine where it is placed in $\hat{\mathcal{Y}}$. The new placement on the set of duplicate machines is feasible since $\hat{\mathcal{Y}}$ is a feasible placement solution. If the computational resources of all machines are quite sufficient to host the training tasks, we can set μ to a smaller value for better search efficiency.

Next, we uniformly randomly choose one server $m \in M_{avail}$ to move j to, and come up with the new placement solution $\mathcal{Y}'$. We compute a probability ($\beta > 0$ is a hyper-parameter set to 0.1 in our evaluation, whose smaller value increases the tendency of our search process to jump out of local optima):

$$\pi(\mathcal{Y}_z \to \mathcal{Y}') = \min\{1, \exp(\beta cost(\mathcal{Y}_z) - \beta cost(\mathcal{Y}'))\} \quad (23)$$

With probability $\pi(\mathcal{Y}_z \to \mathcal{Y}')$, we use $\mathcal{Y}'$ as $\mathcal{Y}_{z+1}$: if $cost(\mathcal{Y}') \le cost(\mathcal{Y}_z)$, we accept $\mathcal{Y}'$ as $\mathcal{Y}_{z+1}$ (probability is 1); otherwise, we still accept $\mathcal{Y}'$ as the next state with probability $\pi(\mathcal{Y}_z \to \mathcal{Y}')$ (for exploration) and maintain $\mathcal{Y}_{z+1}$ the same as $\mathcal{Y}_z$ with probability $1 - \pi(\mathcal{Y}_z \to \mathcal{Y}')$.

Our state transition as designed above ensures that the probability of visiting $\mathcal{Y}$ is linear to $\exp(-\beta cost(\mathcal{Y}))$ [21], *i.e.*, solutions with lower costs are more frequently visited than ones with larger costs. We return the best feasible placement found after I transitions, which does not violate any original resource capacity constraints in (2), and leads to the minimum (simulated) training time as compared to all other feasible placements visited. Alg. 2 summarizes our exploratory task placement algorithm (**ETP**).

C. Complete Distributed GNN Training Planning Algorithm

Our complete distributed GNN training planning (*DGTP*) algorithm is given in Alg. 3. We first leverage Alg. 2 to identify

Algorithm 3: Distributed GNN Training Planning (*DGTP*)

Input: $T, N, J, M, R, \{C_m^r\}, \{w_j^r\}$
Output: $\mathcal{Y}_{min}, \{x_{j,n}^t\}, \{k_{(j,n)\to(j',n')}^t\}$
1 $\mathcal{Y}_{min} \leftarrow$ **ETP**$(T, J, M, R, \{C_m^r\}, \{w_j^r\})$
2 $\{x_{j,n}^t\}, \{k_{(j,n)\to(j',n')}^t\}, T_{OES} \leftarrow$ **OES**$(T, J, M, N, \mathcal{Y}_{min})$
3 **return** $\mathcal{Y}_{min}, \{x_{j,n}^t\}, \{k_{(j,n)\to(j',n')}^t\}$

the best placement $\mathcal{Y}_{min}$ and then use Alg. 1 to decide the task and flow schedules $\{x_{j,n}^t\}, \{k_{(j,n)\to(j',n')}^t\}$ based on $\mathcal{Y}_{min}$, in an online manner.

VI. Performance Evaluation

We evaluate *DGTP* by both testbed experiments and simulation studies.

A. Testbed Experiments

Implementation. We implement *DGTP* using Python on DGL 0.6.1 [22] and PyTorch 1.8.1 [25] with 1056 LoC for the training system and 1522 LoC for the search and scheduling algorithms. Parameter synchronization through a PS is built on PyTorch. We use the Stochastic Fairness Queueing provided by tc qdisc [45] to control flow transmission rates according to our online scheduling algorithm, dynamically assigning ongoing data flows into separate queues and ensuring fairness among them with negligible scheduling overhead.

Testbed. Our testbed consists of 4 GPU servers interconnected by a Dell Z9100-ON switch, with 50Gbps peak bandwidth between any two servers. Each server is equipped with one 50GbE NIC, one 8-core Intel E5-1660 CPU, two GTX 1080Ti GPUs and 48GB DDR4 RAM. To emulate resource heterogeneity, we use tc to limit the bandwidth capacity of two servers to 10Gbps.

