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 fitted 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, tested 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 172046/19, 172089/20 and 172076/21.
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 strategic 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 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 \( m \) \( \in [M] \) is equipped with \( R \) types of computational resources (\( e.g. \), GPU, CPU and memory), with type-\( r \) resource available at the amount of \( C_r^m \). Let \( B^m_{in} (B^m_{out}) \) represent the available incoming (outgoing) NIC bandwidth on machine \( m \).

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 \( W_r^j \) 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 PSSs and then updated parameters are dispatched from PSSs 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 $y^m_j$ to indicate task placement: $y^m_j = 1$ if task $j$ is deployed on machine $m$, and 0, otherwise. The placement constraints are:

$$\sum_{m \in [M]} y^m_j = 1, \forall j \in J$$  \hspace{1cm} (1)

$$\sum_{j \in J} w^r_j y^m_j \leq C^r_m, \forall m \in [M], r \in [R]$$  \hspace{1cm} (2)

$$y^m_j = 1, \forall j \in J, y^m_j \text{ is placed on machine } m$$  \hspace{1cm} (3)

$$y^m_j \in \{0, 1\}, \forall j \in J, m \in [M]$$  \hspace{1cm} (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 $x^t_{j,n}$ indicate the start time of task $j$ in iteration $n$: $x^t_{j,n} = 1$ if task $j$ in iteration $n$ starts at time $t$, and 0, otherwise. We use $k^t_{j,n} \rightarrow (j',n')$ 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:

$$\sum_{t \in [T]} x^t_{j,n} = 1, \forall j \in J \text{ } n \in [N]$$  \hspace{1cm} (5)

$$\min\{t[k^t_{j,n} \rightarrow (j',n') > 0, t \in [T]]\} \geq \sum_{t \in [T]} x^t_{j,n} + p_j, \forall j \in J, n \in [N], (j', n') \in \text{succ}(j, n), j \text{ and } j' \text{ are on different servers}$$  \hspace{1cm} (6)

$$\max\{t[k^t_{j,n} \rightarrow (j',n') > 0, t \in [T]]\} < \sum_{t \in [T]} x^t_{j',n'}, \forall j \in J, n \in [N], (j', n') \in \text{succ}(j, n), j \text{ and } j' \text{ are on different servers}$$  \hspace{1cm} (7)

$$\sum_{t \in [T]} x^t_{j,n} + p_j \leq \sum_{t \in [T]} x^t_{j',n'}, \forall j \in J, n \in [N], (j', n') \in \text{succ}(j, n), j \text{ and } j' \text{ are on the same server}$$  \hspace{1cm} (8)

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 converge. Constraint (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 $\text{succ}(j, n)$ 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 $(j, n + 1) \rightarrow (j', n' + 1)$ cannot start before transmission $(j, n) \rightarrow (j', n')$ has been completed. These inter-iteration dependencies are formulated as in (10) and (11):

$$\sum_{t \in [T]} tx^t_{j,n} + p_j \leq \sum_{t \in [T]} tx^t_{j,n+1}, \forall j \in J, n \in [N - 1]$$  \hspace{1cm} (10)

$$\max\{t[k^t_{j,n} \rightarrow (j',n') > 0, t \in [T]]\} < \min\{t[k^t_{j,n+1} \rightarrow (j',n'+1) > 0, t \in [T]], \forall j \in J, n \in [N - 1], (j', n') \in \text{succ}(j, n), j \text{ and } j' \text{ are placed on different servers}$$  \hspace{1cm} (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 $d((j, n) \rightarrow (j', n'))$. 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).

$$\sum_{t \in [T]} k^t_{j,n} \rightarrow (j',n') = d((j, n) \rightarrow (j', n')), \forall j \in J, n \in [N], (j', n') \in \text{succ}(j, n), j \text{ and } j' \text{ are placed on different servers}$$  \hspace{1cm} (12)

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

$$\sum_{n \in [N], j \in J} y^m_{j,n} = 1 (j', n') \in \text{succ}(j, n); y^m_{j,n} = 0$$  \hspace{1cm} (13)

$$\sum_{n \in [N], j \in J} y^m_{j,n} = 0 (j', n') \in \text{succ}(j, n); y^m_{j,n} = 1$$  \hspace{1cm} (14)
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:

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

subject to:

$$(5)-(14)$$

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

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

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.

