AdaGL: Adaptive Learning for Agile Distributed Training of Gigantic GNNs

Ruisi Zhang, Mojan Javaheripi
University of California, San Diego
{ruz032,mojan}@ucsd.edu

Zahra Ghodsi
Purdue University
zahra@purdue.edu

Amit Bleiweiss
Intel Lab
amit.bleiweiss@intel.com

Farinaz Koushanfar
University of California, San Diego
farinaz@ucsd.edu

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10248003

***Abstract*—Distributed GNN training on contemporary massive and densely connected graphs requires information aggregation from all neighboring nodes, which leads to an explosion of inter-server communications. This paper proposes AdaGL, a highly scalable end-to-end framework for rapid distributed GNN training. AdaGL novelty lies upon our adaptive-learning based graph-allocation engine as well as utilizing multi-resolution coarse representation of dense graphs. As a result, AdaGL achieves an unprecedented level of balanced server computation while minimizing the communication overhead. Extensive proof-of-concept evaluations on billion-scale graphs show AdaGL attains $\sim 30-40\%$ faster convergence compared with prior arts.**

I. Introduction

Graph Neural Networks (GNNs) have reportedly been achieving state-of-the-art performance in various important applications including chip design [1], recommender systems [2], and drug discovery [3]. Emerging large-scale graph datasets like social networks contain billions of nodes and edges that require multi-server training due to memory size and bandwidth constraints. Typical distributed GNN training is composed of two consecutive steps. In the first (pre-processing) step, the original large-scale graph is divided into smaller subgraphs. In the second step, each server loads its allocated subgraph and starts the training. During training, each node aggregates its neighbors' features to update their embedding and subsequently the GNN weights. However, the neighboring nodes may be assigned to different servers during allocation, which results in large inter-server communications that slow down training.

A number of recent work reduce the communication overhead of distributed GNN training by optimizing the graph allocation algorithms. These algorithms mainly fall into two categories: (1) network-based allocation, and (2) multi-step allocation. Network-based algorithms [4], [5] divide the graph based on the inter-server connection topology and reduce the communication overhead by using faster network links [6] or modifying the communication protocols [5]. Such methods, however, are designed for a specific topology and are hard to generalize for new scenarios. In addition, network-based allocation is agnostic to the possible heterogeneity in the servers. Multi-step allocation is used in METIS [7] and BGL [8]. Such algorithms first coarsen the dense graph, then apply graph allocation to minimize inter-server edges, and finally uncoarsen to reach the original graph. METIS uses edge matching for coarsening which is extremely memory-bound, thus hindering scalability to larger graphs. BGL uses one round of breadth-first-search to randomly coarsen nodes into large blocks then assigns each block to servers using heuristic methods. While BGL can scale to billion-scale graphs, the blocks in the coarsened graph are prohibitively large, causing high inter-server communications that cannot be reduced with heuristics.

This paper proposes AdaGL, an end-to-end framework for agile distributed GNN training that automatically customizes the training configuration according to the underlying (heterogeneous) hardware resources. We formalize graph allocation as an optimization problem with well-defined objectives and suggest a learning-based optimizer to iteratively search for and find the best node assignments. We further devise an adaptive initialization for our optimizer to ensure fast convergence to a near-optimal solution.

AdaGL framework comprises three key stages, namely, graph coarsening, allocation optimization, and refining, as shown in Figure 1. AdaGL encapsulates a novel coarsening strategy that transforms the original graph into a concise representation. Our coarsening algorithm iteratively identifies a set of core nodes, then leverages tree-based search to aggregate information from node neighborhoods within core nodes. Notably, the proposed coarsening method scales linearly with the number of graph nodes, thereby enabling us to optimize allocation for billion-scale graphs.

Once the coarsened representation is derived, AdaGL optimizer finds a good allocation by minimizing a custom objective function. We devise the objective function based on two key factors that affect distributed GNN training performance, namely, computational balance among (heterogeneous) servers and inter-server communication overhead. Measuring communication, however, requires running distributed training which incurs a large cost during optimization. Therefore, we develop a simulator that instantly and accurately predicts the data transmit time for a given graph allocation. Our simulator can model various infrastructures by interpolating on a few measured samples of inter-server communication.

