GraNNDis: Efficient Unified Distributed Training Framework for Deep GNNs on Large Clusters

Jaeyong Song
Seoul National University
Seoul, South Korea
jaeyong.song@snu.ac.kr

Hongsun Jang
Seoul National University
Seoul, South Korea
hongsun.jang@snu.ac.kr

Jaewon Jung
Seoul National University
Seoul, South Korea
jungjaewon@snuj.ac.kr

Youngsok Kim
Yonsei University
Seoul, South Korea
youngsok@yonsei.ac.kr

Jinho Lee
Seoul National University
Seoul, South Korea
leejinho@snu.ac.kr

ABSTRACT

Graph neural networks (GNNs) are one of the most rapidly growing fields within deep learning. According to the growth in the dataset and the model size used for GNNs, an important problem is that it becomes nearly impossible to keep the whole network on GPU memory. Among numerous attempts, distributed training is one popular approach to address the problem. However, due to the nature of GNNs, existing distributed approaches suffer from poor scalability, mainly due to the slow external server communications.

In this paper, we propose *GraNNDis*, an efficient distributed GNN training framework for training GNNs on large graphs and deep layers. GraNNDis introduces three new techniques. First, *shared preloading* provides a training structure for a cluster of multi-GPU servers. We suggest server-wise preloading of essential vertex dependencies to reduce the low-bandwidth external server communications. Second, we present *expansion-aware sampling*. Because shared preloading alone has limitations because of the neighbor explosion, expansion-aware sampling reduces vertex dependencies that span across server boundaries. Third, we propose *cooperative batching* to create a unified framework for full-graph and mini-batch training. It significantly reduces redundant memory usage in mini-batch training. From this, GraNNDis enables a reasonable trade-off between full-graph and mini-batch training through unification especially when the entire graph does not fit into the GPU memory. With experiments conducted on a multi-server/multi-GPU cluster, we show that GraNNDis provides superior speedup over the state-of-the-art distributed GNN training frameworks.

1 INTRODUCTION

Over the last few years, graph neural networks (GNNs) have received increasing attention among various types of deep learning models. As graphs can represent many non-structured data that do not fit into structured formats (e.g., images, or texts), they are successfully being used in various data-mining fields, such as social network user clustering [12], protein analysis [14], physics [51], to even traditional machine learning problems [4, 21].

Following a few pioneering GNN algorithms [2, 6, 27], various algorithms have been proposed [17, 56, 62, 66] that capture different aspects of the knowledge within graph data. Such development of the algorithms was followed by the advance of the training software frameworks which provide higher throughput as well as convenient interfaces for implementation [13, 22, 59].

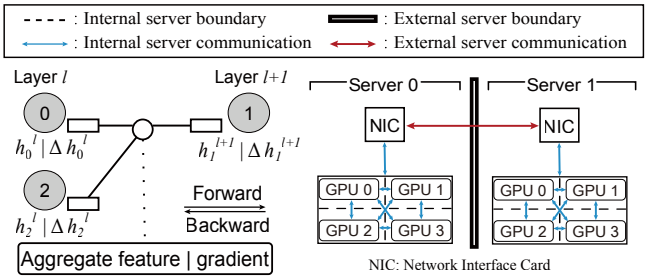
One classic challenge of GNN training is its large memory requirements [69] and extreme communication overhead. Because the real-world graph datasets can incorporate billions of vertices and even more edges [20, 29], the intermediate representations become too large for learner devices (e.g., GPUs) to hold in their device memories, and the communication amount is huge. The memory problem is often handled by deploying multiple GPUs to recruit more memory capacity [15, 22, 36–38, 60, 67, 69, 70, 72], which sometimes span over multiple servers. The cost of communication is tackled by many efficient frameworks [7, 47, 68], especially in the data management field.

However, the continuous evolution of GNNs creates several challenges to be addressed for efficient training frameworks:

(1) *GNNs require more memory than ever.* Traditional GNNs often targeted modestly sized graphs with thousands of vertices [49] with 2-5 layers. However, recent GNNs target much larger graphs that easily exceed millions of vertices [20, 29]. In addition, modern GNN structures now comprise tens [32], hundreds [33] to even a thousand layers [31]. This states that users will find it more challenging to keep the GNNs in GPU memory.
(2) *External server interconnect speed is not keeping up.* When multiple servers are used to increase aggregate memory capacity, the vertex dependencies cause frequent communications between the servers. However, even high-speed interconnects (around a few tens of GB/s) [42] are two orders of magnitude lower than that of the GPU device memory bandwidth [23] and several times slower than internal server interconnects [30].
(3) *GNN distributed training frameworks are disunited.* In GNN training, there are two popular methods: *full-graph* [22, 58] and *mini-batch training* [5, 66, 69, 70], where mini-batch training is a popular option when the system has insufficient memory. Unfortunately, current distributed GNN training frameworks support only one method of the two methods. This has a problem of forcing the model developer to fix the training method before designing GNNs.

To address the aforementioned problems in a more efficient way, we propose *GraNNDis* , a fast, unified framework for distributed GNN training. GraNNDis comprises three components.

Under review. First submitted on 2023-02-02.



(a) GNN vertex dependency (b) An example of GPU cluster

Figure 1: (a) An example of vertex dependency on neighbor vertices. (b) An example of a Multi-GPU cluster environment.

First, *shared preloading* provides a novel way of reducing the external server communication for GNN training on multi-server clusters. With shared preloading, each server preloads the vertex dependencies required for its partition, and distributes the dependency subgraph to the GPUs attached to it. Ideally, it reduces the external server communication to only twice per step: one at the beginning for inputs and the other at the end for gradient sharing.

Second, we devise a new sampling method named *expansion-aware sampling*. While shared preloading removes most of the external server communication, it suffers from neighbor explosion [5] when deep-layer GNNs are targeted. While the existing sampling methods mainly focus on manipulating the whole graph structure, we expose the server interconnect architecture to the sampling algorithm. By applying a sampling algorithm that is aware of the external server boundary, the neighbor explosion problem can be mitigated without compromising the final accuracy.