### 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 $k^t_{(j,n)\rightarrow(j',n')}$ over time.

We maintain two flow sets: (i) $F_{\text{act}}$, that stores every active flow $(j, n) \rightarrow (j', n')$ which currently has started but not finished transmission yet; (ii) $F_{\text{pend}}$, to store every pending flow $(j, n) \rightarrow (j', n')$ whose predecessor task $(j, n)$ has been done, and that has not started because its predecessor flow $(j, n-1) \rightarrow (j', n'-1)$ in the previous iteration has not completed transmission yet. For each task $(j, n)$, we use $F(j, n)$ 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 $(j, n) \rightarrow (j', n') \in F(j, n)$ in $t$: if the flow’s predecessor flow $(j, n-1) \rightarrow (j', n'-1)$ is in $F_{\text{act}}$ or $F_{\text{pend}}$ (indicating it not done yet), we add $(j, n) \rightarrow (j', n')$ to $F_{\text{act}}$; otherwise, it is scheduled to transmit in $t$ and added to $F_{\text{act}}$ (lines 8-13). In addition, for every flow $(j, n) \rightarrow (j', n')$ ended at $t - 1$, we check if its successor flow $(j, n+1) \rightarrow (j', n'+1)$ is in $F_{\text{pend}}$: if so, we move it from $F_{\text{pend}}$ to $F_{\text{act}}$ and start the flow transmission (lines 14-17). For every flow $(j, n) \rightarrow (j', n')$ which transfers in $t$, supposing $j$ placed on $m$ and $j'$ on $m'$, we set its traffic volume $k^t_{(j,n)\rightarrow(j',n')}$ at $t$ to $\min\{B_{\text{in}}^m/B_{\text{in}}^m, B_{\text{out}}^m/B_{\text{out}}^m\}$ (lines 18-21). $D_{\text{in}}^m (D_{\text{out}}^m)$ is the ingress flow degree (egress flow degree) on machine $m$, counting the number of active flows entering and exiting from $m$, respectively:

$$D_{\text{in}}^m = |\{(j', n') \rightarrow (j, n)| (j, n) \rightarrow (j, n) \in F_{\text{act}}, y^m_j = 1\}|$$

$$D_{\text{out}}^m = |\{(j, n) \rightarrow (j', n')|(j, n) \rightarrow (j', n') \in F_{\text{act}}, y^m_j = 1\}|$$

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

\textbf{Input:} $T, J, M, \{y^m_i\}$
\textbf{Output:} $\{x^l_{j,n}, k^l_{(j,n) \rightarrow (j',n')}\}, T_{OES}$

1. Initialize $F_{act}$ and $F_{pend}$ to $\emptyset$
2. $x^1_{j,n} \leftarrow 1, \forall j \in J$
3. for $i \in [T]$ 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. $x^l_{j,n} \leftarrow 1$ if $(j, n)$ is available
      8. if $(j, n)$ finished at $t - 1$ then
         9. for $(j, n) \rightarrow (j', n') \in F_{act}$ 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 $F_{pend}$
                else
                    12. add $(j, n) \rightarrow (j', n')$ to $F_{act}$
         13. for every flow $(j, n) \rightarrow (j', n')$ finished at $t - 1$ do
            14. if $(j, n + 1) \rightarrow (j', n' + 1) \in F_{pend}$ then
                15. remove $(j, n + 1) \rightarrow (j', n' + 1)$ from $F_{pend}$
                16. add $(j, n + 1) \rightarrow (j', n' + 1)$ to $F_{act}$
         17. for $m \in [M]$ do
            18. calculate $\Delta_{in}^m$ and $\Delta_{out}^m$ according to (18) (19)
            19. for $(j, n) \rightarrow (j', n') \in F_{act}$ do
                20. $k^l_{(j,n) \rightarrow (j',n')} \leftarrow \min \{B_{in}^m / \Delta_{in}^m, B_{out}^m / \Delta_{out}^m\}$
                where $y_{j,n}^m = 1$ and $y_{j',n'}^m = 1$
   21. return $\{x^l_{j,n}, k^l_{(j,n) \rightarrow (j',n')}\}, T_{OES}$