GNN model and datasets. We train one representative GNN model (three layers of hidden size 256), GraphSage [2], on two graph datasets: ogbn-products [11] (an Amazon product copurchasing graph) and Reddit [2] (consisting of Reddit posts within a month and connections between posts if the same user comments on both posts). We implement uniformly random sampling of neighbors of training nodes, with different fanouts (the number of neighbors to sample) at different hops, set according to the official training script provided by the DGL team and other existing studies [2][15]. Same as in *DistDGL* [15], we set the mini-batch size on both datasets to 2000 (subgraphs). We use Adam optimizer [46] with a learning rate of 0.001 during the training.

TABLE II: Benchmark datasets

Dataset	#Nodes	#Edges	Feature Vector Length	Fan-out
ogbn-products	2.4M	61.8M	100	5, 10, 15
Reddit	0.2M	114.6M	602	5, 10, 25
ogbn-papers100M	0.1B	1.6B	128	12, 12, 12

We use 4 graph store servers, 6 workers (each requiring 3GB memory, 1 logical CPU core and 1 GPU card), and 1 PS (requiring 5GB memory, 1 logical CPU core) to train the GNN model. We partition the input graph with METIS partitioner [19] among graph stores. Each worker is associated with two samplers (each requiring 7GB memory, 2 logical CPU cores). We profile data to drive our search algorithm over 50 iterations of training on each dataset.

Baseline. We compare *DGTP* with a modified version of *DistDGL* [15] that enables inter-server communication between a worker and its samplers. *DistDGL* adopts a placement scheme that maximally colocates each worker with its associated samplers within one server, and uses the system default scheduling strategy (running a task when ready and sending data in FIFO queues).

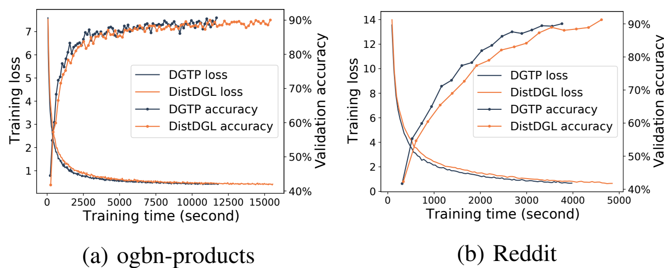
End-to-end training performance. We compare the end-to-end training convergence time between *DGTP* and *DistDGL*. The offline search to obtain *DGTP*'s task placements can be done within 5 minutes (we only simulate 20 iterations of GNN training to obtain $T'_{\mathcal{Y}}$ during search, and set the exploratory time budegt to 10000). Fig. 4 shows the training progress to achieve a 90% target accuracy over the validation sets. *DGTP* outperforms *DistDGL* by 31.75% in terms of the overall training time on ogbn-products, and 22.95% on Reddit.

Resource usage. We also examined resource usage during training. We observed similar GPU, CPU and memory usage between *DGTP* and *DistDGL*, as task execution in both systems is constantly blocked by the large data transfers. Fig. 5 shows the bandwidth usage on the four servers. We observe that *DGTP* has a much better network usage on both datasets: *DGTP* can identify task placements that exploit the heterogeneous bandwidth levels well, while communication in *DistDGL* is often bottlenecked on the low-bandwidth inter-server connections (two pairs of its worker and samplers have to be separated onto different servers due to non-sufficient resources on the same servers).

B. Simulation Studies

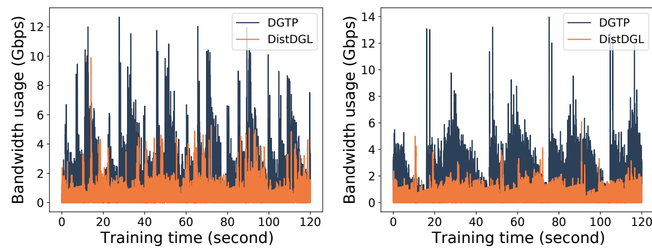
Settings. We further evaluate *DGTP* in detail by simulating the training of the GraphSage model on: 1) ogbn-products on 8 machines using 8 graph store servers, 16 workers each with 2 samplers, and 1 PS; and 2) ogbn-papers100M (Microsoft Academic Graph dataset described in Table II) on 16 machines using 16 graph store servers, 20 workers each with 4 samplers, and 1 PS. We simulate 5 epochs of training (*i.e.*, each sampler goes through the whole set of training nodes specified by the dataset for five times) on ogbn-products (actual training of GraphSage on ogbn-products converges in 5 epochs, as we observed in our experiment), and 25 epochs on ogbn-papers100M (convergence time according to ogbn-papers100M official leaderboard [47]). Our simulation is driven by profiled data collected by training the model on the respective datasets in our testbed.

