Optimizing Task Placement and Online Scheduling for Distributed GNN Training Acceleration in Heterogeneous Systems

Ziyue Luo, *Member, IEEE,* Yixin Bao, and Chuan Wu, *Senior Member, IEEE*

***Abstract*—Training Graph Neural Networks (GNNs) 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 and heterogeneous training devices in heterogeneous clusters. 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 in heterogeneous systems, to better resource utilization, improve execution pipelining, and expedite 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 48% training speed-up with our algorithm as compared to representative baselines in our testbed settings.**

***Index Terms*—Distributed Machine Learning Systems, Graph Neural Network, Online Scheduling.**

I. Introduction

Graph neural networks (GNNs) [1][2], built on top of deep neural networks (DNNs), are capable to learn the high-level representations of graph-structured data, through iterative aggregation of node's neighboring information. GNN has shown its success in various graph-related tasks, *e.g.*, node classification [3], graph classification [4] and link prediction [5].

As compared to traditional graph analysis models [6][7], 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 [8], Microsoft Academic Graph [9]). 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 [10][11] 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. Workers synchronize the trained model parameters with each other through either parameter servers [12] or the AllReduce operation [13].

A few distributed GNN training frameworks have recently been proposed, *e.g.*, distributed DGL [14], PyG [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 [14][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 [14]. 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.

Moreover, a machine learning (ML) cluster commonly consists of GPUs of different models, with thousands of ML jobs running simutaneously [20]. A newly arrived ML training job often faces the dilemma that available GPUs of its desired model are not enough while GPUs of other models are available, calling for ML training with heterogeneous devices. However, distributed GNN training in heterogeneous systems

This work was supported in part by grants from Hong Kong RGC under the contracts HKU 17207621, 17203522, and C7004-22G (CRF). A preliminary version of this work appeared as "Optimizing Task Placement and Online Scheduling for Distributed GNN Training Acceleration" in *Proc. of IEEE INFOCOM, 2022, pp. 890-899*.

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

has not been fully explored.

We focus on optimized planning of distributed GNN training in a heterogeneous system, 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 ML cluster, resource availability and computation capability differ across machines. It is non-trivial to make training task placement decisions to minimize data transfer traffic and maximize GNN training efficiency.

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 [21]. Further, the data transfer volume between graph stores and samplers varies according to the graph nodes and their neighbors sampled in each training iteration [10][11] 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. This further inspires our task placement scheme design to find the placements achieving the minimum expected training time scheduled by our online algorithm.

▷ Next, we propose an exploratory task placement scheme based on the Markov Chain Monte Carlo (MCMC) framework [22]. 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. Inspired by our online scheduling algorithm, we design a placement cost function, defined on the critical path length in an execution graph constructed based on our scheduling algorithm and the placement. The carefully defined cost fuction 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 [23], 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 [14] (up to 48% on ogbn-products dataset [8]) with more efficient network bandwidth utilization. Simulation studies further demonstrate the effectiveness of our design, accelerating training up to 67% compared to representative baselines across various 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

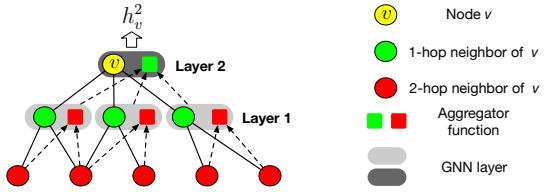


Fig. 1: An example of embedding calculation of node v with a 2-layer GNN: embeddings of v's 1-hop neighbors ($l = 1$) are computed using $h_v^l = \sigma(h_v^{l-1}, \text{AGGREGATE}_{v' \in v\text{'s neighbors}} f(h_v^{l-1}, h_{v'}^{l-1}, e_{v,v'}))$, and then aggregated to derive v's embedding, h_v^2, 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).

GNNs learn effective graph embeddings by iteratively aggregating neighborhood features (Fig. 1) [24][25]. The derived 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 an 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.*, [10][11]), 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) [23] is a package built for easy implementation of GNN models on top of DL frameworks (*e.g.*, PyTorch [26], MXNet [27]). The recent release of DGL, DistDGL [14], supports distributed GNN training on relatively large graphs. It uses random sampling, colocates one worker with one graph store, and does not pipeline GNN training across iterations, leaving a large room for further performance improvement. Euler [28] is integrated with TensorFlow [29] 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 [30] adopts distributed graph storage, optimized sampling operators and runtime to efficiently support GNNs. PyTorch Geometric [15] is a deep learning library designed for irregularly structured input data such as graphs. It supports multi-GPU training on a single machine only, and users are advised to implement their own graph store servers and samplers for training large-scale graphs beyond a single machine. Dorylus [16], on the other hand, distributes GNN training across serverless cloud function threads on CPU servers. It optimizes for cost effectiveness and relies on specialized functions provided by AWS [31]. However, Dorylus introduces asynchronous computation into the training process, which can negatively impact convergence performance. Notably, all of these designs, except for DGL, do not optimize the placement of tasks within a distributed GNN training system. 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 [14] 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. P^3 [17] adopts hybrid parallel strategies for distributed GNN training, colocating the first GNN layer with the graph data and then redistributing activations for later layers' computation in a data-parallel manner. Further, a number of works have optimized the distributed GNN system from the perspectives of graph partition [34], caching [35], *etc*. 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. It focuses on full-graph training (*i.e.*, training with the whole graph without sampling), and would require communication plan re-computation per epoch if directly adopted into sampling-based training.

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.

D. Communication Scheduling for Parallel Flows

The minimization of communication cost for parallel data flows in distributed systems has been the subject of extensive research in several works. RAPIER [45] presents an online coflow (*i.e.*, parallel flows) optimization framework for data-intensive coflows. It makes joint decisions on coflow scheduling and routing to minimize the average coflow completion time. Chowdhury *et al.* [46] investigate coflow scheduling across geo-distributed data centers. They propose a randomized 2-approximation algorithm for coflows with polynomially sized release times and demands, and a $(2+\epsilon)$-approximation design for coflows where release times and demands are super-polynomial. Cheng *et al.* [47] propose NEAL, a cross-layer approach that amalgamates application-layer data locality scheduling with network-layer coflow scheduling. NEAL models the data placement and coflow scheduling problem as a mixed integer linear program, and employs a meta-heuristic approach for scheduling decisions. However, all the aforementioned works primarily focus on optimizing coflows consisting of a single stage (*i.e.*, no dependencies between flows), whereas our problem involves complex dependencies between tasks and flows, which significantly complicates the scheduling and data placement process. Tian *et al.* [21] study the scheduling of multi-stage coflows and propose an approximation algorithm based on a relaxed linear formulation of the problem. Nevertheless, they only consider offline problems where the flow rates are known in advance. Shafiee *et al.* [48] design an offline algorithm for scheduling coflow jobs with directed acyclic graph structures. They propose a randomized Delay-and-Merge algorithm, which randomly delays the start time of each job and greedily merges delayed jobs to maximize the networks. However, distributed GNN training introduces

dependent parallel flows with varying flow rates due to its random node sampling process, which cannot be known in advance. Despite this, we have developed an online task execution and flow scheduling algorithm that delivers competitive performance.