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 $\Delta_{in}^m$ and one-iteration egress flow degree $\Delta_{out}^m$:

\[
\Delta_{in}^m = \{(j', n') \rightarrow (j, n) | (j, n) \in F_{one\_iter}, y_{j,n}^m = 1\}
\]

\[
\Delta_{out}^m = \{(j, n) \rightarrow (j', n') | (j, n) \in F_{one\_iter}, y_{j,n}^m = 1\}
\]

and the maximum degree $\Delta$:

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

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 $\Delta$ 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 $\Delta$.

**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 $\Delta$. 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, $\mathcal{Y}_0 = \{y^m_i\}_0$, followed by generating a sequence of placements $\mathcal{Y}_1, \mathcal{Y}_2, \ldots$, until a time budget $t$ 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 $i$ machines ($m_1$ 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 $A_{q_s, q_w, q_{ps}, i}$ 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 $Ω(i)$ 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 $\eta_s$ denote the maximal number of samplers that can be hosted by any machine, i.e., $\eta_s = \max_{m \in [M]} \min_{r \in [R]} \omega_{rm} / \omega_{sr}$. Note that placements of graph store servers are given (one on a machine). Similarly, we can define an upper bound on the number of workers and PSs per machine, $\eta_w$ and $\eta_{ps}$. For every possible combination of $q_s \in \{0\} \cup [\min\{\frac{J_s}{|J_s|}\}, 1]$ and $q_w \in \{0\} \cup [\min\{\frac{J_w}{|J_w|}\}, 1]$ and $q_{ps} \in \{0\} \cup [\min\{\frac{|J_{ps}|}{|J_{ps}|}\}, 1]$ and $\eta_s$ and $\eta_w$ and $\eta_{ps}$, 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 $Ω(1)$, and set $A_{q_s, q_w, q_{ps}, 1} = \{(q_s, q_w, q_{ps}, 1)\}$.

Next, we iteratively construct $Ω(i)$ based on $Ω(i - 1)$ until finding a complete feasible solution of placing all $|J_s|$ samplers, $|J_w|$ workers and $|J_{ps}|$ PSs onto the machines. For each $[q_s, q_w, q_{ps}, i - 1]$, we check 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 $i - 1$ machines: $A_{solution} = A_{q_s, q_w, q_{ps}, i - 1} \cup \{([J_s] - q_s, [J_w] - q_w, [J_{ps}] - q_{ps}, i)\}$. Otherwise, we find each feasible placement $(q'_s, q'_w, q'_{ps}, i)$ with $q'_s \in \{0\} \cup [J_s] - q_s$, $q'_w \in \{0\} \cup [J_w] - q_w$, $q'_{ps} \in \{0\} \cup [J_{ps}] - q_{ps}$.
Algorithm 2: Exploratory Task Placement - ETP

Input: $T, J, M, R, \{C_m^r\}, \{w_j^r\}$

Output: $Y_{\min}$

1. $Y_0 \leftarrow \text{IFS}(J, M, R, \{C_m^r\}, \{w_j^r\}); Y_{\min} \leftarrow Y_0$
2. $\min\text{makespan} = \text{OES}(T, J, M, N, Y_0)$
3. for $z \in \{0, 1, \ldots, I - 1\}$ do
   4. randomly select a task $j$ from $J \setminus J_g$
   5. construct $M_{\text{avail}}$ of $j$
   6. randomly select a machine $m$ from $M_{\text{avail}}$
   7. construct new placement solution $Y''$
   8. $T_{Y''} = \text{OES}(T, J, M, N, Y'')$
   9. $\pi(Y_z \rightarrow Y'') \leftarrow \min\{1, \exp(\beta \text{cost}(Y_z) - \beta \text{cost}(Y''))\}$
   10. if $\text{rand}() \leq \pi(Y_z \rightarrow Y'')$ then
       11. if no resource violation with $Y_{z+1}$ and
           12. $T_{Y''} < \min\text{makespan}$ then
               13. $\text{min}\text{makespan} \leftarrow T_{Y''}$
               14. $Y_{\min} \leftarrow Y_{z+1}$
       15. else
           16. $Y_{z+1} \leftarrow Y_z$
   17. return $Y_{\min}$