Once the optimizer converges, AdaGL iteratively maps the allocations on the coarsened representation to the original graph. Interleaved in this iterative process, we refine the node allocations via local adjustments to further minimize inter-server connections. In summary, our contributions are as follows:

- Proposing AdaGL, a highly scalable framework for rapid distributed GNN training. AdaGL encompasses a new formulation of graph allocation with a custom objective function to ensure balanced computation and minimized communication in (heterogeneous) distributed settings.
- Devising a learning-based optimizer for AdaGL to search for and find the best graph allocation. AdaGL optimizer incorporates the following to speedup the search: 1) an adaptive initialization based on greedy heuristic methods, and 2) an accurate and low-cost simulator to model inter-server data transmit time for evaluating candidate allocations.
- Ensuring scalability of AdaGL by designing a memory-efficient, multi-resolution graph coarsening algorithm that scales linearly in the number of nodes.

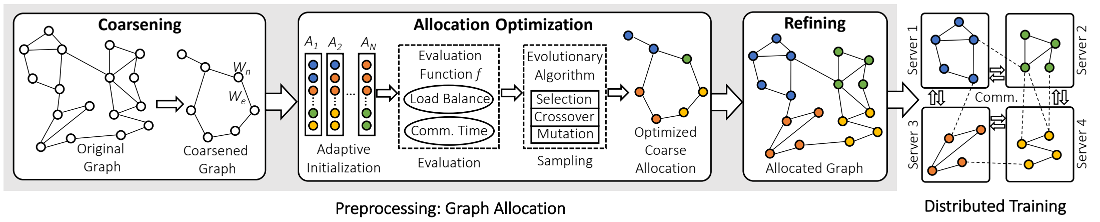


Fig. 1: Overview of AdaGL distributed GNN training pipeline. The original graph is first transformed into a coarsened weighted representation. We then optimize graph allocation across servers to minimize training communication while ensuring a balanced load. Finally the allocated coarsened graph is refined to achieve the original representation and sent to servers for distributed training.

- Conducting extensive experiments on AdaGL with graph datasets of varying sizes, and achieved 30% lower communication time on billion-scale graph datasets compared with state-of-the-art.

II. Preliminaries on Distributed GNN Training

Distributed stochastic GNN training comprises two consecutive stages. During the pre-processing stage, the original large-scale graph is divided into several disjoint subgraphs and assigned to different servers. During (stochastic) training, a sampler randomly selects a subset of training nodes and their k-hop neighbors to be fed into the GNN model at a given iteration. To perform the GNN computations, each server should thus retrieve the node and edge features for its missing neighbor nodes from other remote machines. Due to dense connections in contemporary graphs, a large number of features have to be moved between servers, raising two major challenges, i.e., *heavy communication overhead* and *GPU underutilization*.

Heavy Communication Overhead. To analyze this overhead, we randomly allocate a given graph among four servers and report the training time for one epoch in Table I. Here, the communication time[1] T_{comm} includes fetching node features from remote servers and constructing the sampled graph and the computation time T_{comp} is for training the GNN model on the constructed graph. As shown, random graph allocation can lead to excessive communication overhead that takes up $70\%-80\%$ of the total training time.

Several prior works reduce the overhead by altering the communication procedure. P3 [9] transmits node IDs instead of features to reduce communication. DGCL [5] proposes new communication protocols with fast link transmissions to avoid contention. Our paper, instead, focuses on reducing the communication overhead by optimizing the allocation to reduce the inter-server edges. As a result, we systematically reduce a significant fraction of the required communication between servers during training.

Benchmark	T_{comm} (s)	T_{comp} (s)
OGB-Arxiv	29.8	7.1
OGB-Products	87.9	19.9

TABLE I: Per-epoch time breakdown of distributed GNN training.

GPU Underutilization. We define GPU utilization as the portion of training time when the GPU is actively used: $\frac{T_{\text{comp}}}{T_{\text{comm}}+T_{\text{comp}}}$. From Table I, most of the time is spent on communication handled by the CPU. Therefore, GPU utilization is lower in distributed GNN training compared to other deep neural networks. Recent works propose better execution pipelines to increase utilization. P3 [9] uses push-pull parallelism that switches between data and model parallelism to improve GPU efficiency. BGL [8] divides GNN training into asynchronous pipeline stages with isolated resources.

[1]Throughout the paper, we use the terms "communication time" and "data transmit time" interchangeably.

In AdaGL, GPU usage improvement is a natural byproduct of our proposed allocation; by reducing inter-server edge connections, we ensure that the majority of data required for the GPU computations is locally available or can be fetched with low overhead. Thus we reduce the communication time T_{comm} and increase GPU utilization. We note that the proposed methods for improving the execution pipeline are orthogonal to AdaGL and can be combined with our allocation optimization to further boost the training performance.