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. Besides heterogeneous bandwidth levels among machines, computation capabilities on different machines can differ, *e.g.*, machines with different GPU and CPU models.

There are several types of tasks in our distributed 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, and synchronizes model parameters with a parameter synchronization scheme. A worker is typically associated with one or multiple samplers, which supply mini-batches dedicatedly to the worker. We use J_g, J_s and J_w to represent the sets of graph store servers, samplers and workers, respectively, in the training job.

For model parameter synchronization among workers, we consider two popular synchronization schemes: the parameter server (PS)-based [12], and the Ring AllReduce operation-based [13] parameter synchronization. For PS-based parameter synchronization, we leverage another type of task, *parameter server*, to aggregate gradients from all workers, update the GNN model parameters and distribute updated parameters to all workers. Let J_{ps} denote the set of PSs in a PS-based training job. For Ring AllReduce-based parameter synchronization, J_{ps} equals $\emptyset$ for the Ring AllReduce-based training job. A ring is formed among workers during the Ring AllReduce operation, with each worker receiving partial gradients from its upstream worker, performing gradient aggregation, and sending the results to the downstream worker in every step. The synchronization includes two phases: *ReduceScatter* for aggregating gradients, and *AllGather* for synchronizing the aggregated gradients among workers. For a training job of $|J_w|$ workers, both ReduceScatter and AllGather take $|J_w|-1$ steps, respectively, yielding $2|J_w|-2$ steps in total. We use J_a^j, $j \in J_w$, to denote the set of Ring AllReduce tasks associated with worker j (*i.e.*, tasks performing gradients aggregation on worker j), and J_a to denote the set of all tasks in one Ring AllReduce operation. Fig. 2 illustrates an example Ring AllReduce operation among three workers, which takes 4 steps.

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

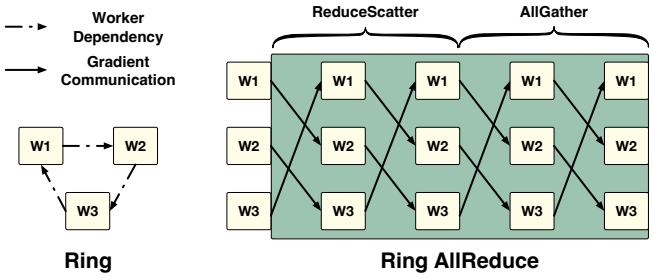


Fig. 2: Ring AllReduce

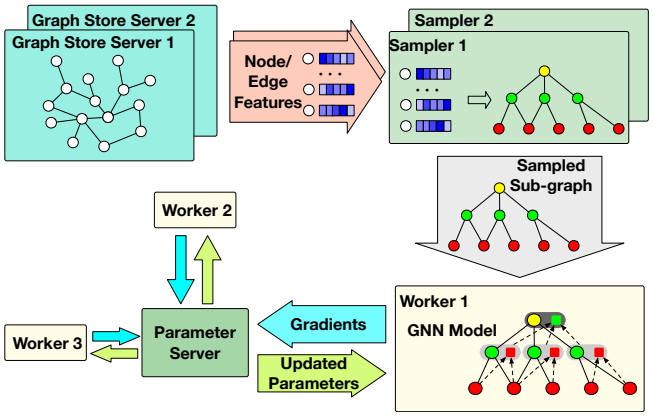


Fig. 3: Distributed GNN training workflow

Let J denote the set of all tasks, *i.e.*, $J = J_g \cup J_s \cup J_w \cup J_{ps} \cup J_a$. 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). 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 [14], and consume the respective memory. Tasks of the same kind (*e.g.*, all samplers) occupy the same amount of resources. In addition, Ring AllReduce tasks in J_a^j are colocated with their associated worker j, and share the same set of computational resources with j (*i.e.*, GPU). Hence, we set the resource demands for all tasks in J_a to be 0. Let p_j^m denote the execution time of task j in each iteration on machine m.

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. 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. For PS-based training, computed gradients are sent from workers to the PSs and then updated parameters are dispatched from PSs

to workers. For Ring AllReduce-based training, workers share and update the gradients via the Ring AllReduce operation, and apply the updated gradients locally. The workflow with PS-based training is illustrated in Fig. 3.

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. Each server hosts exactly one graph store server. The Ring AllReduce tasks are colocated with their associated workers, *e.g.*, tasks in J_a^j are colocated with worker j. During distributed GNN training, the placement of each task is determined at the beginning of the training and remains unchanged throughout the entire training process. 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:

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

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

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

$$y_j^m = y_{j'}^m, \forall j \in J_a^{j'}, j' \in J_w \tag{4}$$

$$y_j^m \in \{0,1\}, \forall j \in J, m \in [M] \tag{5}$$

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. (4) guarantees the colocation of the Ring AllReduce tasks and their associated workers. Fig. 4(a) shows an example task placement of a PS-based 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. We use $m(j)$ to denote the machine where task j resides, *i.e.*, $m(j)$ equals 1 if y_j^m is 1 and 0 otherwise. 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, parameters updated at a PS (j) in iteration-n training to a worker (j') for iteration-$(n+1)$ training ($n' = n+1$), or gradient communication from one worker's task j to its downstream worker's task j' during the Ring AllReduce operation.

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

$$x_{j,1}^1 = 1, \forall j \in J_g \tag{6}$$

$$\sum_{t\in[T]} x_{j,n}^t = 1, \forall j \in J, n \in [N] \tag{7}$$

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

$$\max\{t|k_{(j,n)\to(j',n')}^t > 0, t \in [T]\} < \sum_{t\in[T]} t x_{j',n'}^t, \forall j \in J,$$
$$n \in [N], (j',n') \in succ(j,n), j \text{ and } j' \text{ are on different servers} \tag{9}$$

$$\sum_{t\in[T]} t x_{j,n}^t + p_j^{m(j)} \leq \sum_{t\in[T]} t x_{j',n'}^t, \forall j \in J, n \in [N],$$
$$(j',n') \in succ(j,n), j \text{ and } j' \text{ are on the same server} \tag{10}$$

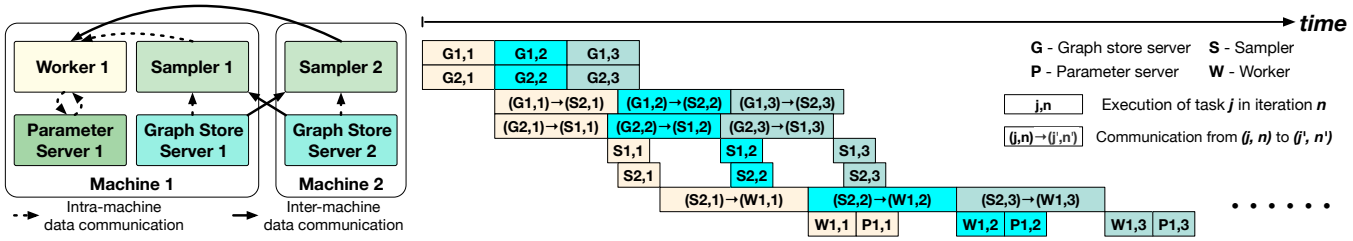
We ignore the training node selection time at a sampler, and message passing from a sampler to a graph store server for graph data requests, as the traffic volume is negligible. Constraint (6) indicates that graph store servers run first to sample neighbors. (7) ensures that each task in each training 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 $n-1$ via PS or Ring AllReduce operation; (iii) for PS-based training, a PS can start after receiving gradients from all workers, computed in this iteration; (iv) for Ring AllReduce-based training, the Ring AllReduce operation can start after all workers compute their gradients locally; (v) the Ring AllReduce tasks in one iteration follow the communication dependencies as shown in Fig. 2, and tasks associated with one worker can only be executed sequentially in one 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 (8) specifies that transmission from (j,n) to its successor (j',n') starts after (j,n) is done. (9) 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 (10).