We consider four types of resources on each machine: memory, CPU, GPU and network bandwidth. The available memory size on each machine is set within $[32, 128]$GB, the number of available CPU cores between $[4, 16]$, the number of available GPUs within $[1, 4]$, and network bandwidth among $\{10Gbps, 20Gbps, 50Gbps\}$.



(a) ogbn-products (b) Reddit

Fig. 4: Training loss & validation accuracy: *DGTP* vs. *DistDGL*



(a) ogbn-products (b) Reddit

Fig. 5: Total network bandwidth usage: *DGTP* vs. *DistDGL*

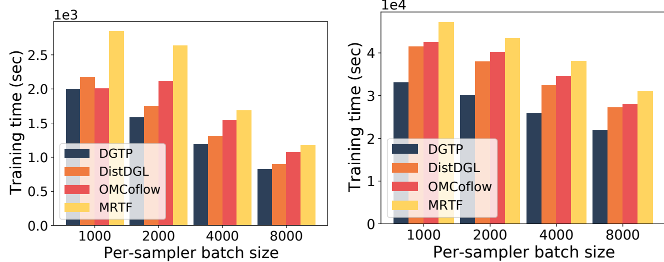


Fig. 6: Training time on ogbn-products: different batch sizes

Fig. 7: Training time on ogbn-papers100M: diff. batch sizes

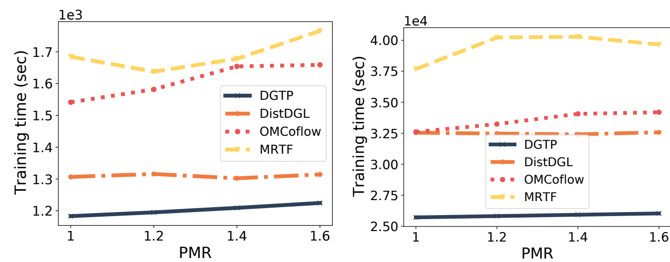


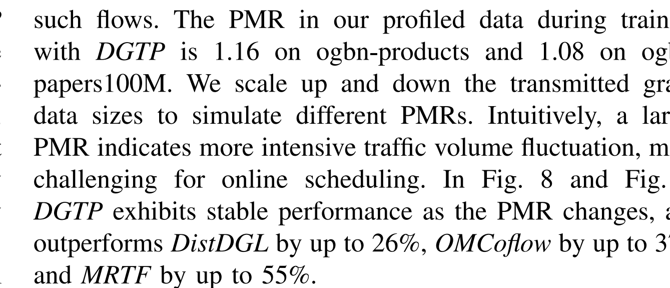
Fig. 8: Training time on ogbn-products: different PMRs

Fig. 9: Training time on ogbn-papers100M: different PMRs

Baselines. Apart from *DistDGL*, we further compare *DGTP* with two flow scheduling schemes (in which we use the same placements as computed by *DGTP* and a task starts immediately once its dependencies have been cleared): (i) *OMCoflow*, a state-of-the-art online coflow scheduling algorithm [48] that groups flows to the same task as one coflow, and sets the flow rates in each coflow inversely proportional to predicted flow finish time (supposing it is the only coflow in the network); (ii) *MRTF*, which schedules flows according to the minimum remaining time first (MRTF) heuristic.

Different per-sampler batch sizes. A larger per-sampler batch size (a worker's mini-batch size divided by the number of samplers it uses) results in larger sampling data traffic, potentially yielding more communication overhead when poorly planned. As Fig. 6 and Fig. 7 show, *DGTP* outperforms all three baselines, reducing the training makespan on ogbn-products by up to 11%, and on ogbn-papers100M by up to 25%, compared to *DistDGL*. Larger data traffic is incurred for training on ogbn-papers100M due to the larger fan-outs, and its training environment is more complex (with more servers, resource heterogeneity, etc.). We identify *DGTP*'s larger speed-up on ogbn-papers100M is because *DGTP* can find better task placements that reduce the overall data traffic during training and schedule the traffic over the complex network environment well. Further, *DGTP* achieves up to 33% less training time as compared to *OMCoflow*, and up to 67% to *MRTF*, on the two datasets. The advantage of *DGTP* improves with batch size. These indicate that *DGTP* can efficiently schedule flow transfers to minimize the overall training time in an online manner.