III. Problem Formulation and Objective

We identify three key goals for our graph allocation to address existing challenges, namely, maintaining load balance, minimizing edge cuts, and ensuring scalability to billion-scale graphs.

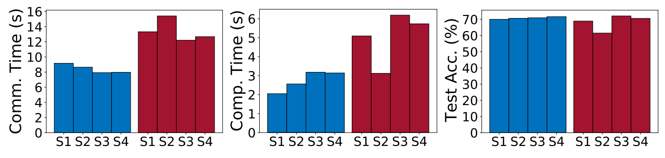
Load Balance. Our graph allocation fairly divides the computational load among distributed servers to ensure proper synchronization, maximal GPU usage, and minimum stall time on all machines during training. We define a penalty for load imbalance $\mathcal{L}$ in Eq. 1 for a weighted graph with V nodes and node weights W_n, distributed among M servers. Here, A is a vector holding the server assignments for all nodes. L_i is the summation of node weights assigned to the i-th server which simplifies to node count for unweighted graphs. B_i is the balanced number of nodes in each subgraph. For homogeneous machines $B_i = \frac{V}{M}$ and for heterogeneous servers $B_i = V \times \frac{P_i}{\sum_{j=1}^{M} P_j}$, where P_i is the computational capacity of the i-th server.

$$\mathcal{L}(A, W_n) = \frac{1}{M-1} \sum_{i=1}^{M} \left| \frac{L_i}{B_i} - 1 \right| \tag{1}$$

To show the effect of load balance on distributed training, we visualize several performance metrics in Fig. 2, namely communication time, computation time, and test accuracy. The evaluations are performed on the OGB-Arxiv dataset [10], trained on four servers for 50 epochs. We compare AdaGL with the popularly used METIS [7] allocation strategy which assigns the nodes to servers by minimizing edge cuts with less emphasis on load balancing.

We observe several downsides of load imbalance in METIS. Fig. 2a, 2b show that METIS has a higher communication and computation time that is imbalanced across different servers. This time imbalance means some machines need to stall computation while others finish data transmission or computation, leading to elongated training time and under-utilization of resources.

Additionally, as shown in Fig. 2c, the test accuracy of METIS is lower on some subgraphs compared to others which causes slower training convergence among all servers. Compared to METIS, AdaGL successfully alleviates the aforesaid shortcomings by directly incorporating load balancing in the allocation objective.



(a) Communication Time (b) Computation Time (c) Test Accuracy

Fig. 2: Various performance metrics of distributed GNN training, measured for AdaGL (blue bars) and METIS [7] (red bars) allocation algorithms. Here the load imbalance $\mathcal{L}$ and edge cuts $\mathcal{E}$ are 0.01 and 0.09 for AdaGL, and 0.03 and 0.11 for METIS.

Edge Cuts. When distributing the graph among servers, the number of edge cuts directly correlates with inter-server communications during training. We therefore minimize the frequency of edge cuts $\mathcal{E}$ as formalized in Eq. 2. Here, E denotes the sum of all edge weights W_e in the original weighted graph and E_i is the sum of edge weights contained inside the i-th subgraph. The definitions also apply to unweighted graphs by simply assigning a weight of 1 to all edges.

$$\mathcal{E}(A, W_e) = \frac{E - \sum_{i=1}^{M} E_i}{E} \quad (2)$$

As shown in Fig. 2a, AdaGL successfully lowers the communication time by reducing edge cuts. If naively applied, reducing edge cuts can lower the validation accuracy in certain subgraphs since the sampled training nodes are limited to a small, clustered subgraph and lack diversity. AdaGL circumvents this issue by enforcing load balance in the graph allocation objective in addition to edge cut minimization. As such, compared to METIS, AdaGL has a more uniform validation accuracy across subgraphs as illustrated in Fig. 2c.

Scalability. The best existing graph allocation techniques like METIS suffer from a high memory complexity of $\mathcal{O}(V^2)$ during coarsening. This memory bottleneck hinders their applicability to large-scale graphs. AdaGL proposes a multi-resolution graph coarsening technique with $\mathcal{O}(V)$ space complexity, which allows easy scalability of our allocation for mega graphs.

IV. System Overview

Fig. 1 shows a high-level overview of AdaGL workflow comprising three main steps, namely, *coarsening*, *allocation optimization*, and *refining*. We propose a highly scalable coarsening algorithm which transforms the original complex graph into a less densely connected variant for easier allocation. We formulate graph allocation as an optimization problem with custom objectives designed for improving distributed GNN training performance, namely, load balancing and minimizing inter-server communications. We then solve the optimization problem for the coarsened graph using an evolutionary algorithm. To ensure fast convergence to a good solution, we propose an adaptive initialization for the samples in the evolutionary algorithm. Finally, the allocated coarsened graph is refined to reach the original edge density. Below we describe AdaGL steps in detail.