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 (11) and (12):

$$\sum_{t\in[T]} t x_{j,n}^t + p_j^{m(j)} \leq \sum_{t\in[T]} t x_{j,n+1}^t, \forall j \in J, n \in [N-1] \tag{11}$$

$$\max\{t|k_{(j,n)\to(j',n')}^t > 0, t \in [T]\} <$$
$$\min\{t|k_{(j,n+1)\to(j',n'+1)}^t > 0, t \in [T]\}, \forall j \in J, n \in [N-1],$$
$$(j',n') \in succ(j,n), j \text{ and } j' \text{ are placed on different servers} \tag{12}$$



(a) Task placement

(b) Execution and flow schedule

Fig. 4: A PS-based distributed GNN training planning example: a job with 2 graph store servers, 1 worker with 2 samplers, and 1 PS.

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) or during a Ring AllReduce operation.

$$\sum_{t\in[T]} k^t_{(j,n)\to(j',n')} = d_{(j,n)\to(j',n')}, \forall j \in J, n \in [N],$$
$$(j',n') \in succ(j,n), j \text{ and } j' \text{ are placed on different servers} \quad (13)$$

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

$$\sum_{n\in[N]} \sum_{j\in J: y^m_j=1} \sum_{(j',n')\in succ(j,n): y^m_{j'}=0} k^t_{(j,n)\to(j',n')} \le B^m_{out},$$
$$\forall m \in [M], t \in [T] \quad (14)$$

$$\sum_{n\in[N]} \sum_{j\in J: y^m_j=0} \sum_{(j',n')\in succ(j,n): y^m_{j'}=1} k^t_{(j,n)\to(j',n')} \le B^m_{in},$$
$$\forall m \in [M], t \in [T] \quad (15)$$

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

$$\min \max_{t\in T, j\in J} \{tx^t_{j,N} + p^{m(j)}_j\} \quad (16)$$

subject to:

$$(6)\text{–}(15)$$
$$x^t_{j,n} \in \{0,1\}, \forall j \in J, n \in [N], t \in [T] \quad (17)$$
$$k^t_{(j,n)\to(j',n')} \ge 0, \forall j \in J, n \in [N], t \in [T],$$
$$(j',n') \in succ(j,n), j \text{ and } j' \text{ are placed on different servers} \quad (18)$$

Problem (16) is a generalization of the strongly NP-hard multi-stage coflow scheduling problem (MSCSP) [21], 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 [10], 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.

TABLE I: Notation

T	total time span
J	set of all tasks
$J_g/J_s/J_w/J_{ps}/J_a$	set of graph store servers/samplers/workers/ PSs/Ring AllReduce tasks
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^m_j	execution time of task j in one iteration on machine m
$d_{(j,n)\to(j',n')}$	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)
$m(j)$	the machine that task j is placed
$x^t_{j,n}$	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

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. 4(b), where we depict task execution and flow communication for the first three training iterations, based on the task placement in Fig. 4(a). Each training iteration is denoted using a different color.

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

IV. Online Execution and Flow Scheduling

We begin by introducing our online algorithm, **OES**, for task execution and flow transmission. To minimize the training makespan, we maximize GPU utilization by promptly initiating task execution once all necessary data are received. Regarding flow transmission, we implement a strategy of balanced bandwidth allocation among all active flows on each server. This approach prevents any individual flow from becoming a bottleneck that could impede the overall training process. Consequently, we can effectively reduce the training makespan in a competitive manner.

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).

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

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

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** $(j,n) \to (j',n') \in \mathcal{F}(j,n)$ **do**
10 **if** $(j,n-1) \to (j',n'-1) \in F_{act} \cup F_{pend}$ **then**
11 add $(j,n) \to (j',n')$ to F_{pend}
12 **else**
13 add $(j,n) \to (j',n')$ to F_{act}
14 **for** *every flow* $(j,n) \to (j',n')$ *finished at* $t-1$ **do**
15 **if** $(j,n+1) \to (j',n'+1) \in F_{pend}$ **then**
16 remove $(j,n+1) \to (j',n'+1)$ from F_{pend}
17 add $(j,n+1) \to (j',n'+1)$ to F_{act}
18 **for** $m \in [M]$ **do**
19 calculate Δ_{in}^m and Δ_{out}^m according to (19) (20)
20 **for** $(j,n) \to (j',n') \in F_{act}$ **do**
21 $k_{(j,n)\to(j',n')}^t \leftarrow \min\{B_{in}^{m'}/\Delta_{in}^{m'}, B_{out}^m/\Delta_{out}^m\}$, where $y_j^m = 1$ and $y_{j'}^{m'} = 1$
22 **return** $\{x_{j,n}^t\}, \{k_{(j,n)\to(j',n')}^t\}, T_{OES}$

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 $k_{(j,n)\to(j',n')}^t$ at t to $\min\{B_{in}^{m'}/\Delta_{in}^{m'}, B_{out}^m/\Delta_{out}^m\}$ (lines 18-21). Δ_{in}^m (Δ_{out}^m) is the *ingress flow degree* (*egress flow degree*) on machine m, counting the number of active flows entering and exiting from m, respectively:

$$\Delta_{in}^m = |\{(j',n') \to (j,n) | (j',n') \to (j,n) \in F_{act}, y_j^m = 1\}| \tag{19}$$

$$\Delta_{out}^m = |\{(j,n) \to (j',n') | (j,n) \to (j',n') \in F_{act}, y_j^m = 1\}| \tag{20}$$

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

The procedure of distributed GNN training generates fluctuating and intensive data flows, and it also introduces complex task dependencies that our algorithm, **OES**, is designed to address. Our approach dynamically responds to uncertainties during training: a flow is initiated once the corresponding tasks are finalized, and tasks are launched when all necessary data flows are available. This strategy inherently ensures the fulfillment of task flow dependencies. By distributing server bandwidth equally among active flows, our design prevents the training process from stalling due to a single bottleneck flow. Moreover, this approach can effectively manage dynamic flows, as it is supported by the Linux traffic control system [49], thereby ensuring minimal scheduling overhead.

