LeapGNN: Accelerating Distributed GNN Training Leveraging Feature-Centric Model Migration

Weijian Chen, Shuibing He, and Haoyang Qu, *Zhejiang University;*
Xuechen Zhang, *Washington State University Vancouver*

https://www.usenix.org/conference/fast25/presentation/chen-weijian-leap

This paper is included in the Proceedings of the
23rd USENIX Conference on File and Storage Technologies.

February 25–27, 2025 • Santa Clara, CA, USA

ISBN 978-1-939133-45-8

LeapGNN: Accelerating Distributed GNN Training Leveraging Feature-Centric Model Migration

Weijian Chen[1,2,3,4], Shuibing He[1,2,3,4*], Haoyang Qu[1,2,3,4], and Xuechen Zhang[5]

[1]The State Key Laboratory of Blockchain and Data Security, Zhejiang University
[2]Zhejiang Lab
[3]Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security
[4]Zhejiang Key Laboratory of Big Data Intelligent Computing
[5]Washington State University Vancouver

Abstract

Distributed training of graph neural networks (GNNs) has become a crucial technique for processing large graphs. Prevalent GNN frameworks are model-centric, necessitating the transfer of massive graph vertex features to GNN models, which leads to a significant communication bottleneck. Recognizing that the model size is often significantly smaller than the feature size, we propose LeapGNN, a feature-centric framework that reverses this paradigm by bringing GNN models to vertex features. To make it truly effective, we first propose a micrograph-based training strategy that leverages a refined structure to enhance locality, combined with the model migration technique, to minimize remote feature retrieval. Then, we devise a feature pre-gathering approach that merges multiple fetch operations into a single one to eliminate redundant feature transmissions. Finally, we employ a micrograph-based merging method that adjusts the number of micrographs for each worker to minimize kernel switches and synchronization overhead. Our experimental results demonstrate that LeapGNN achieves a performance speedup of up to 4.2× compared to the state-of-the-art method, namely P^3.

1 Introduction

Motivation. The emerging graph neural networks (GNNs) are designed for learning from graph-structured data. They are widely employed in various graph-related tasks (e.g., vertex classification [13, 52], edge prediction [48, 49], and graph classification [3, 33]) and have shown superior performance compared to traditional graph algorithms in diverse domains, such as recommendation systems [41, 45], social networks analysis [50], and drug discovery [38].

Input graph datasets for GNN training consist of both topology and vertex features [44, 55]. The volume of real-world graph datasets can easily surpass the memory capacity of a single machine. For example, the sizes of the Pinterest [47] and ByteDance [25] datasets are 18 TB and 100 TB respectively. Therefore, GNN models are typically trained on distributed clusters. In distributed GNN training, graph datasets are partitioned and distributed across multiple servers. During each iteration, each worker on the server uses a subgraph as the input to train a local copy of the GNN model. During the training, a large amount of vertex features need to be fetched from remote servers, leading to significant communication bottlenecks [11, 25, 28].

Limitations of the state-of-the-art systems. Many recent works are proposed to reduce the time of remote feature fetching. For the convenience of discussion, we name the existing approaches as *model-centric* GNN frameworks in which vertex features are moved to the GPU servers where GNN models are trained. Specifically, [24, 25, 28, 55] improve the hit rate of local features but compromise the model accuracy (§7.9) using approximation-based methods. Such approaches are unsuitable for scenarios requiring high precision, as even 0.1% accuracy drop in recommendation systems may lead to revenue losses of millions of dollars [2, 10, 23, 53]. Other studies [25, 32, 44] use GPU memory to cache popular vertex features. They are limited by the cache size especially for large graphs. To avoid remote feature fetching, P^3 [11] combines model parallelism and data parallelism based on random hash partitioning. However, it is designed for GNNs that have small hidden dimensions and its performance gain is reduced as the number of hidden layers increases [11, 25]. Because of the deficiencies of the model-centric GNN frameworks, we need a novel solution, which provides high model accuracy and can be applied to a wide range of GNNs.

Our work. In this paper, we propose LeapGNN, *the first feature-centric* GNN framework that reverses the existing *model-centric* paradigm by moving GNN models to the requested vertex features. LeapGNN is motivated by the observation that the size of model parameters is significantly smaller than the volume of vertex features (§3.1), thus transferring model parameters incurs less cost than fetching graph features. However, a naive implementation of this framework could still result in considerable data movement, due to the complex computation dependencies of GNN models and the

[*]Shuibing He is the corresponding author (heshuibing@zju.edu.cn).

need to transmit intermediate data, such as partial aggregation results and activations. In fact, despite being beneficial in certain scenarios, it may increase the data movement by up to 2.59× compared to the model-centric approach (§3.2).

To make LeapGNN truly effective, we devise three optimization techniques. First, we propose to use *micrograph* as the fundamental training unit for each worker (§4). As a more refined unit compared to the traditional *subgraph*, the micrograph offers superior data locality, i.e., there is a higher probability that the root vertex of a micrograph and its fanout neighbors reside on the same GPU server. Consequently, on top of model migration, micrograph-based training streamlines computational dependencies, mitigates the production of intermediate data, and reduces remote feature retrieval (§5.1). Second, we devise a *pre-gathering* method to elevate communication efficiency (§5.2). This approach consolidates multiple feature fetch operations into a single operation, thereby reducing redundant feature transmissions. Finally, we introduce a micrograph merging approach that adaptively allocates micrographs to each worker (§5.3) for reduced model migrations, thus minimizing kernel switches and synchronization overhead.

Contributions. We make the following contributions:

- To the best of our knowledge, we are the first to propose the feature-centric distributed GNN training strategy using model migration to reduce the feature communication overhead.
- We further analyze the new challenges introduced by the naive feature-centric approach and propose three techniques which collectively enhance its efficacy and practicality, without compromising model accuracy.
- We implement LeapGNN on the DGL framework [36] and test it on five datasets with five GNN models on a distributed GPU cluster. LeapGNN achieves up to 4.2× speedup compared to the the state-of-the-art counterpart, P^3 [11]. The codebase of LeapGNN is available at https://github.com/ISCS-ZJU/LeapGNN-AE.

2 Background

This section uses a vertex classification task as an example to illustrate the basic concepts and training process of GNNs.

Input graph datasets. The input data for GNN training includes both the graph topology and the vertex features, as shown in Figure 1. In the example, we use a social network graph where each vertex corresponds to a user and each edge represents a relationship between two users. Each vertex is associated with a vertex feature, which is stored in an embedding vector. The embedding vectors can encode vertex features like age, gender, geographical location, etc. The goal of the GNN task is to predict the preferred topic for each user. Some users in the graph have revealed their preferences on

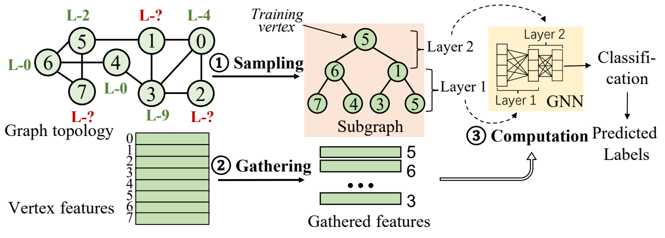


Figure 1: A GNN training example.

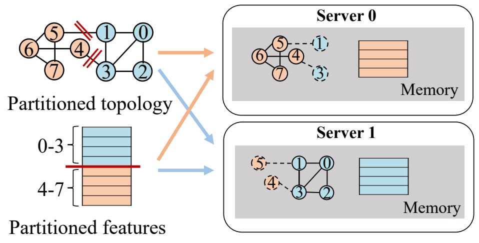


Figure 2: An example of partitioned graph topology and features on two GPU servers.

topics numbered from 0-9 (denoted as 'L-x'), each of which represents a topic, e.g., sports, music, etc. We will use this disclosed information as ground truth for training the GNN model. After training, the trained model is used to predict the preferred topics for users who have not disclosed their preferences.

Subgraph-based GNN training on a single GPU server. GNNs leverage labeled vertices, known as training vertices, to train a multi-layer neural network model across multiple epochs. Similar to traditional DNN training, each epoch involves multiple iterations to process all training vertices once. Each iteration randomly selects a batch of training vertices and consists of three key steps.

The *sampling* step involves k-hop neighbor sampling from the training vertices to generate a k-layer *subgraph*, where k equals the number of model layers. The user-defined parameter 'fanout' dictates the number of neighbors sampled per vertex. For instance, the subgraph depicted in Figure 1 is produced by a 2-hop neighborhood sample from a single training vertex 5 with a fanout of 2. A subgraph may encompass a batch of training vertices, as shown in Figure 3. The *gathering* collects the features of each vertex within the constructed subgraph. The *computation* processes the subgraph layer by layer, beginning with the first layer. For each layer, aggregation operations (like addition or averaging) are conducted on the neighboring features of each vertex. Subsequently, a neural network transformation updates the feature representations. The updated features of the training vertices are fed into a classification network layer to produce predicted labels. Using the true labels, a backward propagation step then follows to refine the model parameters.