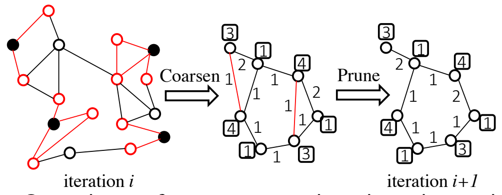


Fig. 3: Overview of one coarsening iteration with 1-hop neighbors (i.e., $k = 1$). Here, sampled core nodes are filled with black, and the nodes/edges which will be removed are marked with red. The numbers denote node/edge weights.

A. Coarsening

Coarsening maps the original densely-connected graph to a weighted variant with fewer nodes and edges. In the coarsened graph, the weights assigned to each node indicate the number of nodes it represents from the original graph. Similarly, the weights assigned to the edges denote the edges connecting two coarsened nodes.

Existing graph coarsening strategies include the heavy-edge matching adopted in METIS [7] which sequentially transforms the original graph into smaller ones by finding sets of matching edges, and collapsing two vertices of each matched edge into one node. Other works apply spectral analysis to the graph to sparsify the connections [11]. These coarsening techniques directly operate on the adjacency matrix of the graph, which creates a memory complexity that is quadratic (or higher order) with respect to the number of nodes. We take a different approach by applying a multi-resolution breadth-first-search (BFS) to iteratively coarsen the graph. As a result, we traverse the training nodes in the graph only once, which reduces the complexity to $\mathcal{O}(V)$.

Fig. 3 demonstrates one iteration of our coarsening algorithm. First, a subset of the training nodes is randomly selected as core nodes. The k-hop neighbors of these core nodes are then removed and replaced with aggregated node/edge weights in the coarsened graph. Each core node will be assigned two sets of weights W and W_t, indicating the total number of coarsened nodes versus coarsened training nodes, respectively. To further reduce the edge density, we drop edges with smaller weights at the end of each iteration. Specifically, only edges with the largest weights that sum to $1.1\times$ the average edge weights are kept while the remaining edges are pruned. By default, we run the coarsening for 10 iterations with $k = 3$.

We note that AdaGL's coarsening method is fundamentally different from BGL [8] which uses BFS over random nodes merely to find connected units within the original graph, dubbed blocks. The obtained blocks are therefore highly dependent on the underlying connectivity in the original graph dataset and may lead to large communication overheads after allocation. AdaGL applies multiple rounds of BFS on the *training* nodes to aggregate the information from node neighborhoods in a new weighted representation. The number of iterations is automatically adjusted for any given graph structure, and the weight accumulation ensures the final coarse representation accurately represents the original graph without loss of information.

B. Allocation Optimization

Graph allocation across M servers can be formulated as an optimization problem over a discrete search space. Each element in the search space is a potential configuration of the graph

Algorithm 1: AdaGL Optimization Algorithm

```
Input: sample size $N$, iterations $I$, evaluation function $f$ (Eq. 4)
Output: optimized graph allocation $A^*$
$\mathbb{S}^0 = \{A_n^0\}_{n=1}^N$                    // Adaptive Initialization
for $i$ = 0 to $I$ do
    $\mathbb{F}^i = \{f(\mathbb{S}^i[n])\}_{n=1}^N$                    // Evaluation
    $\mathbb{S}^{i+1} = \text{mutate}(\text{crossover}(\text{select}(\mathbb{S}^i \mid \mathbb{F}^i)))$  // Sampling
return $A^* = \text{argmin}(\{\mathbb{F}^i\}_{i=1}^I)$
```

Algorithm 2: Adaptive allocation Initialization

```
Input: sorted node list $V_s$, evaluation function $\tilde{f}(\cdot)$ (Eq. 3)
Output: graph allocation $A$
$A$ = []
for $v = 0$ to $|V_s|$ do
    for $i = 0$ to $M-1$ do
        $e_i = \tilde{f}([A,i])$
    $A$.append(argmin $[e_0, e_1, ..., e_{M-1}]$)
return $A$
```

allocation and can be represented by a vector $A = [a_1, a_2, ..., a_V]$. Here, V denotes the total number of nodes in the coarsened graph, and $a_i \in \mathbb{Z}_M$ is the integer server ID assigned to the i-th node. Traditional graph allocation algorithms like METIS [7] use heuristic methods to find a suitable server assignment A. However, heuristic methods may fall into local minima and fail to locate the optimal allocation. More recently, reinforcement learning (RL) was used to learn the allocation [12]. While this method performs well on smaller graphs, training the RL agent incurs significant computation cost as the dimensionality of the search space, i.e., the number of nodes, increases.