Different peak-to-mean ratios. We compute a peak-to-mean ratio (PMR) for flows from graph store servers to samplers, as the maximum data flow rate between any (graph store server, sampler) pair divided by the average flow rates among all such flows. The PMR in our profiled data during training with *DGTP* is 1.16 on ogbn-products and 1.08 on ogbn-papers100M. We scale up and down the transmitted graph data sizes to simulate different PMRs. Intuitively, a larger PMR indicates more intensive traffic volume fluctuation, more challenging for online scheduling. In Fig. 8 and Fig. 9, *DGTP* exhibits stable performance as the PMR changes, and outperforms *DistDGL* by up to 26%, *OMCoflow* by up to 37% and *MRTF* by up to 55%.



VII. Conclusion

This paper designs efficient placement and scheduling algorithms for distributed GNN training over heterogeneous clusters. We propose a competitive online execution algorithm that schedules training task execution and flow transfers for both graph data sampling and parameter synchronization. We also design an explorative algorithm to decide the placement of every task, which, in conjunction with online task/flow scheduling, minimizes the overall training makespan. According to testbed experiments, our design reduces the end-to-end training time by up to 31.75% as compared to a state-of-the-art distributed GNN training solution. Simulation studies further demonstrate that our design significantly outperforms representative schemes by minimizing the total data traffic and maximizing the bandwidth usage through task placement, and strategically scheduling tasks and flows to overlap computation with communication and reduce total communication time. Our design can be easily extended to GNN training with AllReduce-based parameter synchronization, by considering detailed communication flows within AllReduce operations. In the case of multiple GNN training jobs on the same cluster, our algorithms can be adopted for jointly searching for task placements of all jobs, and for online task and flow scheduling for each of the jobs.

REFERENCES

[1] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," in *Proc. of ICLR*, 2017.

[2] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," in *Proc. of NIPS*, 2017.

[3] K. Rusek, J. Suárez-Varela, P. Almasan, P. Barlet-Ros, and A. Cabellos-Aparicio, "RouteNet: Leveraging Graph Neural Networks for network modeling and optimization in SDN," *IEEE Journal on Selected Areas in Communications*, 2020.

[4] C. Li and D. Goldwasser, "Encoding Social Information with Graph Convolutional Networks for Political Perspective Detection in News Media," in *Proc. of ACL*, 2019.

[5] D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional Networks on Graphs for Learning Molecular Fingerprints," in *Proc. of NIPS*, 2015.

[6] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph Attention Networks," in *Proc. of ICLR*, 2018.

[7] F. Errica, M. Podda, D. Bacciu, and A. Micheli, "A Fair Comparison of Graph Neural Networks for Graph Classification," in *Proc. of ICLR*, 2020.

[8] X. Li, Y. Shang, Y. Cao, Y. Li, J. Tan, and Y. Liu, "Type-Aware Anchor Link Prediction across Heterogeneous Networks Based on Graph Attention Network," in *Proc. of AAAI*, 2020.

[9] P. Frasconi, M. Gori, and A. Sperduti, "A General Framework for Adaptive Processing of Data Structures," *IEEE Transactions on Neural Networks*, 1998.

[10] S. Cao, W. Lu, and Q. Xu, "GraRep: Learning Graph Representations with Global Structural Information," in *Proc. of ACM CIKM*, 2015.

[11] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open Graph Benchmark: Datasets for Machine Learning on Graphs," in *Proc. of NeurIPS*, 2020.

[12] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B.-J. Hsu, and K. Wang, "An Overview of Microsoft Academic Service (MAS) and Applications," in *Proc. of WWW*, 2015.

[13] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "GraphSAINT: Graph Sampling Based Inductive Learning Method," in *Proc. of ICLR*, 2020.

[14] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling," in *Proc. of ICLR*, 2018.

[15] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, "DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs," in *IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms*, 2020.

[16] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim *et al.*, "Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads," in *Proc. of USENIX OSDI*, 2021.

[17] S. Gandhi and A. P. Iyer, "P3: Distributed Deep Graph Learning at Scale," in *Proc. of USENIX OSDI*, 2021.