Distributed GNN training. For large-scale graphs that exceed the storage capacity of a single GPU server, it is nec-

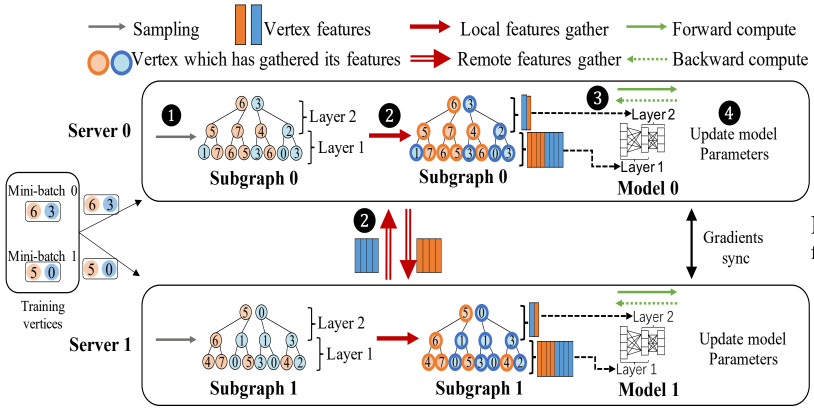


Figure 3: An example of model-centric distributed GNN training approach.

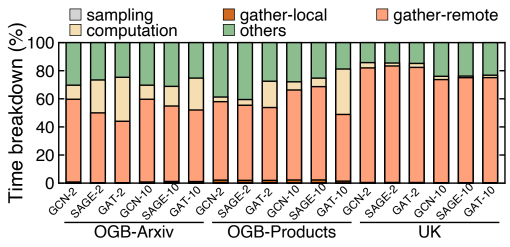


Figure 4: Training time breakdown. 'Model-x' means the fanout is x.'SAGE' denotes GraphSAGE.

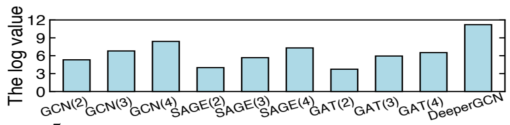


Figure 5: The α value of different models. 'Model (x)' denotes that the number of model layers is x.

essary to distribute the graph's topology and vertex features across multiple servers. Figure 2 depicts the process of partitioning the graph into two parts (denoted by two colors) for distributed GNN training on two servers. Given that the size of the graph topology is smaller than that of the vertex feature embeddings (for instance, 6 GB for topology vs. 53 GB for vertex features in the OGB-Papers100M dataset), several studies [24, 43, 55] opt to redundantly store a subset or the entire topology (e.g. vertices 1 and 3 on server 0) in a small portion of the host memory. This strategy aims to minimize data transmission during the sampling process.

Figure 3 shows a typical distributed GNN training approach that utilizes data parallelism. Each server hosts a complete GNN model copy. At the beginning of each iteration, each model is randomly assigned a disjoint mini-batch of vertices for training. For example, Model 0 is allocated the training vertices $\{6,3\}$, and model 1 receives $\{5,0\}$. Subsequently, the training processes independently perform the 2-hop sampling(❶), feature gathering from local or remote servers (❷), and computation steps(❸). Finally, the parameter gradients from different models are synchronized, and the model parameters are updated (❹). Since the models remain stationary on their respective servers without migration throughout the training process, this method is characterized as *model-centric*.

3 Motivation and Challenges

3.1 Communication Bottleneck in GNN Training

In this section, we study the performance of GNN training using DGL [36] which is a widely used GNN framework in industry. We train three popular GNN models (GCN [20], GraphSAGE [12], and GAT [8]) on three graph datasets [14] (OGB-Arxiv, OGB-Products, and UK). The fanout is two or ten and the number of GNN layers is three, following the settings in the previous works [24,44]. The detailed evaluation methodology is described in §7.

Observation I: Vertex feature gathering causes the communication bottleneck. We utilize PyTorch Profiler [31] to collect time metrics and present the detailed breakdown of the execution times in Figure 4. Notably, gathering remote vertex features consumes between 44% to 83% of the total training time. In comparison, the combined time for sampling and computation stands at an average of merely 11%. Despite the framework's ability to parallelize graph sampling and computation via GPUs, it falls short in mitigating the time required for inter-server communication, particularly when transferring substantial volumes of vertex features. For instance, with GAT [8] on the OGB-Products dataset, a significant 35 GB of vertex features are exchanged per epoch, contrasting sharply with the 0.4 GB of graph topology data. This analysis underscores that remote feature gathering is the predominant factor influencing the end-to-end training time in distributed GNN training.

Observation II: The volume of data transferred for vertex feature gathering is substantially greater than the size of the model. To quantify this, we introduce a ratio α, representing the amount of training data fetched from remote servers per iteration relative to the size of the model parameters. This ratio was measured across prevalent GNN models with various number of layers. Figure 5 presents the findings, with the y-axis depicting $log_2\alpha$ to accommodate the vast range of values. We observe that α varies from 13.4 to 2368.1. Notably, in sophisticated deep GNN architectures, such as DeeperGCN [22] with 112 layers, α reaches an extraordinary 2368.1. This disparity stems from the fact that the number of vertices within a subgraph increases more rapidly than the number of model parameters, a consequence of k-hop sampling where each model layer corresponds to a layer of the subgraph.

Based on these observations, we are inspired to leverage model migration to reduce the amount of data transfers during GNN training. Our goal is to move the model to GPU servers

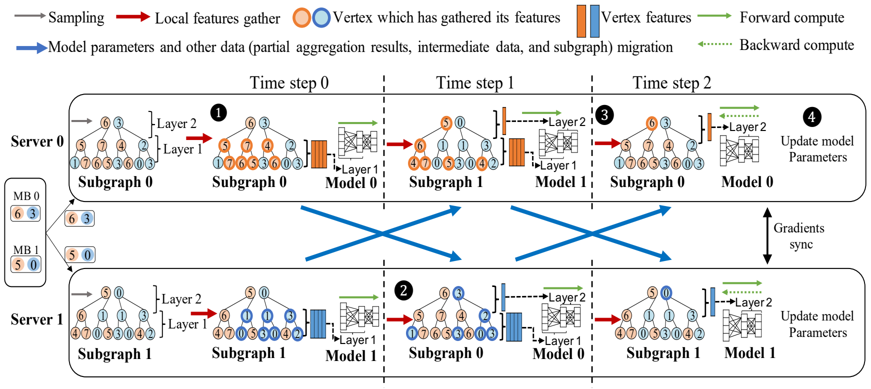


Figure 6: The naive feature-centric approach.

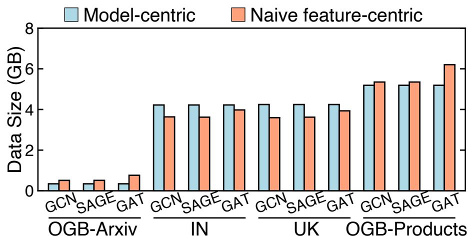


Figure 7: Comparison of the amount of transferred data: model-centric training vs. naive feature-centric training. The y-values are log2 scaled.

where the vertex features are located, rather than fetching features from remote servers. For the sake of clarity, we term this the "*feature-centric*" approach.

3.2 A Naive Feature-Centric Training Approach

A naive feature-centric approach. It involves migrating the model to remote GPU servers when the vertex features needed for a subgraph are not locally available. However, this method necessitates transferring considerable amount of intermediate data along with the model, owing to the computational dependencies intrinsic to GNNs. Specifically, aggregation operations must be finalized after acquiring the vertex features of all fanout neighbors. Additionally, backward propagation is contingent on intermediate data produced during the forward pass, and the computation of subgraphs must proceed sequentially, layer by layer.

Figure 6 shows an example of this method when training two mini-batches in Figure 3. We focus on a single iteration on model 0 for brevity. The process comprises three time steps. At time step 0, model 0 initiates computation on layer 1 of subgraph 0. It gathers the features of vertices 4, 5, 6, and 7 locally and then inputs these into the model's first layer for forward propagation (❶). However, the features for vertices 0, 1, 2, and 3 in layer 1 of subgraph 0 are not locally available, resulting in a partial completion of layer 1's forward computation. We save the partial aggregation results and intermediate data within model 0 for temporary storage. At time step 1, model 0, with the temporarily stored data and the topology of subgraph 0, migrates to server 1. There, it gathers the features of vertices 0, 1, 2, and 3. Consequently, the forward computation for layer 1 is fully executed, and layer 2's forward computation is partially completed (❷). At time step 2, model 0 returns from server 1 to server 0, carrying the stored data and the topology of subgraph 0. It gathers the features of the root vertex, 6, and uses the previously stored intermediate data to complete the forward and backward computations for the entire GNN model (❸). Subsequently, model 0 synchronizes gradients with model 1 and performs parameter updates (❹).