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* $\widehat{\Delta_{out}^m}$:

$$\widehat{\Delta_{in}^m} = |\{(j',n') \to (j,n) | (j',n') \to (j,n) \in F_{one\_iter}, y_j^m = 1\}| \tag{21}$$

$$\widehat{\Delta_{out}^m} = |\{(j,n) \to (j',n') | (j,n) \to (j',n') \in F_{one\_iter}, y_j^m = 1\}| \tag{22}$$

and the *maximum degree* Δ:

$$\Delta = \max_{m \in [M]} \{\max\{\widehat{\Delta_{in}^m}, \widehat{\Delta_{out}^m}\}\} \tag{23}$$

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

Lemma 1. *In any time step* t, Δ_{in}^m (Δ_{out}^m) *are no larger than* $\widehat{\Delta_{in}^m}$ ($\widehat{\Delta_{out}^m}$), *for any* $m \in [M]$.

The detailed proof is given in Appendix A.

Let T_{OES} denote the overall training makespan achieved by Alg. 1. During the execution scheduled by Alg. 1, we now define a *continuous execution chain* as a chain $O: o_L \to o_{L-1} \to \cdots \to o_1$, starting from the execution of one of the graph store servers in iteration 1 to the last task in iteration N, strictly following either the execution dependency or the inter-iteration dependency and being executed continuously with no gap. Each component in such a chain O represents either a task or a data flow between tasks. Hence, the execution and

data communication time of every continuous execution chain covers the overall training makespan T_{OES}.

Lemma 2. *There exists at least one continuous execution chain, O, during the distributed GNN training scheduled by Alg. 1.*

The detail proof is given in Appendix B.

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 (16), i.e., the competitive ratio of the online algorithm in Alg. 1 is Δ.*

Proof. Given Lemma 2, we consider one such continuous execution chain O. Denote the total execution time of the task in O as p_{sum}. Supposing that there are I data transmissions in O, we use d_1 to d_I to denote the amount of data for each transmission. In addition, we use m_i^{in} (m_i^{out}) to denote the server where the i-th flow comes to (from). Clearly, in the optimal offline scheduling strategy with makespan T^*, the chain O also needs to be executed sequentially. Hence, we have that:

$$T^* \geq p_{sum} + \sum_{i\in[I]} \frac{d_i}{\min\{B_{in}^{m_i^{in}}, B_{out}^{m_i^{out}}\}} \tag{24}$$

In addition, we also have that T_{OES} equals the time for executing whole chain O. And in the execution of chain O, following Lemma 1, the i-th flow are transferred with a data rate at least $\min\{B_{in}^{m_i^{in}}/\widehat{\Delta_{in}^{m_i^{in}}}, B_{out}^{m_i^{out}}/\widehat{\Delta_{out}^{m_i^{out}}}\}$. Consequently, we have:

$$T_{OES} \leq p_{sum} + \sum_{i\in[I]} \frac{d_i}{\min\{B_{in}^{m_i^{in}}/\widehat{\Delta_{in}^{m_i^{in}}}, B_{out}^{m_i^{out}}/\widehat{\Delta_{out}^{m_i^{out}}}\}} \tag{25}$$

Combining (24) and (25), we have:

$$\begin{aligned} T_{OES} &\leq p_{sum} + \sum_{i\in[I]} \frac{d_i}{\min\{B_{in}^{m_i^{in}}/\widehat{\Delta_{in}^{m_i^{in}}}, B_{out}^{m_i^{out}}/\widehat{\Delta_{out}^{m_i^{out}}}\}} \\ &\leq \Delta(p_{sum} + \sum_{i\in[I]} \frac{d_i}{\min\{B_{in}^{m_i^{in}}, B_{out}^{m_i^{out}}\}}) \\ &\leq \Delta \times T^* \end{aligned}$$

□

V. Exploratory Task Placement

In distributed GNN training, the communication pattern adheres to an iterative paradigm, wherein data flows exist between task pairs in each iteration, albeit with fluctuating sizes. Consequently, strategic placement of tasks via co-location can effectively eliminate their communication throughout the entirety of the training process. We adopt the Markov Chain Monte Carlo (MCMC) search framework [22] to identify an optimal placement solution that minimizes the training makespan with our online scheduling Alg. 1. We begin 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. The initial placement is constructed using a dynamic programming approach. Leveraging task dependencies during training, our design constructs an execution directed acyclic graph for each possible task placement. We are thus inspired to propose a critical-path-based cost function to evaluate the quality of a particular task placement based on its execution graph. Herein, a placement with a lower cost is more likely to achieve a reduced training makespan when utilizing our online scheduling algorithm, **OES**. By viewing each placement as a sample within the solution space, we employ the MCMC search framework to iteratively explore this space. This approach ensures that a placement with a lower cost is more likely to be encountered during the search process.

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), and the Ring AllReduce tasks are collocated with their associated workers. 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 $q_s' \in \{0\} \cup [|J_s| - q_s]$, $q_w' \in \{0\} \cup [|J_w| - q_w]$, 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 place-

Algorithm 2: Initial Placement Solution Construction - **IFS**

Input: $J, M, R, \{C^r\}, \{w_j^r\}$
Output: $\mathcal{Y}_0$

Randomly order m machine as $\{m_1, m_2, \ldots, m_M\}$
$\Omega(m) \leftarrow \emptyset, \forall m \in [M]$
for $q_s, q_w, q_{ps} \in [\eta_s], [\eta_w], [\eta_{ps}]$ **do**
  if *q_s samplers, q_w workers, and q_{ps} parameter servers can be packed in m_1* **then**
    add $(q_s, q_w, q_{ps}, 1)$ to set $\Omega(1)$
    $\mathcal{A}(q_s, q_w, q_{ps}, 1) \leftarrow \{< q_s, q_w, q_{ps}, 1 >\}$
for $i \in \{2, 3, \ldots, M\}$ **do**
  for $(q_s, q_w, q_{ps}, i-1) \in \Omega(i-1)$ **do**
    if *$|J_s| - q_s$ samplers, $|J_w| - q_w$ workers, and $|J_{ps}| - q_{ps}$ parameter servers can be packed in m_i* **then**
      $\mathcal{A}_{solution} \leftarrow \mathcal{A}(q_s, q_w, q_{ps}, i-1) \cup \{< |J_s| - q_s, |J_w| - q_w, |J_{ps}| - q_{ps}, i >\}$
      Generate the initial placement solution $\mathcal{Y}_0 = \{y_j^m\}_0$ based on $\mathcal{A}_{solution}$
      return $\mathcal{Y}_0$
    for $q_s', q_w', q_{ps}' \in [\eta_s], [\eta_w], [\eta_{ps}]$ **do**
      if *q_s' samplers, q_w' workers, and q_{ps}' parameter servers can be packed in m_i* **then**
        if $q_s + q_s' \le |J_s|$ *and* $q_w + q_w' \le |J_w|$ *and* $q_{ps} + q_{ps}' \le |J_{ps}|$ **then**
          add $(q_s + q_s', q_w + q_w', q_{ps} + q_{ps}', i)$ to set $\Omega(i)$
          $\mathcal{A}(q_s + q_s', q_w + q_w', q_{ps} + q_{ps}', i) \leftarrow \mathcal{A}(q_s, q_w, q_{ps}, i-1) \cup \{< q_s', q_w', q_{ps}', i >\}$

ment solution. We summarize our initial placement solution construction algorithm in Alg. 2, referred to as **IFS**.