Third, *cooperative batching* extends shared preloading and expansion-aware sampling to create large-sized batches for mini-batch training, making GraNNDis a *unified* framework of full-graph training and mini-batch training. In theory, full-graph training can be regarded as a special case of mini-batch training. Surprisingly, even when the cluster can accommodate entire graph, the largest batch size of a mini-batch training is severely limited by the memory size of individual GPUs. By cooperative batching, we enlarge the maximum mini-batch size beyond that of the single GPU memory size, closing the gap toward the full-graph training.

Together, GraNNDis builds an efficient, unified framework for large-scale GNN training. We evaluate GraNNDis on a cluster of multiple servers, and demonstrate its superior throughput over several existing GNN frameworks.

2 BACKGROUND

2.1 Graph Neural Network (GNN) Training

GNNs [27, 56, 62] show powerful potential for graph representation learning. Node, edge, and graph properties can be learned through GNNs. These properties learned through GNNs are widely used for diverse tasks [8, 31–33]. Modern GNNs structures most commonly use neighbor dependency information ($\mathcal{N}$) to generate the final representation. Because of this property, GNNs recursively require neighbor vertices' hidden vectors of the previous layer to generate the next representation of vertex v. This is called *vertex dependency* in GNN training depicted in Figure 1a. Every layer (l) of GNN generates the next hidden vector h_v^l for each target vertex v by aggregating hidden vectors of v's neighbors $\mathcal{N}(v)$, multiplying weights of layer (W^l), and lastly applying activation function (σ). The aggregated vector ($h_{\mathcal{N}(v)}^l$) is often combined with the previous hidden vector of target vertice h_v^{l-1}. This can be formulated as:

$$h_{\mathcal{N}(v)}^l = AGGREGATE(\{h_u^{l-1} | u \in \mathcal{N}(v)\}), \quad (1)$$

$$h_v^l = \overrightarrow{F}^l(h_v^{l-1}, h_{\mathcal{N}(v)}^l). \quad (2)$$

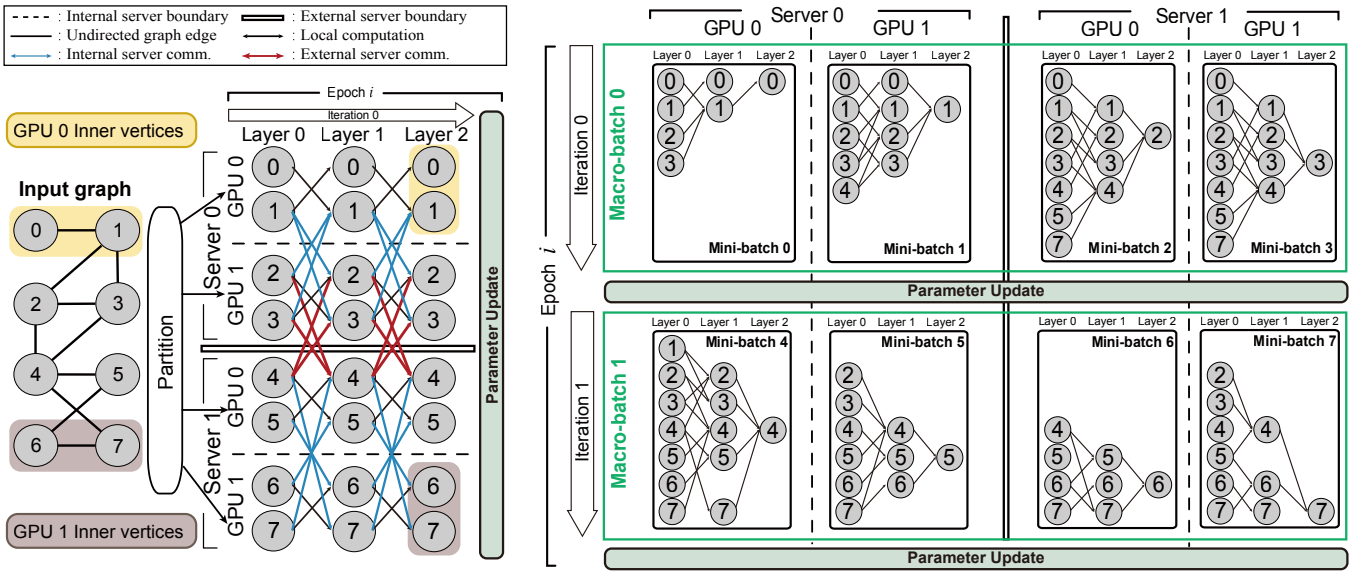
where $AGGREGATE(\cdot)$ is an aggregation function, usually the summation of all the neighbor hidden vectors. $\overrightarrow{F}^l(\cdot)$ is comprised of multiplication of weights (W^l) followed by activation function (σ). Then, by comparing the final representation with the true label, the loss function of the model is computed and its weight is updated through backward propagation. This kind of dependency processing on neighbor vertices is a unique attribute of GNNs compared to other neural network models [28, 52]. Most GNNs mainly suffer from this aggregation process of intermediate representation, causing a severe slowdown in training [71].

Traditional GNNs [17, 27] used 3-5 layers and reported that using deeper layers even shows lower accuracy. Recent deeper graph neural networks [32, 33, 40] mitigate this issue by using residual connections and pre-activation to show improved accuracy in various areas. Especially, deep GNNs are famous for reducing the over-smoothing [45, 64] effect of GNNs, which is the main bottleneck of improving accuracy in complex datasets [14, 20]. In such deep GNNs, the training becomes much harder because they require much higher memory requirements and heavier computations than traditional GNNs. A representative problem is a *neighbor explosion*, which is the result of the increased *Message Flow Graph* (MFG) size by the number of layers. Because the size of intermediate hidden features grows along with the number of layers [31], using multiple accelerators (e.g., GPUs) is becoming more common to overcome such limitations, as illustrated in Section 2.2.

2.2 Distributed GNN Training

To mitigate the outbursting memory usage of GNN training, two representative distributed training methods are used: full-graph training and mini-batch training. *Full-graph training* [22, 57, 58, 60] uses the whole graph representations for every parameter update. As depicted in Figure 2a, all vertex dependencies have to be fully computed for full-graph training. Therefore, a single iteration (model parameter update) essentially becomes an epoch. For this, all the hidden vectors for every layer have to be loaded on the GPUs' memory, so multiple GPUs are usually used for large graphs [20, 29] and deep GNNs [32, 33] in a distributed manner.