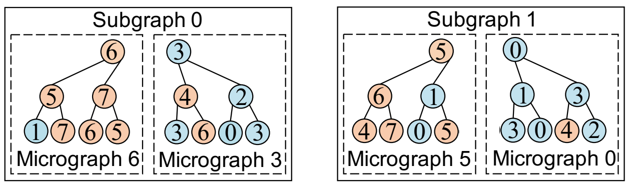
Challenges. While the naive feature-centric approach eliminates the need for remote feature gathering, it may compromise performance due to extensive intermediate data communication and frequent model transfers. Figure 7 compares the total data transmissions of the model-centric and naive feature-centric methods. It shows that, despite being beneficial in certain scenarios, the naive feature-centric approach can demand up to 2.59× the data communication of the model-centric one. This significant communication overhead often results in suboptimal performance for the naive feature-centric strategy.

To fully capitalize on the unique characteristic of GNNs, where model sizes are typically smaller than those of the remote vertex features, a more sophisticated approach is essential. This approach should facilitate model migration while mitigating the high communication overhead stemming from the computational dependencies inherent in GNNs.

4 Locality of Micrograph

A primary source of inefficiency in the naive feature-centric approach lies in its fundamental training unit: the subgraph. There is weak data locality when retrieving features for the subgraph within a distributed environment. To address this, we introduce the concept of a **micrograph**, a more refined data structure. Building on model migration and leveraging enhanced data locality of micrographs, as detailed subsequently, the micrograph significantly diminishes the need for extensive intermediate data communication and frequent model transfers.

Micrograph definition. A micrograph G' is a computation graph derived from a single mini-batch vertex v via k-hop sampling in the original graph G. When the mini-batch size is larger than one, a subgraph will consist of multiple micrographs, as illustrated in Figure 8. For convenience, we use the vertex ID to represent its corresponding micrograph. We assume that (1) the mini-batch size is two, (2) vertex features and graph topology data are distributed across two GPU



(a) (b)

Figure 8: Examples of subgraphs and micrographs.

Sampling	#S	METIS (%)				Heuristic (%)				R_{sub}(%)
		Arxiv		Products		Papers		IT		
		2L	10L	2L	10L	2L	10L	2L	10L	
Node-wise	2	75	73	95	88	93	61	66	64	50
	4	66	45	92	79	89	43	54	46	25
	8	59	27	88	68	84	35	48	36	12
	16	63	35	86	61	84	30	46	32	6
Layer-wise	2	79	54	55	52	85	58	80	53	50
	4	70	30	34	28	77	31	67	30	25
	8	65	18	25	14	56	24	63	18	12
	16	61	12	21	9	57	12	61	12	6

Table 1: Data locality of micrographs with various sampling and graph partition algorithms and model layers.

servers, and (3) the fanout and the number of computation layers are both set to two. Subgraph 0, associated with mini-batch {6, 3}, comprises micrographs 6 and 3. Similarly, subgraph 1, corresponding to mini-batch {5, 0}, includes micrographs 5 and 0.

Data locality in micrographs. Micrographs, due to their finer granularity compared to subgraphs, inherently exhibit superior data locality. Specifically, when employing widely-used graph partitioning algorithms [7, 19, 24–26, 54], there is a high probability that the root vertex of a micrograph and its fanout neighbors reside within the same partition. Consequently, the server hosting the root vertex's features is also likely to hold the features for its neighboring vertices

We use Figure 8 as an example for illustration. For clarification, distinct colors are used to indicate the home locations of vertices; for instance, red represents server 0, and blue represents server 1. In Figure 8(a), we need to retrieve the subgraph for the mini-batch {6, 3}. Consequently, vertices 6 and 3 are designated as the roots of micrographs 6 and 3, respectively. Leveraging data locality, micrograph 6 retrieves vertices 6, 5, and 7 from server 0. This process accesses 75% of the feature vectors needed for training micrograph 6, as three out of four vectors are read from server 0. Similarly, micrograph 3, when trained, accesses 60% of its required feature vectors, with three out of five vectors read from server 1. It's noteworthy that this feature locality is also present in micrograph 5 and 0.

To demonstrate the generality of this observation, we conduct experiments on four real-world open-source graph datasets. The first two datasets are partitioned with METIS [19] used in DGL [36], and the last two large datasets are partitioned with a heuristic algorithm, as utilized in BGL [25] because the METIS algorithm runs out of memory when partitioning these two graphs. We utilize both the node-wise [12] and layer-wise [9] random sampling algorithms. We vary the number of servers (#S) from 2 to 16 and observe the locality of micrographs for both shallow-layer GNNs (i.e., two layers, denoted as '2L') and deep-layer GNNs (i.e., ten layers, denoted as '10L').

We collect the number ($N_{colocated}$) of non-root vertices which are co-located with its root vertex in a micrograph during GNN training. Then, we compute the ratio (R_{micro}) of $N_{colocated}$ and N_{total}, where N_{total} is the total number of vertices in a micrograph. We show R_{mico} in Table 1. Likewise, we calculate the locality (R_{sub}) of subgraphs by dividing the count of non-root vertices co-located with a specified root vertex by the total number of vertices in the subgraph. For ease of presentation, we only show the mean value of R_{sub}. We observe that R_{micro} is consistently larger than R_{sub}, meaning micrographs' better locality. Furthermore, as the number of GPU servers increases from 2 to 16, the difference between R_{micro} and R_{sub} is increased from 1.59× to 10.60×.

We credit this enhanced data locality to the prevalent graph partitioning strategies employed in GNNs, including algorithms like METIS [7, 19, 26] and GNN-specific heuristics [24, 25, 54]. These methods prioritize minimizing cross-machine feature transmission by strategically assigning neighboring vertices to the same partition, thus promoting data locality. As a result, when performing k-hop neighbor sampling from a root vertex, the majority of vertices within the same micrograph are likely to be co-located with the root on the same GPU server.

We clarify that the enhanced locality of the micrograph alone does not mitigate remote feature fetching in model-centric systems; it is essential to combine it with LeapGNN's model migration strategy to address this issue as shown in §5.1.

5 Design of LeapGNN

In this section, we detail the key ideas of LeapGNN that tackles the aforementioned challenges by leveraging data locality in micrographs. Subsequently, we present two enhancements designed to augment the performance of LeapGNN.

5.1 Micrograph-Based GNN Training

Key idea. We propose a novel approach, micrograph-based GNN training, which decomposes a subgraph into multiple micrographs and executes the complete forward and backward computations for each micrograph on a single GPU server. Since a model may be assigned multiple micrographs for training, with their root vertices distributed across multiple servers, LeapGNN first migrates the model to the target server where the root vertex feature of the current micrograph is located before training. Then, child vertex features of the micrograph not available locally are fetched from remote servers. This does not incur significant feature transfer overhead, as the

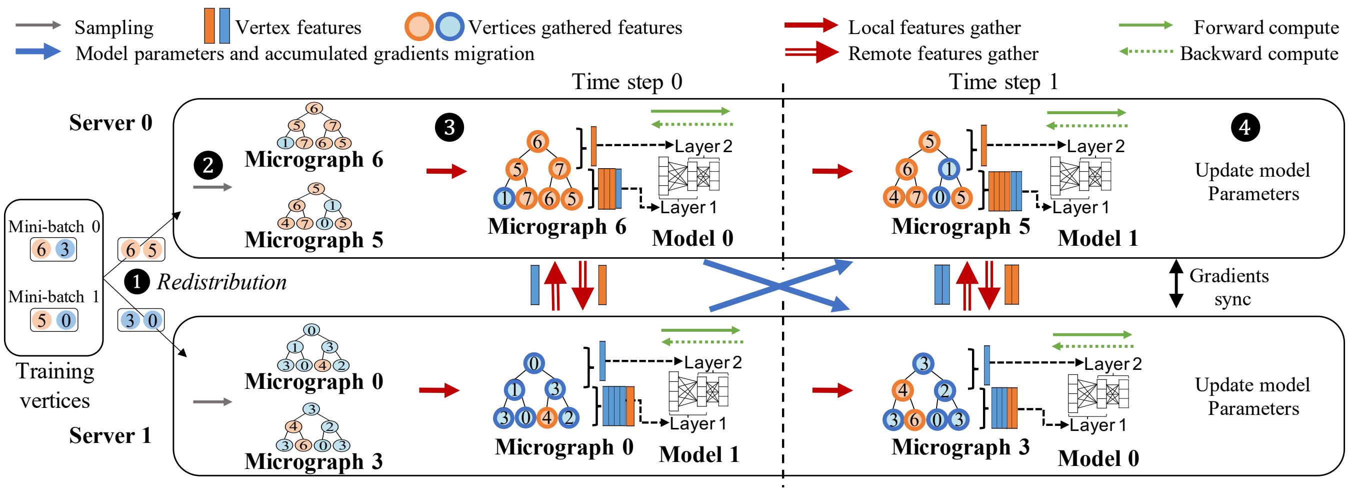


Figure 9: Example of training two mini-batches in one iteration in LeapGNN.

root vertex's feature is already on the local server, meaning that most of the current micrograph's child vertices are also likely to reside on the local server due to the micrograph's data locality. This micrograph-based GNN training offers two significant advantages: (1) It minimizes remote feature gathering by exploiting data locality within micrographs. (2) It eliminates intermediate data retrieval. Since micrographs are processed on a single server, both forward and backward propagation can be completed once vertex features are gathered in each layer of the micrographs.