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

Proof. Alg. 2 searches all the tuples (q_s, q_w, q_{ps}, m) through the construction of $\Omega(1)$ to $\Omega(M)$. If a feasible solution exists, it must correspond to one of the tuples, indicating that Alg. 2 can find the feasible solution.

The construction of $\Omega(1)$ requires $O(\eta_s\eta_w\eta_{ps}R)$ time. Giving $\Omega(m-1)$, for every tuple in it, we can compute all related tuples in $\Omega(m)$ in $O(\eta_s\eta_w\eta_{ps}R)$ time. Since there are at most $O(|J_s||J_w||J_{ps}|)$ tuples in $\Omega(m-1)$, the construction of $\Omega(m)$ giving $\Omega(m-1)$ takes $O(|J_s||J_w||J_{ps}|\eta_s\eta_w\eta_{ps}R)$. Consequently, the time complexity of Alg. 2 is $O(M|J_s||J_w||J_{ps}|\eta_s\eta_w\eta_{ps}R)$. □

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 $\mathcal{Y}$.

Algorithm 3: Exploratory Task Placement - **ETP**

Input: $T, J, M, R, \{C_m^r\}, \{w_j^r\}$
Output: $\mathcal{Y}_{min}$

$\mathcal{Y}_0 \leftarrow$**IFS**$(J, M, R, \{C_m^r\}, \{w_j^r\})$; $\mathcal{Y}_{min} \leftarrow \mathcal{Y}_0$
$\_, \_,$ `min_makespan` $=$ **OES**$(T, J, M, N, \mathcal{Y}_0)$
Construct the execution DAG based on $\mathcal{Y}_0$
Derive the critical path length in the DAG, $\overline{\mathcal{CP}}_{\mathcal{Y}_0}$
`min_cost` $\leftarrow \overline{\mathcal{CP}}_{\mathcal{Y}_0}$
for $z \in \{0, 1, \ldots, I-1\}$ **do**
  randomly select a task j from $J \setminus (J_g \cup J_a)$
  construct M_{avail} of j
  randomly select a machine m from M_{avail}
  construct new placement solution $\mathcal{Y}'$
  Construct the execution DAG based on $\mathcal{Y}'$
  Derive the critical path length in the DAG, $\overline{\mathcal{CP}}_{\mathcal{Y}'}$
  $\pi(\mathcal{Y}_z \to \mathcal{Y}') \leftarrow \min\{1, \exp(\beta cost(\mathcal{Y}_z) - \beta cost(\mathcal{Y}'))\}$
  if *rand()*$\le \pi(\mathcal{Y}_z \to \mathcal{Y}')$ **then**
    $\mathcal{Y}_{z+1} \leftarrow \mathcal{Y}'$
    if *no resource violation with $\mathcal{Y}_{z+1}$ and* $\overline{\mathcal{CP}}_{\mathcal{Y}'} < (1+\gamma)$`min_cost` **then**
      if $\overline{\mathcal{CP}}_{\mathcal{Y}'} <$`min_cost` **then**
        `min_cost`$\leftarrow \overline{\mathcal{CP}}_{\mathcal{Y}'}$
      $\_, \_, T'_{\mathcal{Y}'} =$ **OES**$(T, J, M, N, \mathcal{Y}')$
      if $T'_{\mathcal{Y}'} <$ `min_makespan` **then**
        `min_makespan` $\leftarrow T'_{\mathcal{Y}'}$
        $\mathcal{Y}_{min} \leftarrow \mathcal{Y}'$
  else
    $\mathcal{Y}_{z+1} \leftarrow \mathcal{Y}_z$
return $\mathcal{Y}_{min}$

Cost design. Practically, task placements should be decided before training starts and remain fixed during training (to avoid the substantial overhead of VM/container migration and flow redirection). The online nature of execution scheduling is due to the 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.

A typical GNN training job takes thousands of iterations [8], making directly simulating the overall training time scheduled by Alg. 1 over N iterations prohibitively time-consuming. Hence, we propose a cost function that is both efficient to calculate and can accurately represent the performance with the placement $\mathcal{Y}$.

Given the current placement $\mathcal{Y}$, we construct an execution directed acyclic graph (DAG) $G = (V, E)$, where the vertex set V includes all tuples of every task in each iteration (*i.e.* $(j, n), \forall j \in J, n \in [N]$), and all the inter-machine data transmissions between tasks (*i.e.* $(j, n) \to (j', n'), \forall j \in J, n \in [N], (j', n') \in succ(j, n)$, j and j' are placed on different servers). The edge set E consists of all the dependencies between vertexes in V, including the execution dependencies defined through (8) to (10) and the inter-iteration dependencies

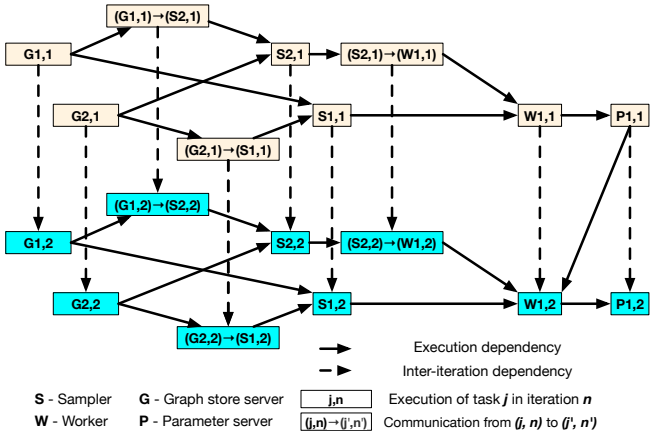


Fig. 5: An execution DAG example

defined in (11) and (12). We give an example of an execution DAG for two iterations of training in Fig. 5, based on the training system and placement specified by Fig. 4.

We assign a weight $\alpha(v)$ to every vertex v in V, denoting the execution/data communication time of the vertex. For vertex (j,n) denoting the component execution, we set the expected time $\alpha(v)$ to be $p_j^{m(j)}$. For vertex $(j,n)\rightarrow(j',n')$ representing the data transmission, we set the corresponding $\alpha(v)$ as follows:

$$\alpha(v)=\frac{d_{(j,n)\rightarrow(j',n')}}{\min\{B_{in}^{m(j')}/\widehat{\Delta_{in}^{m(j')}},B_{out}^{m(j)}/\widehat{\Delta_{out}^{m(j)}}\}} \tag{26}$$

where $\widehat{\Delta_{in}^{m(j')}}$ and $\widehat{\Delta_{out}^{m(j)}}$ are the one-iteration ingress/egree flow degree defined in Eqn. (21) and Eqn. (22), respectively. The following theorem establishes the connection between the critical path in G (*i.e.* the path with the largest aggregated weights) and the training time of all N iterations by Alg. 1, T_{OES}.