[18] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, "PaGraph: Scaling GNN Training on Large Graphs via Computation-aware Caching and Partitioning," in *Proc. of ACM SoCC*, 2020.

[19] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, 1998.

[20] B. Tian, C. Tian, H. Dai, and B. Wang, "Scheduling Coflows of Multi-stage Jobs to Minimize the Total Weighted Job Completion Time," in *Proc. of IEEE INFOCOM*, 2018.

[21] C. J. Geyer, "Practical Markov Chain Monte Carlo," *Statistical Science*, 1992.

[22] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma *et al.*, "Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[23] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural Message Passing for Quantum Chemistry," in *Proc. of ICML*, 2017.

[24] S. Cai, L. Li, J. Deng, B. Zhang, Z.-J. Zha, L. Su, and Q. Huang, "Rethinking Graph Neural Architecture Search from Message-passing," in *Proc. of IEEE/CVF CVPR*, 2021.

[25] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Proc. of NeurIPS*, 2019.

[26] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," in *NIPS Workshop on Machine Learning Systems (LearningSys)*, 2016.

[27] (2021) Euler Graph Library. [Online]. Available: https://github.com/alibaba/euler

[28] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," in *Proc. of USENIX OSDI*, 2016.

[29] K. Zhao, W. Xiao, B. Ai, W. Shen, X. Zhang, Y. Li, and W. Lin, "AliGraph: An Industrial Graph Neural Network Platform," in *Proc. of SOSP Workshop on AI Systems*, 2019.

[30] M. Fey and J. E. Lenssen, "Fast Graph Representation Learning with PyTorch Geometric," in *Proc. of ICLR*, 2019.

[31] *AWS Lambda*, 2021. [Online]. Available: https://aws.amazon.com/lambda

[32] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, "NeuGraph: Parallel Deep Neural Network Computation on Large Graphs," in *Proc. of USENIX ATC*, 2019.

[33] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with ROC," in *Proc. of MLSys*, 2020.

[34] L. Wang, Q. Yin, C. Tian, J. Yang, R. Chen, W. Yu, Z. Yao, and J. Zhou, "FlexGraph: A Flexible and Efficient Distributed Framework for GNN Training," in *Proc. of ACM EuroSys*, 2021.

[35] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks," in *Proc. of ACM KDD*, 2019.

[36] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu, "DGCL: An Efficient Communication Library for Distributed GNN Training," in *Proc. of ACM EuroSys*, 2021.

[37] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters," in *Proc. of USENIX ATC*, 2017.

[38] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-Based Parameter Propagation for Distributed DNN Training," in *Proc. of Systems and Machine Learning (SysML)*, 2019.

[39] S. Shi, X. Chu, and B. Li, "MG-WFBP: Merging Gradients Wisely for Efficient Communication in Distributed Deep Learning," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[40] Y. Bao, Y. Peng, and C. Wu, "Deep Learning-based Job Placement in Distributed Machine Learning Clusters," in *Proc. of IEEE INFOCOM*, 2019.

[41] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device Placement Optimization with Reinforcement Learning," in *Proc. of ICML*, 2017.

[42] S. Wang, D. Li, and J. Geng, "Geryon: Accelerating Distributed CNN Training by Network-level Flow Scheduling," in *Proc. of IEEE INFOCOM*, 2020.

[43] J. H. Park, G. Yun, M. Y. Chang, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y.-r. Choi, "HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism," in *Proc. of USENIX ATC*, 2020.

[44] X. Yi, S. Zhang, Z. Luo, G. Long, L. Diao, C. Wu, Z. Zheng, J. Yang, and W. Lin, "Optimizing Distributed Training Deployment in Heterogeneous GPU Clusters," in *Proc. of ACM CoNEXT*, 2020.

[45] M. A. Brown, "Traffic Control HOWTO," *Guide to IP Layer Network*, 2006.

[46] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[47] *GraphSAGE_res_incep*, 2021. [Online]. Available: https://github.com/mengyangniu/ogbn-papers100m-sage

[48] H. Tan, S. H.-C. Jiang, Y. Li, X.-Y. Li, C. Zhang, Z. Han, and F. C. M. Lau, "Joint Online Coflow Routing and Scheduling in Data Center Networks," *IEEE/ACM Transactions on Networking*, 2019.