The procedure of micrograph-based GNN training. Assume that there are N GPU servers. Before training starts at each iteration, each server s is assigned a model d where $s \in [0, N-1]$ and d is equal to s. Each model is randomly assigned a mini-batch of training vertices. We assign a home server for each vertex in the mini-batch based on where the features for the vertex is located. When training begins, micrograph-based GNN training consists of the following four steps. **(1) Redistribution of root vertices**. We will group root vertices in all mini-batches base on their home server IDs for task redistribution. Then, each vertex group is assigned to the worker running on the corresponding home server. Since root vertices are randomly sampled from the global graph, the number of vertices received by each server is approximately equal at most cases. For instance, when we use four GPU servers for training, the load difference among them is less than 10% for 97.3% training iterations for the four datasets used in our experiments. **(2) Generating micrographs**. After the redistribution, each server i needs to use k-hop sampling to generate a single set mg_d^i of micrographs for each vertex group, where d is determined by looking up the GNN model which is originally selected to be trained using the vertex. **(3) Training in N time steps**. At time step t, model-d migrates to server $s_{new} = (d+t)\%N$ and trains using micrographs $mg_d^{s_{new}}$. If the vertex features of the current micrograph are not available on the local server, they will be fetched from the remote ones. To ensure model parameters are updated only after the whole subgraph training is completed, the temporary gradients obtained by training one micrograph is accumulated. **(4) Updating model parameters.** When the training on the last micrograph of the subgraph is completed, the accumulated gradients are synchronized among all GPU servers. Finally, the model parameters are updated to finish training for one iteration.

Figure 9 shows an example of our micrograph-based GNN training for two mini-batches in one iteration on two GPU servers. Initially, both server 0 and 1 duplicate the DNN model. Eight features are evenly distributed between these servers. Then, training vertices are reassigned, with server 0 obtaining vertices 6 and 5, and server 1 getting vertices 0 and 3 (❶). Next, each server independently generates micrographs through sampling (❷). The training process is then divided into two time steps (❸). During the first step, servers 0 and 1 train using micrographs 6 and 0, respectively. Upon completing the backward computation, intermediate data is discarded, retaining only the accumulated gradients. Models are then migrated with their gradients: model 0 moves to server 1, and model 1 to server 0. In the second time step, the servers continue training using micrographs 5 and 3, respectively. Finally, the gradients from both models are averaged, and parameter updates are conducted (❹). This micrograph-based approach, as opposed to the subgraph-based training, reduces the transmission of vertex features between servers through strategic model migration (8 features in Figure 3 vs. 6 in Figure 9).

Limitations of the locality-optimized approach. One might think that training models on redistributed micrographs without model migration—such as model 0 on micrographs 6 and 5, and model 1 on micrographs 0 and 3—could enhance feature locality. While it is true, this approach could inadvertently disrupt the training sequence for each model, as the sequence would be randomized only within a local context rather than globally. For instance, model 0 would never be exposed to micrograph 3, as its features reside on a separate server 1. This locality-optimized approach could introduce bias into the mini-batch training data, potentially degrading the model's accuracy, as discussed in [30, 42]. Conversely, the micrograph-based training method preserves model accuracy by maintaining the globally randomized data sequence. For instance, model 0 consistently trains on micrographs 6 and 3, matching the composition of the original mini-batch 0. Additionally, the use of gradient accumulation, as shown in prior research [17, 46, 51], does not compromise training accuracy. The impacts of the locality-optimized approach on model accuracy are discussed in §7.9.

A special case arises when the number of micrographs obtained from a subgraph is less than the number of GPU servers in a cluster. This implies that on certain machines there are no corresponding micrographs to train. In such cases, we allow the model to do nothing on those machines until other models have completed the corresponding micrographs' training.

5.2 Vertex Feature Pre-Gathering

Although micrograph-based GNN training significantly diminishes the need for remote feature retrieval by leveraging data locality within micrographs, it can inadvertently lead to redundant transmissions of vertex features across consecutive time steps, potentially resulting in suboptimal performance. We use the example in Figure 9 to illustrate this. Utilizing micrograph-based GNN training, the worker on server 0 at time step 0 must obtain the feature of vertex 1 from server 1. Upon completing the computations for that time step, the vertex feature memory is cleared to prevent GPU memory overflow. However, at time step 1, the worker on server 0 is required to retrieve the features for both vertices 1 and 0 from server 1. Consequently, for processing just two micrographs, server 0 ends up fetching a total of three features from server 1, including a redundant transmission for the feature of vertex 1.

Key idea. We introduce vertex feature pre-gathering to mitigate redundant transmissions. This approach capitalizes on the predictability of which vertices from the micrographs will undergo training on a given server, regardless of the specific models involved (e.g., model 0 or 1). For instance, referring to Figure 9, we can anticipate that at time step 0, vertex 1 will be utilized by micrograph 6, and at time step 1, both vertices 1 and 0 will be utilized by micrograph 5. Pre-gathering allows us to fetch the features for vertex 1 and 0 from server 1 to server 0 in a single batch, thereby reducing the communication cost from three feature transmissions without pre-gathering to just two.

Space overhead. While pre-gathering additional non-local features could further minimize redundant transmissions, it necessitates extra memory space. To manage this, we limit pre-gathering to the features of vertices required for a single iteration of GNN training. As detailed in §4, due to the feature locality inherent in micrographs, this pre-gathering strategy ensures that the memory footprint remains within the bound of that required by model-centric GNN training, as depicted in Figure 3. For example, when training GAT on the OGB-Products dataset, the model-centric approach requires 530 MB of host memory for temporary feature storage, whereas pre-gathering demands only 87 MB.

5.3 Micrograph Merging in GNN Training

Micrograph-based GNN training tends to necessitate more frequent GPU kernel launches and may also entail synchronization overhead at the end of each time step. Consequently, a trade-off is required between the advantages of reduced remote feature fetching through model migration and the additional overhead imposed by micrograph-based training.

Key idea. By merging micrographs, we can potentially decrease the number of time steps during the GNN model training, thereby reducing the associated training overhead. However, the merging process must be approached strategically; random micrograph consolidation could lead to load imbalance across GPU servers. When considering the merging of micrographs, two critical questions arise.

Which micrographs to merge? Merging micrographs could lead to an increase in remote feature fetching. To counteract this, we should strategically select micrographs that rely on the fewest number of remote feature vectors. Additionally, it is imperative to ensure that different models are concurrently trained on separate servers after merging. Therefore, all micrographs utilized within a single time step should be considered for merging. Specifically, for each time step, we calculate the total count of vertex features, denoted as Num_{vertex}, for all micrographs slated for training. We then pinpoint the time step ts_{min} with the lowest Num_{vertex} value. However, since we must make decisions before the execution of an iteration and before micrographs are generated, Num_{vertex} is not yet determinable. To circumvent this, we approximate Num_{vertex} using the total number of root vertices, designated for training in a given time step. Subsequently, we merge the micrographs scheduled for ts_{min} with those from other time steps, ensuring they are used as evenly as possible by the same model. By doing so, we can balance the time the model takes across different time steps.