Theorem 3. *Given the placement $\mathcal{Y}$ and the constructed DAG G, the makespan achieved by Alg. 1, T_{OES}, is no larger than the critical path length, $\mathcal{CP}_{\mathcal{Y}}$, (i.e. length of the longest weighted path) in G.*

Proof. Note that in the proof of Theorem 1, we find a continuous execution chain O whose execution covers the overall training time T_{OES}. Now, we can map every part in O to its corresponding vertex in G to form a path, whose weighted length is denoted as $\mathcal{CP}_O$. Consequently, combining Eqn. (25) and each vertex's weight, we ensure that $T_{OES}\leq\mathcal{CP}_O$. Since $\mathcal{CP}_{\mathcal{Y}}$ is the length of the longest weighted path, we have $T_{OES}\leq\mathcal{CP}_O\leq\mathcal{CP}_{\mathcal{Y}}$. □

Theorem 3 shows that the critical path length in G, $\mathcal{CP}_{\mathcal{Y}}$, is an upper bound of the overall training time T_{OES}, indicating that we can leverage $\mathcal{CP}_{\mathcal{Y}}$ to judge the quality of the current placement. The critical path length in G can be calculated efficiently [50], with time complexity $O(|V|+|E|)$. However, the data communication traffic volume $d_{(j,n)\rightarrow(j',n')}$ cannot be known in advance. Therefore, we can replace $d_{(j,n)\rightarrow(j',n')}$ with $\bar{d}_{(j,n)\rightarrow(j',n')}$ drawn from the distribution generated by the profiled data, and derive the critical path length $\overline{\mathcal{CP}}_{\mathcal{Y}}$. The cost of placement $\mathcal{Y}$ is:

$$cost(\mathcal{Y})=\overline{\mathcal{CP}}_{\mathcal{Y}}(1+\sum_{m\in[M],r\in[R]}\max\{\frac{\sum_{j\in J}w_j^r y_j^m-C_m^r}{C_m^r},0\}) \tag{27}$$

where $\overline{\mathcal{CP}}_{\mathcal{Y}}$ 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).

Search process. 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 $j\in J\setminus(J_g\cup J_a)$. 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:

$$\sum_{j\in J}w_j^r y_j^m\leq(1+\mu)C_m^r,\forall m\in[M],r\in[R] \tag{28}$$

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):

$$\pi(\mathcal{Y}_z\rightarrow\mathcal{Y}')=\min\{1,\exp(\beta cost(\mathcal{Y}_z)-\beta cost(\mathcal{Y}'))\} \tag{29}$$

With probability $\pi(\mathcal{Y}_z\rightarrow\mathcal{Y}')$, we use $\mathcal{Y}'$ as $\mathcal{Y}_{z+1}$: if $cost(\mathcal{Y}')\leq cost(\mathcal{Y}_z)$, we accept $\mathcal{Y}'$ as $\mathcal{Y}_{z+1}$ (probability is 1); otherwise, we still accept $\mathcal{Y}'$ as the next state with probability $\pi(\mathcal{Y}_z\rightarrow\mathcal{Y}')$ (for exploration) and maintain $\mathcal{Y}_{z+1}$ the same as $\mathcal{Y}_z$ with probability $1-\pi(\mathcal{Y}_z\rightarrow\mathcal{Y}')$. Our state transition as designed above ensures that the probability of visiting $\mathcal{Y}$ is linear to $\exp(-\beta cost(\mathcal{Y}))$ [22], *i.e.*, solutions with lower costs are more frequently visited than ones with larger costs.

Although Theorem 3 shows that $\overline{\mathcal{CP}}_{\mathcal{Y}}$ is an indicator of the training makespan, placement with the minimum $\overline{\mathcal{CP}}_{\mathcal{Y}}$ may not be the placement achieving the minimum training makespan. We address this gap by occasionally simulating the training makespan of placements of small $\overline{\mathcal{CP}}_{\mathcal{Y}}$. Throughout this exploratory process, we use `min_cost` to track the minimum $\overline{\mathcal{CP}}_{\mathcal{Y}}$ without any resource violation. Whenever we transit to a new placement $\mathcal{Y}'$, if $\mathcal{Y}'$ does not violate any original resource capacity constraints, and $cost(\mathcal{Y}')\leq(1+\gamma)\times$`min_cost`, we simulate training under placement $\mathcal{Y}'$ using Alg. 1, with time and traffic volume drawn from the profiled distributions, and

Algorithm 4: Distributed GNN Training Planning (*DGTP*)

Input: $T, N, J, M, R, \{C^r_m\}, \{w^r_j\}$
Output: $\mathcal{Y}_{min}, \{x^t_{j,n}\}, \{k^t_{(j,n)\to(j',n')}\}$
$\mathcal{Y}_{min} \leftarrow$ **ETP**$(T, J, M, R, \{C^r_m\}, \{w^r_j\})$
$\{x^t_{j,n}\}, \{k^t_{(j,n)\to(j',n')}\}, T_{OES} \leftarrow$ **OES**$(T, J, M, N, \mathcal{Y}_{min})$
return $\mathcal{Y}_{min}, \{x^t_{j,n}\}, \{k^t_{(j,n)\to(j',n')}\}$

derive the training makespan $T'_{\mathcal{Y}'}$. Here γ is a hyperparameter, and is set to 0.1 in our evaluation.

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 whose training time was simulated. Alg. 3 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. 4. We first leverage Alg. 3 to identify the best placement $\mathcal{Y}_{min}$ and then use Alg. 1 to decide the task and flow schedules $\{x^t_{j,n}\}, \{k^t_{(j,n)\to(j',n')}\}$ 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 0.8.2 [23] and PyTorch 1.12.0 [26] with 1043 LoC for the training system and 1544 LoC for the search and scheduling algorithms. Parameter synchronization through a PS or the Ring AllReduce operation is built on PyTorch. We use the Stochastic Fairness Queueing provided by tc qdisc [49] 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 [8] (an Amazon product co-purchasing 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 fan-outs (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][14]. Same as in *DistDGL* [14], we set the mini-batch size on both datasets to 2000 (subgraphs). We use Adam optimizer [51] 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.

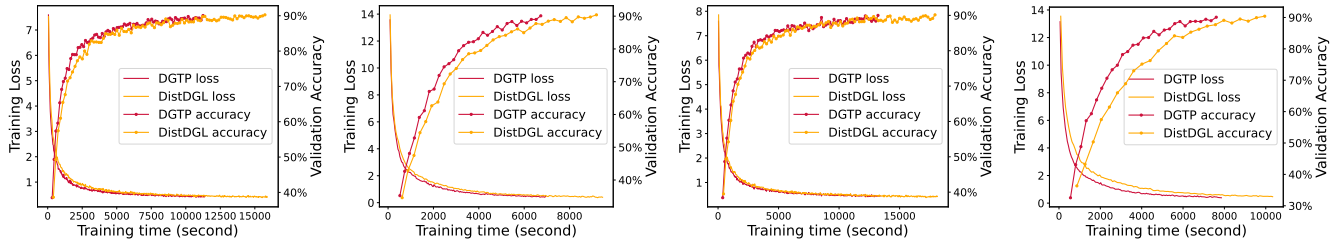
Baseline. We compare *DGTP* with a modified version of *DistDGL* [14] that enables inter-server communication between a worker and its samplers. *DistDGL* adopts a placement scheme that maximally colocates each worker with its associated samplers within one server, and uses the system default scheduling 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 3 minutes (we only simulate 20 iterations of GNN training to obtain $T'_{\mathcal{Y}}$ during search, and set the exploratory time budget to 10000). Fig. 6 shows the training progress to achieve a 90% target accuracy over the validation sets. We demonstrate the training speed-up of our design relative to *DistDGL* across four different configurations, as shown in Table III. The speed-up is calculated using the formula $\frac{\text{Time-to-Convergence of DistDGL}-\text{Time-to-Convergence of DGTP}}{\text{Time-to-Convergence of DistDGL}}$. In PS-based training, *DGTP* outperforms *DistDGL* by 28% in terms of the training speed-up on ogbn-products, and 26% on Reddit. Moreover, in Ring AllReduce-based training, *DGTP* outperforms *DistDGL* by 26% on ogbn-products, and 24% on Reddit.