In distributed full-graph training, inter-GPU communications are essential to retrieve intermediate vertex representations from other GPUs, as illustrated by **blue** and **red** arrows in Figure 2a. The overhead from this communication is significant, so it generally uses a partitioning algorithm [25] to minimize the communication volume when partitioning a graph for GPUs. However, most existing works [22, 38, 67] do not consider multi-GPU cluster settings, relying all the dependency on slow external communications. GraNNDis provides higher throughput than previous works



(a) Distributed full-graph training approach

(b) Distributed macro-batch training approach with mini-batch size = 1

Figure 2: Forward-pass of a 2-layer GNN. (a) Distributed full-graph training is conducted on two servers, and each server has two GPUs. (b) Distributed mini-batch training with macro-batch size = 4. Each GPU process the mini-batch with size = 1.

by considering multi-GPU cluster environment such as Figure 1b, especially the difference between the internal and external communication bandwidth.

Mini-batch training [5, 59, 66, 69, 70] is a technique used when the GPU memory capacity is insufficient for full-graph training. It uses the batch concept to mitigate the memory capacity issue. It packs the partial dependencies into MFGs for the fixed number of target vertices to mitigate the heavy memory overheads, as illustrated by Mini-batch *i* in Figure 2b. By adjusting batch size, training can be performed with flexible memory requirements, at the cost of some redundant memory usage and computation between mini-batches.

Mini-batch training can be easily extended to distributed training [15, 69, 70] by forming *macro-batch*es as depicted with **green** boxes in Figure 1b. Each GPU loads the vertex dependencies as if it processes a mini-batch so that there are no dependencies between GPUs. After each GPU processes their mini-batch, weight update is performed together as a macro-batch by sharing the gradients among the GPUs. In this case, the (macro-) batch size is the mini-batch size × the number of GPUs. Compared to full-graph training, an epoch requires several iterations from the macro-batches to handle all vertices. However, this implementation scales poorly because the GPU-wise mini-batch makes extreme redundancy of memory usage and computations, as we will demonstrate in Section 3.

2.3 Sampling Methods

Because of high memory requirements and heavy dependencies processing overhead of GNNs, many sampling methods [3, 5, 17, 48, 50, 66] are actively explored. While they all share the goal of reducing the number of dependencies of GNN training, they can be roughly divided into *layer-wise sampling approaches* and *subgraph sampling approaches*. Layer-wise sampling approaches [3, 17, 48] apply sampling algorithm layer by layer, usually constraining the maximum degree of edges called *fanout*. Subgraph sampling approaches [5, 66] samples subgraph from the original graph and execute forward, backward propagation on sampled subgraph only.

Mini-batch training approaches [15, 59, 69, 70] are commonly merged with sampling methods. However, recent works [9, 37] show that the overhead of sampling could be severe and degrade the training throughput. GraNNDis introduces a new sampling method to alleviate neighbor explosion considering the characteristic of multi-GPU cluster environment while providing a stable accuracy.

3 MOTIVATIONAL STUDY

Figure 4 displays the result of the motivational study, showing the limitations of current full-graph GNN training [58]. All GNN training is conducted on the four-server cluster, where each server has four RTX A6000 GPUs. We used deep GNNs with residual connections [32]. For the detailed setup, please refer to Section 8.1.

External communication bottleneck. In GNNs, full-graph training is usually the preferred method to get stable and high accuracy in a short training time. However, it suffers from high memory overhead [69] to load intermediate features of all vertices on GPU memory. For instance, a large-size dataset (Reddit) on a 56-layer GNN mandates at least a total of 16 GPUs to conduct full-graph training, and an intermediate-size dataset (Products) with 112 layers requires the same-sized cluster.

Figure 4 also shows the training time breakdown of state-of-the-art full-graph training framework [58] on three widely used datasets. We break down the training time into four parts; computation, external, internal server communication, and parameter gradient synchronization. With the multi-server setting, the external server communication becomes a bottleneck, because it is relatively slower

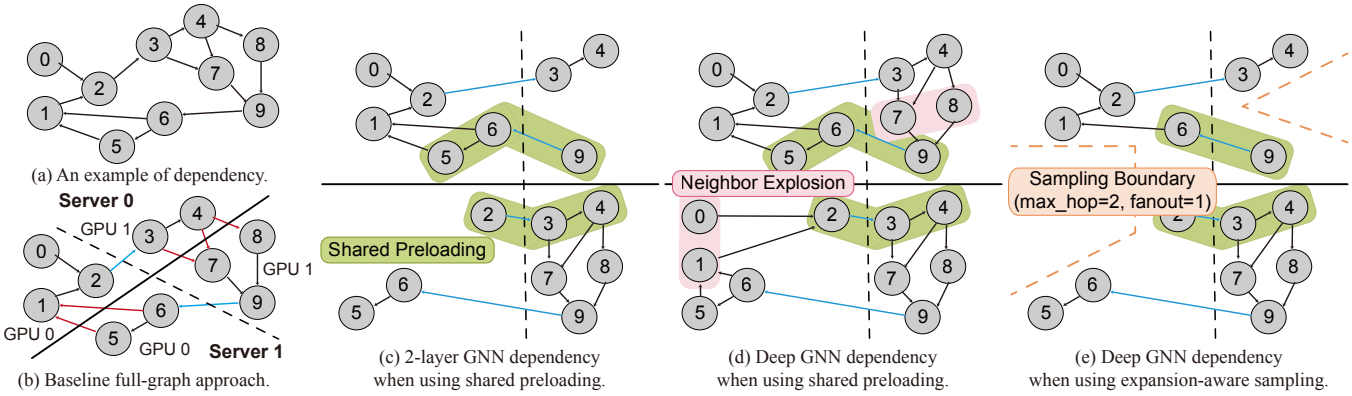


Figure 3: Overall structure of GraNNDis compared to previous distributed full-graph training. (a) Another example of vertex dependencies. (b) Baseline approach to process the vertex dependencies. (c) The use of shared preloading to process 2-layer GNN. (d) The neighbor explosion when using shared preloading with deep GNN (#layers > 3). (e) An illustration of how expansion-aware sampling tackles the neighbor explosion with the external max-hop value of 2 and external fanout value of 1.