How many micrographs should be merged? If we merge all the micrographs, micrograph-based training degrades to subgraph-based training. If we did not merge enough micrographs, the training overhead can be still significant. Therefore, we require an examination period to determine how many micrographs should be merged. During this period starting from the second epoch, for each iteration, we identify a time step ts_{min} and merge the micrographs in the time step with those in other time steps but for the same model. Then, we measure the execution time of the current epoch and compare it to that of the previous epoch. If the execution time is not reduced by merging, we stop the process and use the

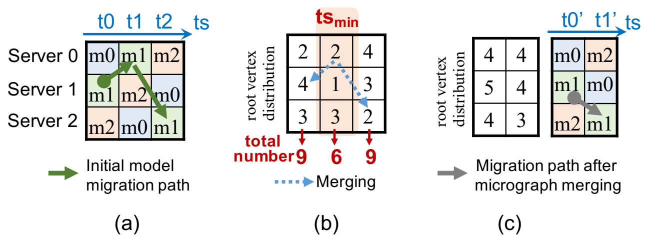


Figure 10: Illustration of micrograph merging. Squares of the same color indicate the same GNN model. Model migration paths of m2 and m3 are omitted. The total number of root vertices of each model keeps consistent before and after merging.

existing micrographs for training. Otherwise, we repeat the identifying-and-merging process until the execution time cannot be reduced. After that, all the following epochs will use the same merging pattern.

We use an example in Figure 10 to illustrate the micrograph merging. We assume that there are three GPU servers including server 0, 1, and 2. Hence, the initial training needs three time steps t0, t1, and t2. We use a matrix to show the assignment of models (i.e., m0, m1, and m2) across the three time steps. The initial model distributions and migration paths are shown in Figure 10(a). For merging, we count the total number of redistributed root vertices for each model at each time step and show them in Figure 10(b). Then, t1 is identified as ts_{min}. Consequently, we can merge the micrographs from time step t1 with those from time steps t0 and t2. Taking model m1 for instance, its two root vertices assigned at time step t1 are evenly distributed across model m1 at time step t0 and t2, resulting in model m1 having 5 vertices at t0 and 3 at t2. Models m2 and m3 follow a similar redistribution process. After merging, we can remove time step t1. The revised training process consists of only two time steps t0' and t1'. The root vertex distribution and model migration paths after merging are shown in Figure 10(c).

6 Implementation

We implemented LeapGNN based on one of the most popular GNN frameworks, DGL [36] (with PyTorch backend). We reutilized DGL's graph data partitioning module, sampling module. The GNN computation module includes both forward and backward propagation, gradient synchronization, and parameter updates. Our primary focus is on the gathering phase, which has been identified as the bottleneck of distributed GNN training.

Before implementing the micrograph-based training, we first developed a distributed cache using Golang to store the partitioned graph data on each machine. The Python-based GNN application utilizes Google Remote Procedure Call (gRPC) to request and retrieve vertex features from other machines. The model migration is implemented using PyTorch's distributed module. For pre-gathering, we utilized a Python list to temporarily store multiple micrographs and detected

Dataset	#Vertex	#Edge	Dim.	Vol_G	Vol_F
Arxiv	169K	1.17M	128	3.3 MB	85 MB
Products	2.45M	61.9M	100	464 MB	980 MB
UK	1M	41.2M	600	12 MB	2.3 GB
IN	1.38M	16.9M	600	8.2 MB	3.2 GB
IT	41.3M	1.15B	600	363MB	92.3 GB

Table 2: The details of graph datasets used in GNN training. #Vertex and #Edge denote the number of graph vertices and edges. Dim. denotes the dimension of vertex features. Vol_G and Vol_F denote the data volume sizes of the graph topology and features.

and removed duplicate vertices before requesting features from the cache server. To implement micrograph merging, we monitored the runtime for each epoch's training and stored its value in a temporary list. After that, we utilized this information to adjust the number of time steps of one iteration as described in §5.3.

7 Evaluation

7.1 Experimental Setup

System configurations. We conduct the experiments on a cluster with four GPU servers, each with 2×Intel(R) Xeon(R) Gold 5318Y CPUs (48 cores), 128 GB CPU memory, and an NVIDIA A100 40GB GPU. All servers are interconnected with a 10 Gb/s Ethernet network, running Ubuntu v18.04, PyTorch v1.10.1+cu113, and Python v3.9.0.

Models and datasets. We use three shallow models (GCN [20], GraphSAGE [12], GAT [8]) and two deep models (DeepGCN [21] and GNN-FiLM [6]) to evaluate LeapGNN. Following the paper [6], we set DeepGCN to include seven layers and GNN-FiLM to comprise ten layers. Other models all have three layers. Beyond the variation in the number of layers, these models are distinguished by their distinct methods for aggregating neighboring vertices. We use 'Model(16)' and 'Model(128)' to denote the neural network with hidden dimension sizes of 16 and 128 respectively.

We carefully choose five well-established datasets, detailed in Table 2, to ensure that both the raw graph data and intermediate data (e.g., the buffered features and temporary data for sampling) can fit in the server memory. The Arxiv [15] and Products [15] datasets represent smaller graph instances, while the UK [4] and IN [4] datasets are indicative of medium-scale graphs. In contrast, the IT [5] dataset exemplifies a large-scale graph [27]. It is important to note that the original UK, IN, and IT datasets lack vertex features; therefore, we introduce random features for these datasets, assigning a dimension of 600 to each vertex, a method akin to those in [11, 24]. Across all datasets, we implement a standard neighbor sampling fanout of 10, aligning with the setup in [44]. Owing to the protracted training durations associated with certain evaluations on the large IT dataset, we limit our analysis to a select

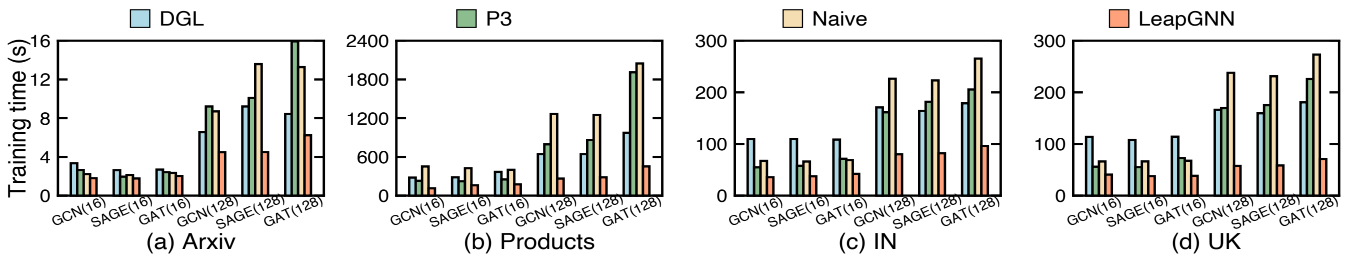


Figure 11: Performance comparison of various training frameworks for three shallow models.

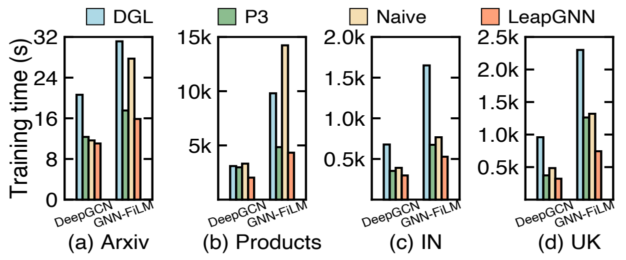


Figure 12: Performance comparison results on deep models.

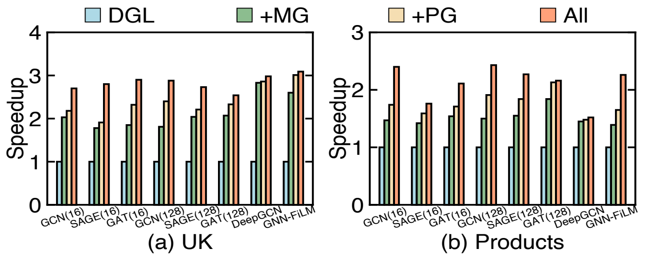


Figure 13: Improvements of each individual technique. The training time of DGL is normalized to one.

subset of tests, with outcomes presented in §7.5. We use the default METIS partitioning algorithm in the DGL framework, and the maximum variance in the number of vertices across partitions is < 1% for partition load balance.

Compared systems. We evaluate LeapGNN against the industry-leading DGL framework [36] and the state-of-the-art P^3 [11] and NeutronStar [37] frameworks. DGL facilitates GNN model training by fetching required features, either locally or remotely. P^3 integrates model-parallel and data-parallel approaches, minimizing the transfer of original vertex features but necessitating additional intermediate data movement. NeutronStar enhances training efficiency by optimizing the balance between redundant computation and communication time. Unlike these frameworks, which adopt a "model-centric" approach, LeapGNN is "feature-centric". Note that, we categorize P^3 as model-centric in our paper, even though it avoids moving vertices' input features. This is because it introduces the transmission of vertices' hidden features (i.e., activations), while LeapGNN does not. This is also why LeapGNN outperforms P^3 when the hidden dimension is large (§7.2). As P^3 is not open-source, we reimplemented it based on the original paper's description. We also assess a naive feature-centric approach (Naive) in §3.2 to underscore the value of LeapGNN's techniques. Direct comparison with ROC [18] is omitted, as P^3 has already surpassed it [11]. Most experiments use mini-batch training, except for NeutronStar, which only supports full-batch training. We compare LeapGNN with NeutronStar in §7.7.