Alternatively, evolutionary algorithms have shown competitive performance as RL on challenging optimization tasks, while enjoying higher scalability and significantly lower training overhead [13]. Inspired by this, we develop our graph allocation optimizer based on evolutionary algorithms. Our optimizer operates on a population of samples, each corresponding to a graph allocation A. Using evolutionary algorithms, we iteratively and adaptively improve the quality of our generated samples to find a (near)optimal allocation. In this context, optimally is evaluated using our custom objective function formulated based on the performance metrics for distributed GNN training described in Section III. Algorithm 1 outlines the pseudocode of AdaGL optimizer. At a high level, our optimizer performs three main steps, namely, *adaptive initialization*, *evaluation*, and *sampling*. Below, we describe each step in detail.

Adaptive Initialization. The naive way to initialize samples for our optimizer is to assign nodes randomly to each server. However, due to the extremely large search space, such random initialization contains several suboptimal samples that will slow down the search. We propose an adaptive initialization that directly incorporates the performance of distributed GNN training as characterized in Section III. This ensures that our first set of samples is strategically close to the (unknown) optimal configuration.

Our initial samples $\mathbb{S}^0$ are found by minimizing the objective $\tilde{f}(\cdot)$ in Eq. 3 with different coefficients α, β, γ. For a given graph allocation A, the first term measures load imbalance among all nodes in the coarsened graph with node weights W_n. The second term measures the load imbalance only among the training nodes with weights W_n^t. Finally, the third term measures the edge cut when allocating the coarsened graph with edge weights W_e.

$$\tilde{f}(A) = \alpha \cdot \mathcal{L}(A, W_n) + \beta \cdot \mathcal{L}(A, W_n^t) + \gamma \cdot \mathcal{E}(A, W_e) \quad (3)$$

To minimize $\tilde{f}(\cdot)$, we develop a greedy heuristic search algorithm inspired by [14]. Our heuristic optimization shown in Algorithm 2, starts with an empty allocation and a sorted node list in descending order of node weights. The greedy algorithm then progressively builds the allocation vector A by iterating over the node list. For each new node, we assign it to the server with minimum $\tilde{f}(\cdot)$ from Eq. 3, calculated using all prior node assignments.

Evaluation. To measure the optimally of different graph allocations, we define a custom evaluation function based on load balancing $\mathcal{L}(\cdot)$ and communication time $\mathcal{T}(\cdot)$ as shown in Eq. 4. We note that compared to edge cuts, communication time more accurately reflects the performance of distributed GNN training. This is because the communication time is determined by training nodes (and their neighbors) while edge cuts is measured over all nodes which is a higher bound on the required communications.

$$f(A) = \alpha \cdot \mathcal{L}(A, W_n) + \beta \cdot \mathcal{L}(A, W_n^t) + \gamma \cdot \mathcal{T}(A) \quad (4)$$

Measuring the communication time for a given graph allocation requires running at least one iteration of the distributed training. This incurs a non-negligible cost which prevents us from swiftly evaluating various graph allocations. We therefore develop an accurate simulator which instantly predicts the data transmit time without running distributed training on the allocated graph. In distributed training, the transmission time t from one server to the other can be formulated using the α-β model [4] as $t = \alpha + \beta n$ where n denotes the number of transmitted features.

Given a configuration of servers for distributed training, we can learn the transmit parameters α, β using few examples of real-world data transmit time. To ensure the learned parameters are robust to different configurations, we use different variations of the graph allocations with different load imbalance $\mathcal{L} \in [0, 0.3]$ to train the α, β parameters. After learning α and β, we can simulate the communication time for each subgraph G_i. Given training nodes N_t, we iterate over batched training nodes in $G_i \cap N_t$ and use a neighborhood sampler to gather nodes N_s within k hops of the current node batch. Data transmit time $\mathcal{T}_i$ for the current batch is then calculated as

$$\mathcal{T}_i = \beta \times W_r + \alpha \times r \quad (5)$$

where r is the number of remote servers interacting with G_i through N_s. Here, $W_r = \sum_{v \in N_s \setminus G_i} W_e[v] \times W_n[v]$ where $N_s \setminus G_i$ is the subset of sampled nodes that are not on server i, $W_n[v]$ is their node weights, and $W_e[v]$ is the edge weights connecting them to the nodes in G_i. The above $\mathcal{T}_i$ value is accumulated over all batches to achieve the communication time for G_i. The final data transmit time $\mathcal{T}$ is the maximum communication time of all graph allocations.