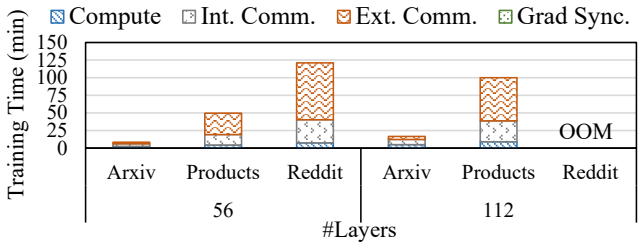


Figure 4: Training time breakdown of full-graph training.

than the internal communication and the current full-graph GNN training frameworks do not consider the difference.

Disunited training frameworks on large batches. With 112-layer GNNs, the Reddit dataset does not fit on the 16-GPU cluster anymore. In this case, using mini-batch training with smaller batches should solve the problem. Unfortunately, no existing distributed training framework supports the transition from full-graph training to mini-batch training. Moreover, even when we port the model to the state-of-the-art distributed mini-batch training framework [70], it also runs out of memory – even with the mini-batch size of 1! There are multiple causes, but we find the major reason to be the redundancy caused by the mini-batches being formed in individual GPUs. This severely limits the size of the mini-batch, preventing large mini-batches to be formed. We identify this as the disunited region of distributed training frameworks. Theoretically, full-graph training is a special case of mini-batch training where the number of batches is 1. GraNNDis makes this possible, by forming mini-batches through aggregated memories of multiple GPUs. Later in the evaluation (Section 8.3), we show that GraNNDis can train the 112-layer GNN for Reddit, or even much larger datasets in the same cluster, demonstrating the benefit of the unified framework.

4 GRANNDIS DESIGN

In this section, we propose the main schemes of GraNNDis, whose overall structure is depicted in Figure 3. First, we introduce shared preloading, which considers the multi-server environments and prevents the use of the slow interconnect by preloading vertex dependencies (Figure 3c). Second, expansion-aware sampling reduces redundant computation and internal server communication of shared preloading (Figure 3d) by sampling neighbor vertices with awareness of the external server boundary without compromising the final accuracy (Figure 3e). Third, GraNNDis supports a unified framework for both full-graph and mini-batch training with cooperative batching.

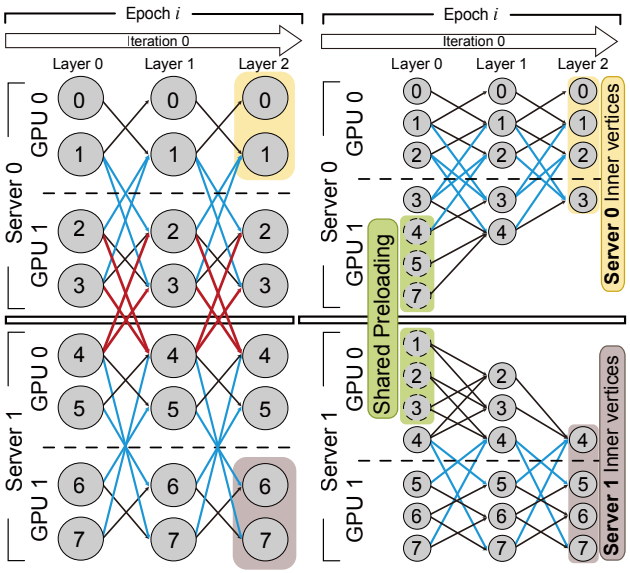
4.1 Notations and Performance Model

For analysis, we set some notations to explain the latency model of distributed GNN training. N_s and N_g indicate the total number of servers and the number of GPUs per server, respectively. V means the total number of vertices in a given graph and E is the number of edges. C is the computational throughput measured in vertices per second. $B_{internal}$ and $B_{external}$ are internal and external server bandwidths split per GPU, which are also measured in vertices per second. L is the total number of GNN layers, which means that the model extracts the information from L-hop neighbors. In the analysis, we assume that hidden dimension sizes and vertex feature sizes are equal in every layer for convenience.

We formulate the training time of the previous full-graph training method (T_{prev}) as follows, concentrating on the communication of the vertex dependencies which cannot be overlapped.

$$T_{prev} = \frac{V}{N_s N_g C} + \frac{E}{N_s N_g}\left[\frac{((N_g-1)/N_g N_s)}{B_{internal}} + \frac{(1-1/N_s)}{B_{external}}\right]. \quad (3)$$

The first term of Equation (3) presents the total compute time of the previous full-graph training. The second term calculates the total communication time, which is the summation of internal- and external server communication time. One observation from the communication term is that the communication for external server has larger numerators (more external server GPUs) and smaller denominators (lower external interconnect bandwidth). This makes external server communication become the bottleneck under the latency model. The parameter gradient synchronization



(a) Full graph training baseline (b) Shared Preloading

Figure 5: An illustration of shared preloading. (a) Baseline full-graph training suffers from slow external server communication. (b) Shared preloading mitigates this issue using faster internal server communication. The green area represents the preloaded initial graph features.

by all-reduce communication is omitted because it is almost entirely overlapped by computation through gradient bucketing [46].

4.2 Shared Preloading

In Section 3, we identified that previous distributed GNN training methods do not consider the external and internal server bandwidth difference of cluster environments. Shared preloading utilizes this difference to reduce external server communications. As illustrated in Figure 5a, existing full-graph training methods [22, 57, 58] use external server communications (red arrows) in the same manner as internal-server communications (blue arrows). However, the external-server connections (e.g., Ethernet, Infiniband) are usually an order of magnitude slower than internal-server connections (e.g., PCIe, NVLink) in cluster environments. Therefore, it is important to avoid using external server connections as much as possible.

To minimize external server communication, the vertex dependencies should be handled in another way. Shared preloading uses a preloading strategy to fulfill such dependencies. Figure 5b illustrates the details of shared preloading. Inspired by mini-batching methods [69, 70], shared preloading calculates the L-hop dependency graph in a server level, and preloads the initial features of vertices that belong to the dependency subgraph (green boxes). Then, instead of fulfilling those dependencies by transferring them from the remote servers, the preloaded vertex features are used to compute the intermediate features, satisfying the vertex dependencies.