7.2 Overall Performance

Figures 11 and 12 respectively show the end-to-end training times for shallow and deep GNN models, due to their significant numerical differences. We train each model for ten epochs and report the average training time. We have four observations.

First, LeapGNN outperforms other frameworks by minimizing feature and intermediate data transmissions, offering 1.3–3.1× speedups over DGL and 1.2–4.2× over P^3 across various GNN models, and up to 4.8× over Naive. Second, while Naive can be efficient, it does not consistently improve on DGL and P^3. In constrast, LeapGNN consistently outperforms existing frameworks, highlighting the necessity of LeapGNN's techniques. Third, LeapGNN's speedup varies for different GNN models. It achieves 2.5× acceleration for GCN whereas 2.2× for GAT on Products. This variation mainly arises from the varying time proportion of feature gathering in the training, resulting in different potential for performance improvements. Fourth, LeapGNN's improvement is independent of the hidden dimension of the models, while P^3's speedup is sensitive to this [11, 25]. For example, with GAT on the IN dataset, when the hidden dimension size is 128, P^3 is 1.2× slower than DGL, while LeapGNN still achieves 1.8× performance improvement. This is because LeapGNN performs the forward and backward propagation of a micrograph on a single server, eliminating the inter-server transmission of hidden embeddings of P^3.

7.3 Impact of Individual Techniques

Figure 13 illustrates the impact of each optimization. Using DGL as the baseline, *+MG* enables micrograph-based GNN training, *+PG* adds pre-gathering on *+MG*, and *All* further includes micrograph merging. We observe that each technique enhances performance, with *All* achieving the highest (2.14×

		Miss Rate
Arxiv	DGL	74%
	+MG	43%
Products	DGL	77%
	+MG	22%
UK	DGL	78%
	+MG	19%
IN	DGL	77%
	+MG	9.2%

Figure 14: Miss rates.

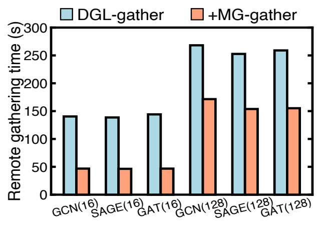
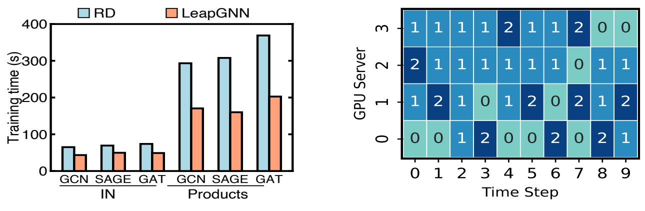


Figure 15: Details after using micrograph-based training.



(a) Training time. (b) Workload distribution.

Figure 18: The impact of selection schemes.

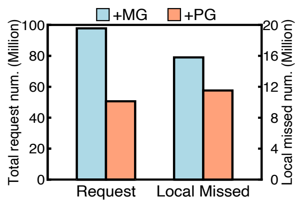


Figure 16: Details after using pre-gathering.

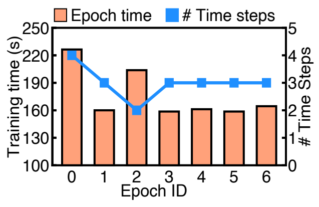


Figure 17: Details after using micrograph merging.

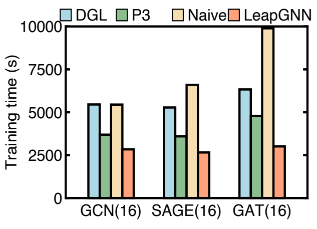


Figure 19: Performance on the IT graph dataset.

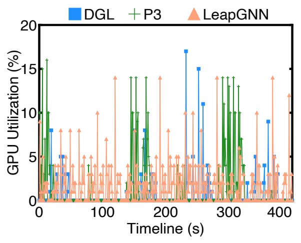


Figure 20: GPU utilization with different systems.

on Products, 2.72× on UK). Besides, the most impactful technique varies across scenarios. For example, in GCN(16), the third technique contributes the most (i.e., 47%) on Products, while the first technique is most effective (i.e., 61%) on UK.

Figure 14 shows that micrograph-based GNN training reduces the local feature missing rate by an average of 53%, cutting the average remote feature gathering time by 2.3× for Products (Figure 15). Figure 16 displays the number of remote feature requests and local feature miss requests with pre-gathering enabled. It shows pre-gathering further reduces the former by 1.9× and the latter by 1.4×. Figure 17 shows the epoch time and time steps per iteration for the GAT model on the Products dataset using micrograph merging. Starting with four machines and four time steps in epoch 0, LeapGNN dynamically reduces the time steps to three in epoch 1 and two by epoch 2. It then settles on three time steps per iteration for the rest of training, achieving optimal training efficiency.

7.4 Micrograph Merging Selection

To demonstrate the effectiveness of our selection method (§5.3) in the micrograph merging, we compare it with a random scheme (RD), where a micrograph is randomly selected to merge with other micrographs for each model. Figure 18(a) illustrates that our method outperforms RD by 1.4–1.9× on IN and Products. Figure 18(b) shows the number of GNN training models on each GPU server at each time step with RD. It shows that RD has an uneven workload distribution among servers, thereby degrading the training performance.

7.5 Results on Large-Scale Graph

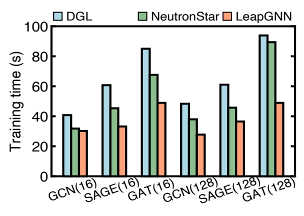
Figure 19 illustrates the epoch training time of different systems on the large dataset IT. Due to the extensive training times, we only conduct a subset of the tests. LeapGNN achieves an average acceleration of 1.91× and 1.48× against DGL and P^3, respectively. The improvement is attributed to the increase in the local feature hit rate from 24.4% to 92.3% after employing the techniques in LeapGNN. This result shows LeapGNN is still effective on the large dataset.

7.6 GPU Utilization

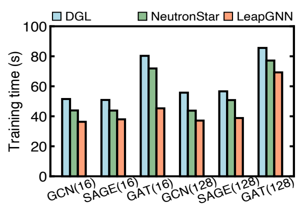
Figure 20 illustrates the GPU utilization of LeapGNN, DGL, and P^3 for the GAT model on the UK dataset. Similar results are observed on other models and datasets. We utilize the Python library GPUtil [1] (which relies on nvidia-smi) to capture the GPU utilizations every 250ms during a steady 400-second running time window. We observe that the peak GPU utilization is smaller than 20% in all these systems due to the sparse nature of computations [11] and the high speed of A100. However, LeapGNN is able to keep GPU busy (i.e., at least one core active) for 52% of the total time, while DGL and P^3 only achieve 13% and 18%, respectively. This explains why LeapGNN achieves the shortest training time.

7.7 Comparison with NeutronStar

Since NeutronStar does not support sampling, we disable sampling in all compared systems. For fair comparison, we reproduce NeutronStar based on the DGL framework. Figure 21 shows that both NeutronStar and LeapGNN outperform DGL, with LeapGNN being the best. LeapGNN is 1.05–1.82× faster than NeutronStar. This is because, despite NeutronStar's acceleration of DGL by reducing redundant computations, the proportion of feature communication is larger in our test scenario. LeapGNN reduces this communication through model migration, leading to a shorter overall training time.

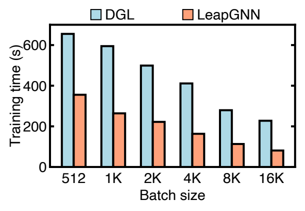


(a) IN dataset.

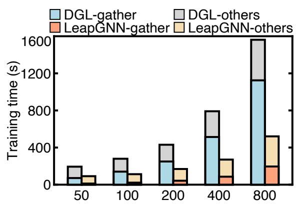


(b) UK dataset.

Figure 21: Performance comparison with full-batch training.



(a) Various batch sizes.



(b) Various dimensions of features.

Figure 22: Performance comparison with various batch sizes and feature dimensions.

7.8 Sensitivity Analysis

Batch size. Figure 22(a) depicts the training time of GCN on Products with various batch sizes. LeapGNN consistently outperforms DGL for batch sizes from 512 to 16K, with performance improvements of 2.2–2.8×. This can be mainly attributed to LeapGNN's ability to reduce remote feature fetching time.