Fig. 4 shows our predicted communication time versus real measurements of data transmit time, evaluated on random allocations of the OGB-Arxiv dataset. As seen, our simulation closely follows the ground-truth values, thus providing a reliable estimator for communication in distributed GNN training.

Sampling. We adopt evolutionary algorithms to guide our sampling where the best-performing samples in each iteration are used to generate new ones. This allows us to gradually learn

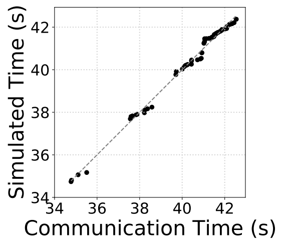


Fig. 4: Simulated communication time versus real measurements for various allocations of OGB-Arxiv dataset. The mean absolute prediction error is only 0.6%.

and combine desirable features from multiple best-performing samples to find a good solution. Evolutionary algorithms perform three consecutive steps to evolve and update the population of samples, namely, *selection*, *crossover*, and *mutation*.

During selection, a subset of previously evaluated samples is picked to be used for new sample generation. We construct a probability distribution over the samples as $p_i = \frac{f_{\max} - f_i}{\sum_{i=1}^{N}(f_{\max} - f_i)}$, which promotes selection of better performing candidates. Here, p_i is the probability of selecting sample i and f_i is its evaluation score using Eq. 4. Since the optimization goal is to minimize the evaluation score, we subtract all scores from the maximum observed value denoted by $f_{\max}$. Our formulation allows samples with a lower evaluation score to have a higher probability of selection, as desired.

Once N samples are selected using the above distribution, they are randomly divided into pairs, and each pair undergoes crossover to generate a new pair of child samples. During crossover, the elements inside parent samples are randomly swapped. This allows us to exploit the node assignments observed in good samples, which can lead to a better allocation. Finally, the generated samples undergo mutation which randomly tweaks their elements. In the search for better graph allocations, mutation randomly explores new node assignments which have not been seen in prior evaluations.

C. Refining

Refining maps the coarsened graph allocation back to the original graph. Inspired by METIS [7], we apply local improvements during refining by allowing minimal node exchange between allocations to further reduce edge cuts while maintaining load balance. To decide on an exchange, we count the number of neighbors the node shares with other servers and move it to the server with the maximum number of neighbors. Exchanging allocations can be time-consuming as the graph becomes larger. We therefore use early stopping criteria on load balance to limit the number of node exchanges. Additionally, we only exchange nodes with weight values between $1-5\%$ of the total weight in the current graph allocation. Nodes with a larger weight than this range significantly alter the overall load balance while nodes with a smaller weight barely affect the edge cuts. These nodes are thus not suitable for exchange.

V. Experiment

A. Evaluation Setup

Environment. Our homogeneous setting includes a CPU/GPU cluster with four nodes. Each node has one NVIDIA RTX 8000 GPU, 256 GB RAM memory, and Intel Xeon CPUs connected via a 10Gbps Ethernet interface. Our heterogeneous setting has four nodes with Nvidia Titan Xp GPUs and Intel Xeon CPUs, where two nodes have 6GB GPU memory, and the other two nodes have 12GB GPU memory.

Datasets. We benchmark OGB-Arxiv ($V = 0.2$M, $E = 1.2$M), OGB-Products ($V = 2.4$M, $E = 124$M), OGB-Papers ($V = 111$M, $E = 1.6$B), and the largest opens-source graph dataset MAG ($V = 243$M, $E = 1.7$B). Here, V and E are the number of nodes and edges in the dataset, respectively.
GNN Models. For all OGB datasets [10], we use the 3-layer GraphSage [15] following the setting in BGL [8]; For the MAG dataset [10], we use the 5-layer R-GAT as implemented in [10].
Baselines. We compare with the following graph allocation methods: *Hash* allocation adopted in P3 [9] uses a fixed hash function to randomly map nodes to different servers. *Heuristic* allocation uses the greedy algorithm in [16] to minimize the objective function in Eq. 3. *METIS* [7], *optimized METIS* [17], and *BGL* [8] use coarsening with heuristic allocation to minimize edge cuts.