Figure 5b provides an example of shared preloading on the eight vertices input graph from Figure 2. The total vertices are split into two servers, so vertices 0-3 belong to server 0, which are called inner vertices of server 0. To produce the output of those vertices at the last layer (layer 2), intermediate features of vertex 0-4 at layer 1 are needed. The features of vertices 0-3 are calculated within the same server, but vertex 4 is not in the inner vertices set. Instead of transferring it from server 1, the preloaded input features from vertices 2-5 and 7 are used to calculate the feature of vertex 4 at layer 1, which is used to produce outputs of vertices 2 and 3. Without considering vertex dependency, the inner vertices and preloaded vertices are split in the server and assigned to each GPU. Some vertex dependencies between GPUs are handled by communication through high internal server connections. Note that the partition in Figure 5b is drawn to be imbalanced for a clear view. In practice, we apply partition algorithm [25] when distributing vertices such that the vertex features are well-balanced. In fact, shared preloading has more redundant computation across servers as a cost for reduced external server communication. This is advantageous when the computation power of the devices far exceeds that of the communication, which perfectly matches the environment for multi-GPU clusters.

From Equation (3), we can derive the training time and the expected speedup using shared preloading. Here, we assume that the dependency graph size grows at the rate of D^α every layer, where D is the average degree of vertices, and $0 < \alpha < 1$ is a graph-dependent per-layer expansion factor to represent overlapping neighbors. The empirical value of D is around 10 and D^α is around $1.5 - 2.0$. The external server bandwidth ($B_{external}$) is replaced by the computation of preloaded vertices. In addition, the preloaded vertices increase the amount of internal server bandwidth. With these considerations, the training time changes as follows.

$$T_{preload} = \frac{VD^\alpha L}{N_s N_g C}, + \frac{ED^\alpha L}{N_s N_g}\left[\frac{(N_g-1)/N_g}{B_{internal}}\right], \tag{4}$$

$$1 < D^\alpha L \leq N_s N_g. \tag{5}$$

The condition for $T_{preload}$ to be shorter than T_{prev} from Equation (3) is $T_{preload} - T_{prev} < 0$. Making some rough approximations as $1/B_{internal} \approx 0$ and $1/N \approx 0$, the condition becomes

$$\frac{VD^\alpha L}{N_s N_g C} - \frac{V}{N_s N_g C} - \frac{E}{N_s N_g}\frac{1}{B_{external}} < 0. \tag{6}$$

Because $E = V \cdot D$ by definition,

$$\frac{C}{B_{external}} > \frac{(D^\alpha L - 1)}{D}. \tag{7}$$

Equation (7) suggests how much gap is needed between computational throughput and communication bandwidth for shared preloading to gain speedup. Using $D^\alpha = 2$ and $D = 10$ for a three-layer GNN ($L = 3$), shared preloading has an advantage when the processing rate (vertex / second) of computation is at least half that of the external communication. This depends on many factors, such as the average degree and the number of weight parameters.

4.3 Expansion-aware Sampling

Real graphs are known to have power-law degree distribution [29], so when a GNN becomes deeper, the vertex dependencies from the neighbors expand quickly. This is known as the neighbor explosion phenomenon [5, 57]. When considering recent GNNs [31–33] have

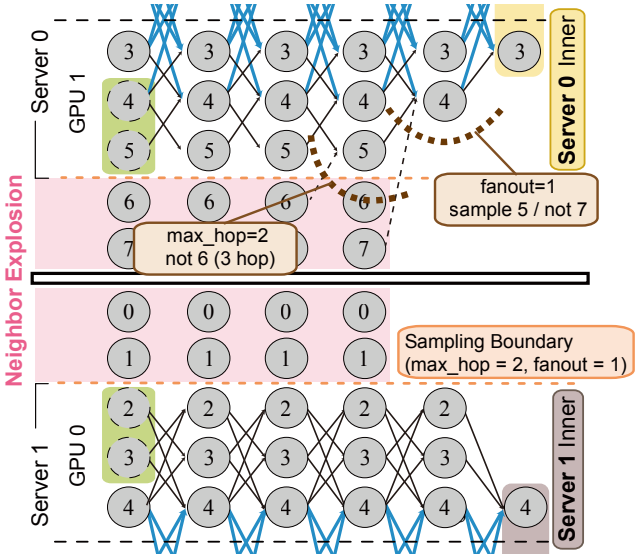
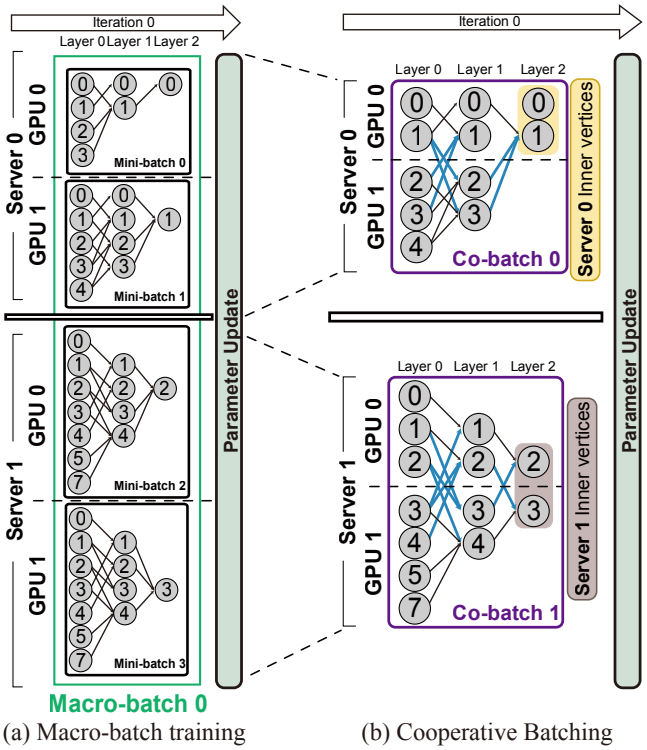


Figure 6: Expansion-aware sampling with a multi-GPU cluster. Expansion-aware sampling applies a sampling method on the server boundary nodes to mitigate such an issue.



(a) Macro-batch training (b) Cooperative Batching

Figure 7: An illustration of cooperative batching. (a) Conventional mini-batch approach where the mini-batch size is bounded by the memory size of a single GPU. (b) Cooperative batching uses larger batches.

more than 6 layers, shared preloading may preload all dependencies because of the neighbor explosion as illustrated in Figure 3e. Equation (7) also suggests that shared preloading alone is difficult to scale to deep-layer GNNs as the right-hand side gets larger with increasing L.