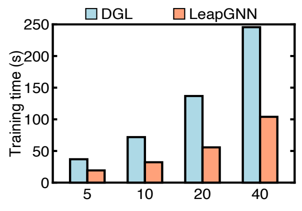
Feature dimension. Figure 22(b) shows the system performance on Products with different feature dimensions. As the feature dimension increases, LeapGNN's speedup is increased from 2.1× to 2.9×. This is because the proportion of remote feature gathering time in DGL rises from 36.8% to 72.0%, leaving more space for acceleration with LeapGNN.

Fanout size. Figure 23(a) shows the system performance of LeapGNN with different fanouts. LeapGNN consistently outperforms DGL by 2.3× on average. Furthermore, it offers better scalability than DGL on high-dimensional large graph datasets. Specifically, when the feature dimension expands by a factor of 8 (from 5 to 40), LeapGNN's training time is increased by 5.3× while DGL's time is increased by 6.6×.

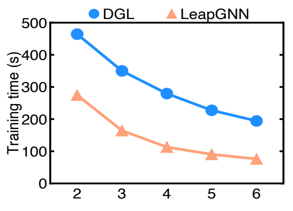
Number of distributed machines. Figure 23(b) presents the system performance of GCN on Products with various number of machines. We observe that LeapGNN consistently outperforms DGL by 2.27× on average. Furthermore, as the number of machines increases from 2 to 6, LeapGNN's speedup is increased from 1.69× to 2.55×. This shows LeapGNN has better scalability than DGL in multi-machine scenarios.

7.9 Model Accuracy

So far we have compared LeapGNN with the state-of-the-art systems with accuracy fidelity. Approximate methods, such



(a) Various fanouts.



(b) Various number of machines.

Figure 23: Performance comparison with various fanouts and number of machines.

Dat-aset	Models	DGL Acc.	LO		LeapGNN	
			Acc.	Drop	Acc.	Drop
Arxiv	GCN	60.24	59.71	**0.53**	60.24	S
	SAGE	61.96	61.96	S	61.98	S
	GAT	59.28	59.15	**0.13**	59.25	S
Products	GCN	83.16	83.17	S	83.20	S
	SAGE	82.80	82.77	S	82.80	S
	GAT	83.95	83.96	S	83.92	S

Table 3: Model accuracy (%). The column in bold indicates accuracy drop. "S" stands for "same", indicating that the accuracy drop is within 0.1%.

as selectively ignoring remote vertex features [28], proximity-aware ordering [25], and locality-optimized method (LO) [24, 55], have also been proposed to reduce remote feature gathering time. However, these systems may compromise model accuracy. For example, [28] has demonstrated a 0.95% accuracy drop for SAGE on Products and [25] has shown a 0.2% accuracy drop on the OGB-Papers dataset [15]. Since the impact of LO on GNN accuracy has not been studied, we conduct tests to study this on the Arxiv and Product datasets. We do not test accuracy for the other three datasets because their vertex features are randomly generated and meaningless for accuracy testing. Table 3 shows that LeapGNN maintains the same accuracy as DGL on both datasets, while LO fails to maintain accuracy on the Arxiv dataset. This is because LO only chooses the vertices from the local node, introducing bias into the training sequence, potentially degrading the model's accuracy, as discussed in [30, 42]. Given that 0.1% accuracy loss could lead to substantial economic consequences [2, 10, 23, 53] and our work is able to maintain accuracy fidelity, we do not compare LeapGNN's training time with systems that may compromise accuracy [24, 25, 28, 55].

8 Discussion

Graph partitioning time. While the METIS graph partitioning algorithm used by LeapGNN is more time-consuming than P^3's random partitioning method, it runs offline and only once. Thus, its partitioning time can be amortized over the large number of training epochs and GNN tasks. For example, when partitioning a large IT graph, although LeapGNN takes approximately 2800 seconds (roughly the time of one training epoch) longer than random partitioning, it still outperforms P^3 by 1.6× on GAT in a typical 200-epoch training scenario, with partitioning time included. Moreover, several recent GNN-tailored partitioning algorithms [24, 25, 28, 54] have shown potential to further reduce partitioning durations while maintaining high locality.

Time and space overhead. LeapGNN incurs additional communication time due to the migration of models and gradients across servers compared to DGL. However, the added time overhead is negligible, averaging only 4.6% of the total training time. As shown in §7.2, its training time (including the communication) still outperforms other approaches. Regarding memory usage, despite receiving multiple partial gradients per model during each iteration, LeapGNN maintains memory efficiency equivalent to DGL. This is because LeapGNN accumulates incoming partial gradients with the existing ones and updates them in place.

Failure recovery. In LeapGNN, models migrate between machines at each time step but revert to their origin machines at the end of each iteration. Therefore, we perform the checkpointing at the end of each iteration, not after every time step, aligning with standard checkpointing practices. Additionally, since only a single model resides on each server at the iteration's end, each server only needs to handle checkpointing for one model, mirroring typical GNN system designs.

Generality of LeapGNN. Since LeapGNN does not modify the GNN computation kernel functions and performs forward and backward computations for each micrograph on a single server, it maintains robust compatibility with GNN models that use different aggregation operations (e.g., sum, max, LSTM) as supported by the original DGL framework. The efficacy of LeapGNN stems from the locality of the micrograph, rendering it unsuitable for random graph partitioning algorithms [11]. However, most of GNN partitioning algorithms provide strong locality [7, 19, 24–26, 54], endowing LeapGNN with excellent practicality and applicability.

9 Related Work

Graph partitioning optimizations. DGL [36] utilizes the METIS graph partitioning algorithm to minimize the number of cut edges. ByteGNN [54] and BGL [25] considers multiple-hop neighbors to further reduce cross-machine vertex accesses. ROC [18] proposes an online linear regression model to optimize graph partitioning. These efforts aim to maximize the co-location of adjacent vertices on the same machine, thereby reducing inter-machine feature transmission. They are orthogonal to our work and could potentially enhance LeapGNN's performance.

Sampling algorithm optimizations. These works focus on improving the locality of feature accesses by changing sampling algorithms [24, 25, 28, 55]. As mentioned before, these works tend to compromise the randomness of GNN sampling, thereby impacting GNN training accuracy. In contrast, LeapGNN has the model accuracy fidelity.

Cache optimizations. These studies aim to design GPU memory caches to reduce the feature fetching time from CPU memory. PaGraph [24] and GNNLab [44] implement static caches to store features of vertices with the highest degree or access frequency. BGL [25], on the other hand, employs a dynamic FIFO cache to balance cache management overhead with hit rate efficiency. Legion [34] contributes a unified multi-GPU cache strategy to reduce topology and feature transmissions over PCIe. These methods leverage additional GPU memory and are complementary to our research. We posit that integrating these techniques into LeapGNN could significantly enhance training performance.

Computation optimizations. Considerable research efforts [16, 26, 39, 40] accelerate GNN computation via fine-grained pipeline and balanced workload scheduling across multiple GPU cores. They are designed for small graphs that can fit entirely into GPU memory. In contrast, LeapGNN focuses on large graphs with distributed GNN training.

Others system optimizations. DGCL [7] enhances GPU-to-GPU communication via a multi-path selection algorithm. Betty [46] tackles large batch training on single GPUs using batch splitting. Dorylus [35] optimizes communication for serverless training scenarios with Lambda servers. These methods are orthogonal and complementary to LeapGNN.

10 Conclusion

In this paper, we present LeapGNN, a feature-centric distributed GNN training framework to reduce inter-machine communication overhead. LeapGNN moves the GNN model towards the training features, rather than moving the features to the models as in the existing model-centric frameworks. To tackle the challenges of implementing this approach, we propose the micrograph-based GNN training, vertex feature pre-gathering, and micrograph merging, to reduce remote feature fetching, intermediate data, and synchronization overhead over network. Our experimental results demonstrate that LeapGNN can achieve a speedup of up to 4.2× compared to the state-of-the-art framework, P^3, across a variety of GNN models and datasets.

Acknowledgments

We sincerely thank the anonymous reviewers for their constructive suggestions. This work was supported in part by the National Key Research and Development Program of China (2023YFB4502100), the National Science Foundation of China (62172361), the Major Projects of Zhejiang Province (LD24F020012), the Open Project Program of Wuhan National Laboratory for Optoelectronics (2023WNLOKF005), and the Pioneer and Leading Goose R&D Program of Zhejiang Province (2024SSYS0002).

References

[1] Gputil, 2023. https://github.com/anderskm/gputil.

[2] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. Understanding Training Efficiency of Deep Learning Recommendation Models at Scale. In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, pages 802–814. IEEE, 2021.

[3] Beatrice Bevilacqua, Yangze Zhou, and Bruno Ribeiro. Size-Invariant Graph Representations for Graph Classification Extrapolations. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 837–851. PMLR, 2021.