	ogbn-products	**Reddit**
PS-based training	28.45%	26.54%
Ring AllReduce-based training	26.11%	24.00%

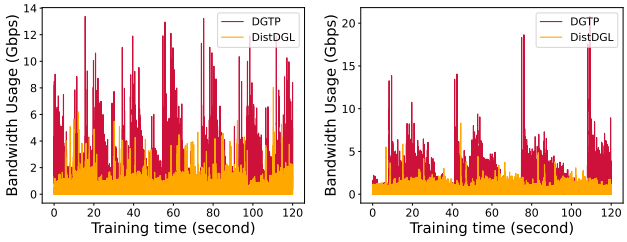
TABLE III: Training Speed-up

Resource usage. We also examined resource usage during PS-based 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. 7 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).



(a) PS-based training on ogbn-products dataset (b) PS-based training on Reddit dataset (c) Ring AllReduce-based training on ogbn-products dataset (d) Ring AllReduce-based training on Reddit dataset

Fig. 6: Training loss & validation accuracy: *DGTP* vs. *DistDGL*



(a) ogbn-products (b) Reddit

Fig. 7: Total network bandwidth usage: *DGTP* vs. *DistDGL*

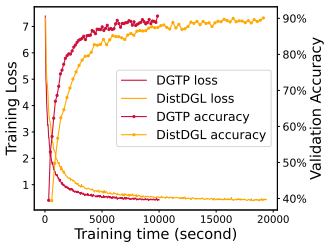


Fig. 8: Training in the heterogeneous testbed: *DGTP* vs. *DistDGL*

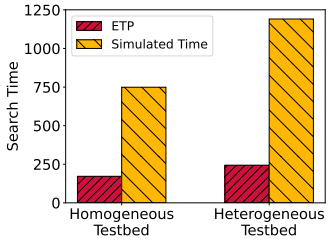
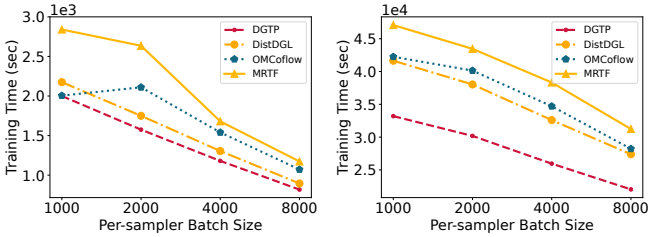
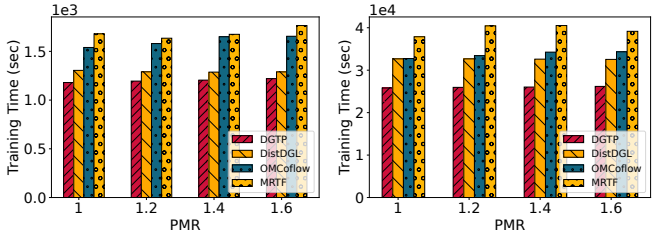


Fig. 9: Search Time: different cost criteria



(a) ogbn-products (b) ogbn-papers100M

Fig. 10: Training time: different batch sizes



(a) ogbn-products (b) ogbn-papers100M

Fig. 11: Training time: different PMRs

Heterogeneous testbed. We further evaluated *DGTP* in a heterogeneous testbed, consisting of two GTX 1080Ti GPU servers described before and two Tesla V100 GPU servers. Each V100 server is equipped with 4 Tesla V100 GPUs, two 10-core Intel Xeon E5-2630 v4 CPUs, 256GB DDR4 RAM and a 100GbE NIC. The four servers are interconnected by a Dell Z9100-ON switch, with NIC bandwidth of the two 1080Ti servers limited to 10Gbps. We train *DGTP* and *DistDGL* on ogbn-products with Ring AllReduce, and increase the number of workers to 8. Fig. 8 depicts the training progress of *DGTP* and *DistDGL*, showing superiority of *DGTP* in the heterogeneous environment, which achieves a training speed-up of 48% as compared to *DistDGL*.

Search time. We now evaluate the effectiveness of our exploratory task placement design. We compare the search time of our design, **ETP**, with the algorithm that directly uses the simulated training time, $T'_{\mathcal{Y}'}$, as the cost for PS-based training on ogbn-products in both the homogeneous and heterogeneous training testbeds. Both algorithms search for 10000 steps and are able to identify the placement with similar training efficiency. As shown in Fig. 9, our design achieves significantly less searching time as compared to the simulated-training-time-based method.

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 [52]). 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 $\{10Gbps, 20Gbps, 50Gbps\}$.

Baselines. We reuse *DistDGL* as our baseline, following a

placement scheme that maximally colocates each worker with its associated samplers within one server, and leveraging our online task execution and flow scheduling design **OES** for scheduling. 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 [53] 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. 10 shows, *DGTP* outperforms all three baselines, reducing the training makespan 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* can achieve more than 30% 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. Furthermore, we note that the average training speed-up in comparison to *OMCoflow*, which utilizes the same task placements as *DGTP*, remains relatively consistent in the two settings, *i.e.*, 18% with the ogbn-products training setting, and 23% with the ogbn-papers100M training setting. Conversely, the average training speed-up of our design in comparison to *DistDGL* experiences a significant increase, jumping from 9% with the ogbn-products training setting to 29% with the ogbn-papers100M training setting. This substantial performance improvement suggests that as the workload increases, strategic task placement can yield significantly larger performance gains compared to online flow scheduling.

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. 11, *DGTP* exhibits stable performance as the PMR changes, and outperforms representative baselines by up to 55%.

VII. Conclusion

This paper designs efficient placement and scheduling algorithms for distributed GNN training over heterogeneous clusters, with various resource availability. We propose a competitive online execution algorithm that schedules training task execution and flow transfers for both graph data sampling and parameter synchronization with either PS or Ring AllReduce architecture. We also design an explorative algorithm to decide the placement of every task in a short period of time, which, in conjunction with online task/flow scheduling, minimizes the overall training makespan. According to both homogeneous and heterogeneous testbed experiments, our design reduces the end-to-end training time by up to 48% 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. While our design has made significant strides towards efficient distributed GNN training, several promising avenues remain for future exploration. One limitation of our current model is the assumption of a homogeneous network with a classic starfish topology. Investigating the impact of bandwidth heterogeneity on performance in a topology more reflective of real-world conditions represents a meaningful direction for future research. Moreover, extending our approach to scenarios where multiple GNN training jobs operate concurrently on the same cluster is a promising prospect. Future efforts could involve designing a joint task placement search algorithm for all jobs, and orchestrating online task and flow scheduling for the jobs.