One widely used solution in GNN training is sampling strategy [5, 17, 50, 66], which sparsifies the dependencies to reduce the computation and neighbor explosion of GNNs. Although they provide speedup over the non-sampled training, they often fail to provide stable convergence [22, 58, 60]. With shared preloading method, we observe that the sparsification of dependencies between GPUs in the same server does not contribute much to the speedup, while still having a detrimental effect on the convergence.

Therefore, we propose expansion-aware sampling, an efficient sampling method for shared preloading method, which is depicted in Figure 6. When shared preloading decides the vertices for each GPU which are not in inner vertices, expansion-aware sampling is applying constraints for deciding whether a certain vertex is included. We define two tunable parameters, external $max\_hop(m)$ and external $fanout(k)$. In Figure 6, we provide an example of $max\_hop = 2$ and $fanout = 1$. If we apply expansion-aware sampling, shared preloading selects external vertices which are at a maximum two-hop distance ($max\_hop = 2$) from the server's inner vertices. Additionally, expansion-aware sampling restricts the hop traversing to find only one external neighbor ($fanout = 1$). Therefore, the vertices from 4 to 7 were to be preloaded by Server 0 when using shared preloading, but through expansion-aware sampling, shared preloading decides only to preload the vertex 4 and 5, and does not preload 6 and 7, illustrated by the light red area. When traversing is finished, shared preloading with expansion-aware sampling generates a subgraph from the traversed vertices and it does not sample the internal dependency of the batched graph. Empirically, we used $max\_hop = 1$ and $fanout = 15$ settings because the number of inner vertices is at least 10K which is a sufficiently large batch to contain the whole graph's structural information. We provide the detailed sensitivity of the tunable parameters in Section 8. The cost of expansion-aware sampling is formulated as below:

$$T_{sampling} = \frac{Vm^{\alpha}k}{N_s N_g C} + \frac{Em^{\alpha}k}{N_s N_g}\left[\frac{(N_s N_g - 1)}{B_{internal}}\right], \tag{8}$$

$$1 < m^{\alpha}k \ll D^{\alpha}L \leq N_s N_g. \tag{9}$$

While this preserves the number of edges in the perspective of the whole graph, it has the effect of reducing the growth to $m^{\alpha k}$, much smaller than $D^{\alpha L}$ since $m \ll D$ and $k \ll L$.

4.4 Cooperative Batching

In Section 4.2 and Section 4.3, we have shown how shared preloading and expansion-aware sampling can be used to perform full-graph training. In this section, we introduce cooperative batching, a unique opportunity enabled by shared preloading for mini-batch training. In current mini-batch training frameworks, the vertex dependencies are fetched by individual GPU. In other words, the target vertices are split first, and then their dependencies are fetched. This split-then-fetch strategy for batch generation not only results in a lot of memory inefficiency but also greatly limits the size of a

mini-batch. With cooperative batching, we use the GPUs within a server to cooperate as if it were a single large GPU to enable a much larger batch size, as illustrated in Figure 7. We first choose the target vertices to form a *co-batch* and fetch-then-split their dependencies to each server using the same manner as Section 4.2. This fetch-then-split strategy for batch generation has a clear advantage of much less redundant memory and thus a larger batch size. Most importantly, we already preload the vertex dependency through shared preloading and expansion-aware sampling, so we can choose server-wise mini-batch-based training with a much large batch size when the entire graph does not fit into the GPU memories for full-graph training.

The disadvantages are increased internal server communication, and possibly imbalanced workload between workers. However, both are not severe overheads because the internal bandwidth is very high, and the number of vertices in each batch is greatly increased. These small penalties are dwarfed by the accuracy and speed benefits of being able to train with larger batch sizes.

5 OVERALL GRANNDIS PROCEDURE

In this section, we explain the overall procedure of GraNNDis, which is illustrated in Section 4, in an algorithmic manner. The pseudo-code of GraNNDis is displayed in Algorithm 1. At first, subgraphs ($S_{i,N}$) are extracted from an input graph (G) in each server, while considering shared preloading and expansion-aware sampling (line 4). These subgraphs are overlapped with each other for shared preloading ($S_i \cap S_j \neq \emptyset \; for \; i \neq j$). The amount of overlapping among subgraphs is determined by the two parameters (k, m) from expansion-aware sampling. GraNNDis supports mini-batch training with cooperative batching and therefore each subgraph is additionally partitioned if GPU memory is not enough to hold the subgraphs ($S_{i,N}$) (line 6). GraNNDis generates mini-batches until partitioned subgraphs ($S_{i,d,R}$) become small enough for the GPUs.

At the beginning of the training loop, GraNNDis loads subgraphs ($S_{i,w,b}$) in each server, which are generated in lines 2 to 8 (line 17). If mini-batch training is not used, GraNNDis loads subgraphs only once, since GPUs already hold them. The previous full graph training methods need to aggregate (*AGGREGATE*) feature vectors of neighboring vertices through all-to-all communication from the whole cluster. However, through shared preloading, these feature vectors are all located in the same server, so GraNNDis only communicates from internal server GPUs (lines 21-24). For every layer l, the hidden feature of vertices is updated with this aggregation (*AGGREGATE*) before forward ($\overrightarrow{F}^l$) and backward ($\overleftarrow{F}^l$) operation of GNN training (line19-34). All-reduce operation is followed by these operations in order to synchronize parameter gradients (U_i) computed by each GPU (line 36). Weights are updated by these parameter gradients (line 37). And then each GPU unloads the assigned subgraphs ($S_{i,w,b}$) in this training iteration when mini-batch training is used (line 40).