[4] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered Label Propagation: A Multiresolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web (WWW)*, pages 587–596, 2011.

[5] Paolo Boldi and Sebastiano Vigna. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th international conference on World Wide Web (WWW)*, pages 595–602, 2004.

[6] Marc Brockschmidt. Gnn-Film: Graph Neural Networks With Feature-Wise Linear Modulation. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 1144–1152, 2020.

[7] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. DGCL: An Efficient Communication Library for Distributed GNN Training. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*, pages 130–144, 2021.

[8] Petar Veličković Guillem Cucurull Arantxa Casanova, Adriana Romero Pietro Lio, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations (ICLR)*, 2018.

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

[10] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-N-Run: A Checkpointing System for Training Deep Learning Recommendation Models. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 929–943, 2022.

[11] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed Deep Graph Learning at Scale. In *Proceedings of 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 551–568, 2021.

[12] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. *In Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017.

[13] Mengyue Hang, Jennifer Neville, and Bruno Ribeiro. A Collective Learning Framework to Boost GNN Expressiveness for Node Classification. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 4040–4050. PMLR, 2021.

[14] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[15] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[16] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. Understanding and Bridging the Gaps in Current GNN Performance Optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 119–132, 2021.

[17] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. *In Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 32, 2019.

[18] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with ROC. *In Proceedings of Machine Learning and Systems (MLSys)*, 2:187–198, 2020.

[19] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on scientific Computing (SISC)*, 20(1):359–392, 1998.

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

[21] Guohao Li, Matthias Muller, Ali Thabet, and Bernard Ghanem. DeepGCNs: Can GCNs Go as Deep as CNNs? In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 9267–9276, 2019.

[22] Guohao Li, Chenxin Xiong, Ali Thabet, and Bernard Ghanem. Deepergcn: All You Need to Train Deeper GCNs. *arXiv preprint arXiv:2006.07739*, 2020.

[23] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, et al. Persia: An Open, Hybrid System Scaling Deep Learning-Based Recommenders up to 100 Trillion Parameters. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 3288–3298, 2022.

[24] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling GNN Training on Large Graphs via Computation-Aware Caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*, pages 401–415, 2020.

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

[26] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *Proceedings of USENIX Annual Technical Conference (ATC)*, pages 443–458, 2019.

[27] Takanori Maehara, Takuya Akiba, Yoichi Iwata, and Ken-ichi Kawarabayashi. Computing Personalized PageRank Quickly by Exploiting Graph Structures. *Proc. VLDB Endow.*, 7(12):1023–1034, 2014.

[28] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K Ahmed, and Sasikanth Avancha. DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–14, 2021.

[29] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, pages 203–216, 2021.

[30] Truong Thao Nguyen, François Trahay, Jens Domke, Aleksandr Drozd, Emil Vatai, Jianwei Liao, Mohamed Wahib, and Balazs Gerofi. Why Globally Re-Shuffle? Revisiting Data Shuffling in Large Scale Deep Learning. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1085–1096. IEEE, 2022.

[31] PyTorch. Pytorch profiler, 2023. https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html.

[32] Shihui Song and Peng Jiang. Rethinking Graph Data Placement for Graph Neural Network Training on Multiple GPUs. In *Proceedings of the 36th ACM International Conference on Supercomputing (ICS)*, pages 1–10, 2022.

[33] Yongduo Sui, Xiang Wang, Jiancan Wu, Min Lin, Xiangnan He, and Tat-Seng Chua. Causal Attention for Interpretable and Generalizable Graph Classification. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 1696–1705, 2022.

[34] Jie Sun, Li Su, Zuocheng Shi, Wenting Shen, Zeke Wang, Lei Wang, Jie Zhang, Yong Li, Wenyuan Yu, Jingren Zhou, and Fei Wu. Legion: Automatically Pushing the Envelope of Multi-GPU System for Billion-Scale GNN Training. In *Proceedings of USENIX Annual Technical Conference (ATC)*, pages 165–179, 2023.

[35] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 495–514, 2021.

[36] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315*, 2019.

[37] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. Neutronstar: Distributed GNN Training With Hybrid Dependency Management. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1301–1315, 2022.

[38] Qianwen Wang, Kexin Huang, Payal Chandak, Marinka Zitnik, and Nils Gehlenborg. Extending the Nested Model for User-Centric XAI: A Design Study on GNN-Based Drug Repurposing. *Transactions on Visualization and Computer Graphics (TVCG)*, 29(1):1266–1276, 2022.

[39] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *Proceedings of the 15th USENIX symposium on operating systems design and implementation (OSDI)*, pages 515–531, 2021.

[40] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. MGG: Accelerating Graph Neural Networks with Fine-Grained Intra-Kernel Communication-Computation Pipelining on Multi-GPU Platforms. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 779–795, 2023.

[41] Shu Wu, Yuyuan Tang, Yanqiao Zhu, Liang Wang, Xing Xie, and Tieniu Tan. Session-Based Recommendation with Graph Neural Networks. In *Proceedings of the AAAI conference on artificial intelligence (AAAI)*, volume 33, pages 346–353, 2019.

[42] Chih-Chieh Yang and Guojing Cong. Accelerating Data Loading in Deep Neural Network Training. In *Proceedings of IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 235–245, 2019.

[43] Hongxia Yang. AliGraph: A Comprehensive Graph Neural Network Platform. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 3165–3166, 2019.

[44] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. GNNLab: A Factored System for Sample-Based GNN Training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, pages 417–434, 2022.

[45] Liangwei Yang, Zhiwei Liu, Yingtong Dou, Jing Ma, and Philip S Yu. ConsisRec: Enhancing GNN for Social Recommendation via Consistent Neighbor Aggregation. In *Proceedings of the 44th international ACM SIGIR conference on Research and development in information retrieval (SIGIR)*, pages 2141–2145, 2021.

[46] Shuangyan Yang, Minjia Zhang, Wenqian Dong, and Dong Li. Betty: Enabling Large-Scale GNN Training with Batch-Level Graph Partitioning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS)*, pages 103–117, 2023.

[47] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery and data mining (SIGKDD)*, pages 974–983, 2018.

[48] Muhan Zhang and Yixin Chen. Link Prediction Based on Graph Neural Networks. *In Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 31, 2018.

[49] Shichang Zhang, Jiani Zhang, Xiang Song, Soji Adeshina, Da Zheng, Christos Faloutsos, and Yizhou Sun. PaGE-Link: Path-Based Graph Neural Network Explanation for Heterogeneous Link Prediction. In *Proceedings of the ACM Web Conference (WWW)*, pages 3784–3793, 2023.

[50] Yanfu Zhang, Shangqian Gao, Jian Pei, and Heng Huang. Improving Social Network Embedding via New Second-Order Continuous Graph Neural Networks. In *Proceedings of the 28th ACM SIGKDD conference on knowledge discovery and data mining (SIGKDD)*, pages 2515–2523, 2022.

[51] Yijia Zhang, Yibo Han, Shijie Cao, Guohao Dai, Youshan Miao, Ting Cao, Fan Yang, and Ningyi Xu. Adam Accumulation to Reduce Memory Footprints of both Activations and Gradients for Large-Scale DNN Training. *arXiv preprint arXiv:2305.19982*, 2023.

[52] Yufeng Zhang, Xueli Yu, Zeyu Cui, Shu Wu, Zhongzhen Wen, and Liang Wang. Every Document Owns Its Structure: Inductive Text Classification via Graph Neural Networks. *arXiv preprint arXiv:2004.13826*, 2020.

[53] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. AIBox: CTR Prediction Model Training on a Single Node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 319–328, 2019.

[54] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. ByteGNN: Efficient Graph Neural Network Training at Large Scale. *the VLDB Endowment (VLDB)*, 15(6):1228–1242, 2022.

[55] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *Proceedings*

of IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3), pages 36–44. IEEE, 2020.

A Artifact Appendix

A.1 Abstract

The artifact provides the implementation code, testing preparations, and scripts for our LeapGNN system. It implements a feature-centric distributed GNN training method using model migration, based on the DGL framework.

A.2 Scope

The artifact facilitates a deeper understanding of our design and implementation details, including aspects not covered in the paper due to space constraints. It enables the reproduction of the experimental results presented in the paper and allows developers to integrate our system into their own applications.

A.3 Contents

The implementation includes the three key optimization techniques of LeapGNN. Additionally, the README.md file provides instructions for setting up the environment, downloading datasets, and using the testing scripts. Each test script corresponds to an evaluation figure in the paper, and all scripts are located in the test directory.

A.4 Hosting

The artifact is hosted at `https://github.com/ISCS-ZJU/LeapGNN-AE`. The latest content can be found in the distributed_version branch, with commit version 9af29d1e7e.