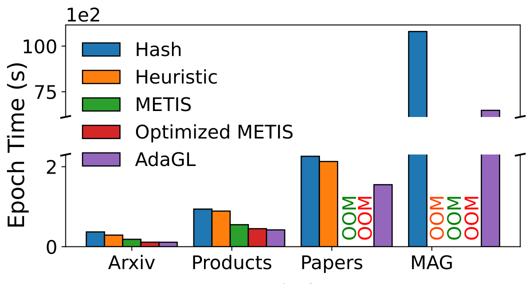
B. Results

Homogeneous Servers. Fig. 5 demonstrates the epoch and communication time for various graph allocation methods when all servers have the same configuration. The evaluated methods require the same number of training epochs to converge to similar final test accuracy. As seen, *METIS* can not be used to allocate large-scale graphs, i.e., OGB-Papers and MAG, due to its quadratic space complexity which results in out-of-memory errors. Since *BGL* code is not open-source, we only compare with benchmarks evaluated in their paper and scale the numbers according to the reported time for *Hash* (random) allocation.

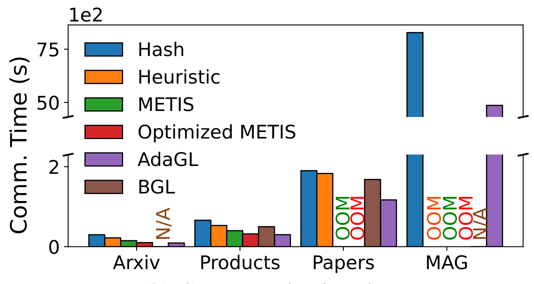
Intuitively, *Hash* allocation incurs the largest epoch and communication time as it randomly assigns nodes, resulting in a high node retrieval time across servers. *Heuristic* allocation applied to the original graph also results in a sub-optimal solution that is on average 46% slower than AdaGL. This is due to the extremely high dimensional optimization space when that graph is not coarsened. Compared with BGL, AdaGL achieves, on average, 30% lower communication time. While *optimized METIS* performs on par with AdaGL in smaller graphs, it fails to scale to larger graphs. Notably, AdaGL is the first graph allocation algorithm that successfully scales to the largest open-source graph dataset, i.e., MAG, achieving $1.78\times$ faster training compared to the only previously feasible allocation, i.e, random.
Heterogeneous Servers. Table II shows the epoch and communication time for distributed training on servers with heterogeneous GPU memories. The per-server computation is determined by the allocated number of training nodes. For both AdaGL and *Hash*, the heterogeneous server memories are directly taken into account when determining the per-server node allocation. AdaGL further minimizes communication time during graph allocation and thus achieves 37.5% faster training compared with *Hash*, on average. *METIS* does not incorporate the server capacity in graph allocation, and as such, it takes a longer time for the server with a smaller capacity to iterate over its assigned subgraph. Compared with *METIS*, AdaGL graph allocation trains 33.1% faster.

C. Analysis and Discussion

We evaluate different design choices for AdaGL components on the example OGB-Arxiv dataset. For each setting, all components follow AdaGL's design in Fig. 1 and only one module is changed.
Effect of Coarsening Method. We analyze the performance of distributed GNN training when different coarsening strategies



(a) Epoch time



(b) Commmunication time

Fig. 5: (a) Epoch and (b) communication time of allocation methods in the homogeneous setting. "OOM" denotes out-of-memory errors. BGL only reports communication time on Products and Papers.

	Method	T_{comm} (s)	T_{epoch} (s)
Arxiv	*Hash*	22.6	32.1
	METIS	20.2	27.5
	AdaGL	14.7	18.3
Products	*Hash*	54.7	82.2
	METIS	49.4	83.0
	AdaGL	35.4	55.8

TABLE II: Epoch and communication time for various allocation methods in heterogeneous setting.

are used in AdaGL, i.e., *heavy edge matching* proposed in METIS [7], *local variance* [11] spectral analysis, and our multi-resolution coarsening in Section V-C. As shown in Table III, AdaGL coarsening outperforms *local variance* and *edge matching* by using BFS to cluster the training nodes while reducing edge cuts. In addition to the timing benefits, AdaGL coarsening is the only method that can scale to larger datasets as heavy edge matching and local variance have $\mathcal{O}(V^2)$ and $\mathcal{O}(V^3)$ space complexity, respectively.

Methods	T_{comm} (s)	T_{epoch} (s)
edge matching	11.5	13.6
local variance	11.2	13.0
AdaGL	9.2	11.1

TABLE III: AdaGL distributed training performance with different coarsening methods.