Algorithm 1 GraNNDis Procedure

Input:
$G(V, E)$: Graph, n: #servers, N: #GPUs per server,
L: #layers, T: #epochs, η : learning rate,
k, m: shared preloading external max-hop and fanout,
R : #minibatch
$\{Y_v | v \in V\}$: node labels (truth)
Output:
Updated parameters W_T^l, $(l = 1, 2, ..., L)$

1: // Subgraph generation and partition
2: **for parallel** each server i **do**
3: // Shared preloading and expansion-aware sampling
4: $[S_{i,1}, S_{i,2}, ..., S_{i,N}] = EXTRACT(G, k, m, N, i)$
5: **for** *device* $d = 1, 2, ...N$ **do**
6: $[S_{i,d,1}, S_{i,d,2}, ..., S_{i,d,R}] = PARTITION(S_{i,d}, R, i)$
7: **end for**
8: **end for**
9:
10: // Training loop
11: **for parallel** each $GPU \; w = 1, 2, ..., N$ in each server i **do**
12: **for** *epoch* $t = 1, 2, ..., T$ **do**
13: **for** *batch* $b = 1, 2, ..., R$ **do**
14: // Load graph and features
15: **if** $R > 1$ OR $t == 0$ **then**
16: // If R==1, do not need to load after the first epoch
17: $LOAD(S_{i,w,b})$
18: **end if**
19: **for** *layer* $l = 1, 2, ..., L$ **do**
20: **for** each $\{v | v \in S_{i,w,b}\}$ **do**
21: **for** each $\{u | u \notin S_{i,w,b}, \; u \in \mathcal{N}(v)\}$ **do**
22: // Internal server communication for fetching
23: fetch h_u^{l-1} from GPUs in the same server i
24: **end for**
25: $h^l_{\mathcal{N}(v)} = AGGREGATE(\{h_u^{l-1} | u \in \mathcal{N}(v)\})$
26: $h_v^l = \overrightarrow{F}^l(h_v^{l-1}, h^l_{\mathcal{N}(v)})$
27: **end for**
28: **end for**
29: $loss = f_{Loss}(h_v^L, Y_v) \; for \; v \in S_{i,w,b}$
30: $J_v^L = \frac{\partial loss}{\partial h_v^L}$
31: // Use similar fetching as forward
32: **for** *layer* $l = L, L-1, ..., 1$ **do**
33: $U_i^l = \overleftarrow{F}^l(J_v^L, h_v^l)$
34: **end for**
35: // Parameter gradient synchronization
36: $U = All\_REDUCE(U_i)$
37: $W_t = W_{t-1} - \eta \cdot U$
38: // Unload graphs and features if R > 1
39: **if** $R > 1$ **then**
40: $UNLOAD(S_{i,w,b})$
41: **end if**
42: **end for**
43: **end for**
44: **end for**

6 IMPLEMENTATION

We implement GraNNDis based on [58] which significantly improves the training throughput of [22] and has state-of-the-art performance. It uses GLOO [11] which is slower in GPU RDMA communication than NCCL [43], so we reimplement it with the NCCL. In addition, while original [58] uses custom parameter gradient synchronization, we modified it to use the PyTorch DistributedDataParallel (DDP) package to provide overlapping of gradient synchronization. These optimizations show some speedup, so we set this optimized version [58] as our full-graph training baseline.

Table 1: Comparison of GraNNDis to Prior Art

Name	Baseline Code	System			Method		
		Worker	Multi-server	Multi-GPU	Full-graph	Mini batch	Batch Limit
PyG [13]	PyG	CPU	✗	N/A	✓	✓	CPU memory
DGL [59]	DGL	CPU	✗	N/A	✓	✓	CPU memory
AGL [67]	Custom	CPU	✓	N/A	✓	✓	CPU memory
NeuGraph [38]	Custom	GPU	✗	✓	✓	✗	N/A
NeutronStar [60]	LibTorch	GPU	✓	✗	✓	✗	N/A
ROC[22]	FlexFlow	GPU	✓	✓	✓	✗	N/A
PipeGCN [58]	DGL	GPU	✓	✓	✓	✗	N/A
DistDGL [69, 70]	DGL	GPU	✓	✓	✗	✓	GPU memory
GraNNDis (Proposed)	DGL	GPU	✓	✓	✓	✓	Aggregate GPUs

Table 2: Experimental Environment

HW	Server	GPU	4× NVIDIA RTX A6000
		CPU	1× EPYC 7302, 16C 32T
		Memory	256GB DDR4 ECC
		Int. connect	PCIe 4.0 / NVLink Bridge
	Cluster	#Servers	4
		Ext. connect	Infiniband QDR (32Gbps)
SW	Common	Python	3.10
		PyTorch	1.13
		CUDA	11.6.2
		NCCL	2.10.3
	GNN	DGL	0.9.1 [59]
		Model	ResGCN+ [33]
		Hidden dim.	64
		Task	Node classification

Table 3: Graph Datasets

Size	Name	Dataset Info.			Hyper-parameter		
		#Nodes	#Edges	Feat. size	lr	Dropout	#Epochs
Small~medium	ogbn-arxiv [20]	169K	1.2M	128	0.01	0.5	1000
	ogbn-products [20]	2,449K	61.9M	100	0.003	0.3	1000
	Reddit [63]	232K	114.6M	602	0.01	0.5	1000
Hyperscale	ogbn-papers100M [20]	111M	1.6B	128	0.01	0.5	3000

For general graph-related operations and GNN layers, we used DGL [59] framework. As illustrated in Section 5, shared preloading creates subgraphs for preloading and partitions the server-dedicated subgraphs. For this, we make a 'head group' comprising the first process of each server, and this head group handles creating subgraphs and partitioning. To support cooperative batching with more than one iteration, GraNNDis load and unload the target partition and related features in each iteration.

7 COMPARISON WITH PRIOR ART

Table 1 qualitatively compares GraNNDis with prior art. Notably, GraNNDis is the first unified framework for a multi-server, multi-GPU framework that supports both full-graph and mini-batch training. Furthermore, using cooperative batching, GraNNDis allows for training of large graph, deep layer GNNs which was previously not possible with the prior art, as we will discuss in Section 8.

8 EVALUATION

8.1 Experimental Environment

Cluster Environments and Model Configuration. Our cluster environments are listed in Table 2. We use clusters equipped with four RTX A6000 GPUs, an AMD EPYC 7302 CPU, and 256GB system memory. Our cluster uses Infiniband QDR (32 Gbps) for external connection between servers. PCIe 4.0 is used as default for internal connection unless otherwise noted. We use ResGCN+ [33] as our default model, and fixed the hidden dimension setting to 64 for a fair comparison.

Datasets. We choose four representative graph datasets; ogbn-arxiv (Arxiv), ogbn-products (Products) [20], Reddit [63], and ogbn-papers100M (Papers) [20]. Arxiv is a citation network where each edge indicates that a source paper cites a destination paper. Products represents a co-purchasing network, and each node represents a product in Amazon. Reddit dataset consists of nodes which are posts in the online forum. Two nodes in the Reddit dataset are linked if the same user leaves comments on the two posts. Papers is also a citation network that has orders-of-magnitude larger size than other datasets. It is a representative hyper-scale dataset, so we dedicated a separate section in Section 8.7. The characteristic of each graph dataset is summarized in Table 3. We mainly used hyperparameters chosen in [58].