$q_w^r \in \{0\} \cup [J_w - q_w], \text{ and } q_{ps}^r \in \{0\} \cup [J_{ps} - q_{ps}]$ that satisfies capacity constraint on machine $m$; and if $[q_m^r + q_w^r + q_{ps}^r, \text{ is not in } \Omega(i)$ yet, we add it into $\Omega(i)$, and set $A(q_m^r, q_w^r, q_{ps}^r, \text{ be the union of } A(q_m^r, q_w^r, q_{ps}^r, \text{ and } \Omega(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 exactly one $[q_m^r, q_w^r, q_{ps}^r]$ triple. The construction of each $\Omega(i)$ takes up to $O([J_w |J_{ps}| N |R|)$ 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($Y$) defined on the overall training makespan of placement $Y$.

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 $Y$ using Alg. 1, with time and traffic volume drawn from the distributions, and derive the training makespan $T_{Y'}$. The cost of placement $Y$ is:

$$\text{cost}(Y) = T_{Y'}^{(1 + \sum_{m \in [M], r \in [R]} \max\{\sum_{j \in J} w_j^r y_j^m - C_m^r, 0\})} \quad (21)$$

where $T_{Y'}$ 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 $Y_z$ to another $Y_{z+1}$, for a total of $I$ transfers (the time budget). Give $Y_z$, we uniformly randomly sample a task $j \in J \setminus J_g$. Let $M_{\text{avail}}$ denote the set of machines other than the one where $j$ is placed in $Y_z$, which can host $j$ adhering to relaxed resource capacity constraints:

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

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

Next, we uniformly randomly choose one server $m \in M_{\text{avail}}$ to move $j$ to, and come up with the new placement solution $Y'$. 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):

$$\pi(Y_z \rightarrow Y') = \min\{1, \exp(\beta \text{cost}(Y_z) - \beta \text{cost}(Y'))\} \quad (23)$$

With probability $\pi(Y_z \rightarrow Y')$, we use $Y'$ as $Y_{z+1}$; if $\text{cost}(Y') \leq \text{cost}(Y_z)$, we accept $Y'$ as $Y_{z+1}$ (probability is 1); otherwise, we still accept $Y'$ as the next state with probability $\pi(Y_z \rightarrow Y')$ (for exploration) and maintain $Y_{z+1}$ the same as $Y_z$ with probability $1 - \pi(Y_z \rightarrow Y')$.

Our state transition as designed above ensures that the probability of visiting $Y'$ is linear to $\exp(-\beta \text{cost}(Y'))$ [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: $T, N, J, M, R, \{C^r_n\}, \{w^r\}$
Output: $\gamma'_{min} \leftarrow \{x^t_{j,n} \}, \{k^t_{(j,n) \rightarrow (j',n')}\}$
1 $\gamma_{min} \leftarrow \text{ETP}(T, J, M, R, \{C^r_m\}, \{w^r\})$
2 $\{x^t_{j,n} \}, \{k^t_{(j,n) \rightarrow (j',n')}\}, \text{TOES} \leftarrow \text{OES}(T, J, M, N, \gamma_{min})$
3 return $\gamma'_{min} \leftarrow \{x^t_{j,n} \}, \{k^t_{(j,n) \rightarrow (j',n')}\}$

the best placement $\gamma_{min}$ and then use Alg. 1 to decide the task and flow schedules $\{x^t_{j,n} \}, \{k^t_{(j,n) \rightarrow (j',n')}\}$ based on $\gamma_{min}$, 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 Queuing 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 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][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.

| 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 collocates 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 $T'_{\gamma}$ during search, and set the exploratory time budget 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\}.
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 tested 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.