Effect of Allocation Method. We adopt different allocation algorithms from Section V-A to distribute AdaGL's coarsened graph across remote servers and benchmark training time in Table IV. We observe that applying *Hash* allocation to the coarsened graph results in a worse training performance compared to applying it to the original graph (shown in Fig. 5). This is because *Hash* does not account for node/edge weights in the coarsened graph and thus suffers from heavy load imbalance across servers. AdaGL achieves the best training performance as it directly optimizes our custom objective function to minimize the training time.

Methods	T_{comm} (s)	T_{epoch} (s)
Hash	13.1	15.3
Heuristic	11.4	13.3
AdaGL	9.2	11.1

TABLE IV: AdaGL distributed training performance with different graph allocation algorithms.

Effect of Objective Function. We study how the objective function (Eq. 4) for AdaGL allocation optimizer affects distributed training performance. We compare four variations of the objective coefficients (α, β, γ) in Table V: $(1,0,0)$ minimizing load imbalance for all nodes; $(0,1,0)$ minimizing load imbalance only for training nodes, $(0.05,0.05,0.9)$ minimizing inter-server communication, and $(0.3,0.3,0.4)$ minimizing all three objectives simultaneously, i.e., load imbalance for all nodes and training nodes as well as communication. As seen, balancing all three objectives results in the best allocation. This is because it directly incorporates the key factors in distributed training performance inside the optimization.

VI. Conclusion

In this paper, we propose AdaGL, a highly scalable end-to-end framework for rapid distributed GNN training. AdaGL adaptively learns the optimal graph allocation across distributed servers by incorporating custom objective functions that maintain balanced computation and minimize inter-server communication. Extensive experiments demonstrate AdaGL achieves $30\%-40\%$ faster training convergence over two billion-scale graph datasets when compared with best prior methods.

(α, β, γ)	T_{comm} (s)	T_{epoch} (s)
(1, 0, 0)	12.6	15.1
(0, 1, 0)	10.8	13.2
(0.05, 0.05, 0.9)	10.1	12.9
(0.3, 0.3, 0.4)	9.2	11.1

TABLE V: AdaGL distributed training performance with different objective coefficients in Eq. 4 for graph allocation.

VII. Acknowledgments

This work was supported by NSF Trust-Hub under award number CNS2016737, NSF TILOS under award number CCF-2112665, and Intel PrivateAI Collaborative Research Institute.

References

[1] A. Mirhoseini *et al.*, "A graph placement methodology for fast chip design," *Nature*, vol. 594, no. 7862, pp. 207–212, 2021.
[2] C. Gao *et al.*, "Graph neural networks for recommender system," in *WSDM*, 2022, pp. 1623–1625.
[3] X. Zeng *et al.*, "Toward better drug discovery with knowledge graph," *Current opinion in structural biology*, vol. 72, pp. 114–126, 2022.
[4] A. Tripathy *et al.*, "Reducing communication in graph neural network training," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.
[5] Z. Cai *et al.*, "Dgcl: An efficient communication library for distributed gnn training," in *ECCS*, 2021, pp. 130–144.
[6] A. Li *et al.*, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE TPDS*, vol. 31, no. 1, pp. 94–110, 2019.
[7] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1997.
[8] T. Liu *et al.*, "Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing," *arXiv preprint arXiv:2112.08541*, 2021.
[9] S. Gandhi and A. P. Iyer, "P3: Distributed deep graph learning at scale," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021, pp. 551–568.
[10] W. Hu *et al.*, "Ogb-lsc: A large-scale challenge for machine learning on graphs," *arXiv preprint arXiv:2103.09430*, 2021.
[11] A. Loukas, "Graph reduction with spectral and cut guarantees." *J. Mach. Learn. Res.*, vol. 20, no. 116, pp. 1–42, 2019.
[12] M. H. Mofrad *et al.*, "Revolver: vertex-centric graph partitioning using reinforcement learning," in *CLOUD*. IEEE, 2018, pp. 818–821.
[13] T. Salimans *et al.*, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017.
[14] Z. Abbas *et al.*, "Streaming graph partitioning: An experimental study," *Proc. VLDB Endow.*, p. 1590–1603, jul 2018.
[15] W. Hamilton *et al.*, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.
[16] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," 2012.
[17] D. Zheng *et al.*, "Distdgl: distributed graph neural network training for billion-scale graphs," in *IA3*. IEEE, 2020, pp. 36–44.