Appendix A
Proof of Lemma 1

Proof. Without loss of generality, let us consider a time step t. Due to the inter-iteration dependency (12), there will be at most one data flow from task j to task j'. As a result, for any active flow $(j,n) \to (j',n')$ in F_{act}, we can substitute the flow with its counterpart one (*i.e.* the same flow from task j to task j') in the first iteration. Let us substitute all flows in F_{act} with their counterparts in the first iteration and denote the new set as F'_{act}. Clearly, we can still obtain the same Δ^m_{in} and Δ^m_{out} with F'_{act}.

Noting that F_{one_iter} includes all inter-server flows from the first iteration, we have $F'_{act} \subseteq F_{one\_iter}$. Consequently, $\Delta^m_{in} \leq \widehat{\Delta^m_{in}}$ and $\Delta^m_{out} \leq \widehat{\Delta^m_{out}}$. □

Appendix B
Proof of Lemma 2

Proof. We construct one continuous execution chain reversely by setting o_1 as the last task (*i.e.*, the parameter server or the last task in the Ring AllReduce), denoted as (j_{last}, N), in iteration N that finishes at T_{OES}. We use $\tilde{t}(o_l)$ to denote the start time of $o_l, \forall l \in [L]$. Now, considering $\tilde{t}(o_1)$, there are two possibilities:

1): All flows to o_1 finish before $\tilde{t}(o_1)$.

2): At least one flow to o_1 finishes at $\tilde{t}(o_1)$.

The first possibility indicates that the execution of the same task for previous batch ($(j_{last}, N-1)$ in this case) finishes at $\tilde{t}(o_1)$. Therefore, we can add the execution of $(j_{last}, N-1)$ to chain O denoted as o_2. Now we extend O further by one task to cover more part of the total makespan T_{OES}.

For **the second possibility**, without loss of generality, we consider one flow, denoted as f_1, to o_1 finishing at $\tilde{t}(o_1)$. We can add the flow f_1 to chain O as o_2. Now, considering the start time of f_1, $\tilde{t}(f_1)$, there are two cases:

▷ **Case 1**: The task that f_1 originates from finished at $\tilde{t}(f_1)$.

In this case, we can add the task to O as o_3. We use o_3 and o_2 to further extend O, covering more makespan.

▷ **Case 2**: The task that f_1 originates from finished before $\tilde{t}(f_1)$.

Case 2 indicates that there exists another flow f_2 that blocks the transmission of f_1 due to the inter-iteration dependency. Now, we add both f_1 and f_2 to chain O as o_2 and o_3. Consider f_2, there are again two cases as f_1. We can follow the same procedure to add either another flow f_3 or a task to O. We repeat the procedure until we add a task to O.

In conclusion, we add one task and possibly several flows to extend our chain O that the execution of one part starts immediately after the completion of its previous one. For the newly added task, there are again two possibilities as o_1 and we can follow the same process as above to further extend the coverage of makespan by adding another task to O. Eventually, we can construct a continuous execution chain O to cover the entire makespan T_{OES}. □

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

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

[5] 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.

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

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

[8] 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.

[9] 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.

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

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

[12] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling Distributed Machine Learning with the Parameter Server," in *Proc. of USENIX OSDI*, 2014.

[13] A. Gibiansky, "Bringing hpc techniques to deep learning," *Baidu Research, Tech. Rep.*, 2017.

[14] 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.

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

[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] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters," in *Proc. of OSDI*, 2020.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] (2021) Euler Graph Library. [Online]. Available: https://github.com/alibaba/euler

[29] 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.

[30] 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.

[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] T. Liu, Y. Chen, D. Li, C. Wu, Y. Zhu, J. He, Y. Peng, H. Chen, H. Chen, and C. Guo, "BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing," *arXiv preprint arXiv:2112.08541*, 2021.

[35] J. Yang, D. Tang, X. Song, L. Wang, Q. Yin, R. Chen, W. Yu, and J. Zhou, "GNNLab: A Factored System for Sample-based GNN training over GPUs," in *Proc. of EuroSys*, 2022, pp. 417–434.

[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] Y. Zhao, K. Chen, W. Bai, M. Yu, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang, "Rapier: Integrating Routing and Scheduling for Coflow-Aware Data Center Networks," in *Proc. of IEEE INFOCO)*, 2015.

[46] M. Chowdhury, S. Khuller, M. Purohit, S. Yang, and J. You, "Near Optimal Coflow Scheduling in Networks," in *Proc. of ACM SPAA*, 2019.

[47] L. Cheng, Y. Wang, Q. Liu, D. H. Epema, C. Liu, Y. Mao, and J. Murphy, "Network-Aware Locality Scheduling for Distributed Data Operators in Data Centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 6, pp. 1494–1510, 2021.

[48] M. Shafiee and J. Ghaderi, "Scheduling coflows with dependency graph," *IEEE/ACM Transactions on Networking*, vol. 30, no. 1, pp. 450–463, 2021.

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

[50] R. Sedgewick and K. Wayne, *Algorithms*. Addison-Wesley Professional, 2014.

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

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

[53] 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.

Ziyue Luo received his Ph.D. degree from the Department of Computer Science, The University of Hong Kong in 2022. He received the B.Eng. degree from Wuhan University, China, in 2018. His research interests include cloud computing, network function virtualization, and distributed machine learning systems.

Yixin Bao received her Ph.D. degree from the Department of Computer Science, The University of Hong Kong in 2020. She received her B.Eng. degree from the Department of Automation, Xi'an Jiaotong University, in 2015. Her research interests include cloud computing, machine learning systems, and online learning algorithms.

Chuan Wu received her B.Engr. and M.Engr. degrees in 2000 and 2002 from the Department of Computer Science and Technology, Tsinghua University, China, and her Ph.D. degree in 2008 from the Department of Electrical and Computer Engineering, University of Toronto, Canada. Between 2002 and 2004, She worked in the Information Technology industry in Singapore. Since September 2008, Chuan Wu has been with the Department of Computer Science at the University of Hong Kong, where she is currently a Professor. Her current research is in the areas of distributed machine learning systems and algorithms, and intelligent elderly care technologies. She is a senior member of IEEE, a member of ACM, and served as the Chair of the Interest Group on Multimedia services and applications over Emerging Networks (MEN) of the IEEE Multimedia Communication Technical Committee (MMTC) from 2012 to 2014. She is an associate editor of ACM/IEEE Transactions on Networking and IEEE Transactions on Cloud Computing. She was the co-recipient of the best paper awards of HotPOST 2012 and ACM e-Energy 2016. She received an Amazon Research Award on AWS AI in 2021.