Baselines. Two representative baselines [58, 70] were chosen in our experiments. As described in Table 1, [70] uses mini-batch training and neighborhood sampling in each training iteration. [58] uses full-graph training and tends to have higher accuracy and throughput than mini-batch-based training.

8.2 Training Speedup and Breakdown

Figure 8 shows the speedup of GraNNDis in full-graph training of GNNs compared to the full-graph training baseline [58] (FG). In addition to the default setting where server-internal communications are done via PCIe channel, we test an additional setting where NVLink Bridges [44] are used for high-speed internal server connection. We chose 3, 28, and 56 layers, which are widely used as the number of layers for deep GNNs [32, 33]. We fixed the hidden dimension size as 64, which is the common hidden dimension size for deep GNNs with 28 layers. For expansion-aware sampling, we used 1-hop and 15-fanout sampling as a default, if not stated.

In GNNs with a shallow number of layers, shared preloading provides speedup over the baseline full-graph training, as shown in Figure 8. This speedup comes from the minimizing of external server communication by preloading the vertex dependencies. In the Arxiv dataset with the 56-layer full-graph training, external server communication consumes 41.7% of the total training time. Shared preloading removes this huge portion and provides a 1.65× speedup in PCIe-only settings and 1.80× speedup on settings with NVLink Bridges by benefiting from the high-speed internal server connection.

However, due to the redundant computation and internal communication in deeper layer GNNs, as we discussed in Section 4.3, shared preloading sometimes shows a slowdown in deeper GNNs, especially with the PCIe-only setting. For example, in the Product dataset with a 56-layer case, shared preloading minimizes the external communication, which is 68.6% in the baseline full-graph training. Even though the speedup is 1.77× in the setting with NVLink Bridges, it shows a 14.2% slowdown with relatively slower PCIe-only settings due to a significant amount of redundancy. Expansion-aware sampling restricts the neighbor explosion and minimizes

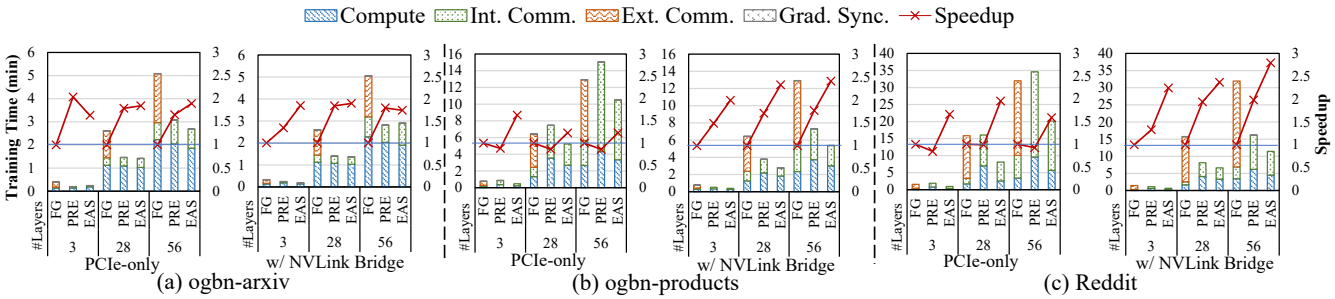


Figure 8: Speedup and time breakdown of the baseline full-graph training (FG), shared preloading (PRE), and expansion-aware sampling (EAS) in shallow to deep GNNs.

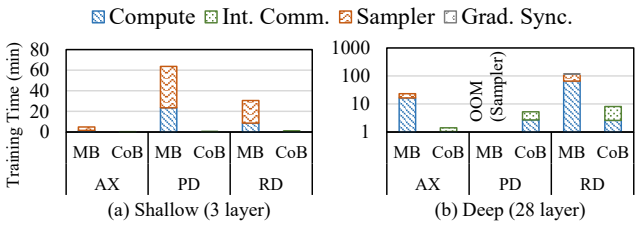


Figure 9: Training time breakdown of cooperative batching (CoB) and the DistDGLv2 mini-batch training (MB) in shallow and deep GNN under the four-server cluster.

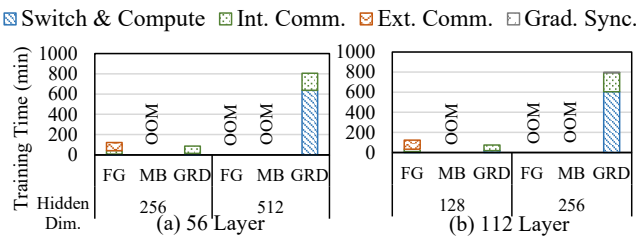


Figure 10: Training time breakdown of the full-graph training (FG), DistDGLv2 mini-batch training (MB), and GraNNDis (GRD) on deeper and wider GNNs for the Reddit dataset.

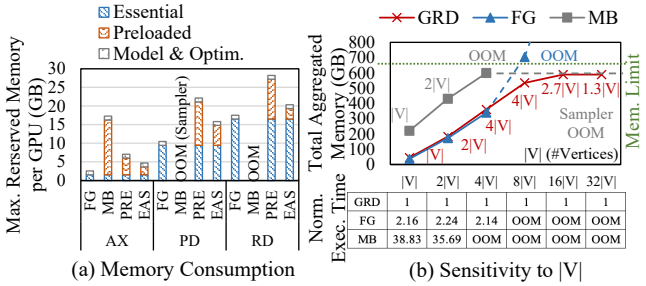
such redundancies. When expansion-aware sampling is adopted, much of the redundant computation and internal server communications are reduced, and it provides 1.89×, 1.23×, and 1.59× speedup on Arxiv, Products, and Reddit datasets, respectively in 56-layer deep GNN even on PCIe-only settings.

In Figure 9, we show comparison between GraNNDis and DistDGL, a famous mini-batch-based training method (MB). Both methods use batch size as large as the memory can handle. Because DistDGL uses sampling methods as default, we use GraphSAGE sampling as the baseline. We used fixed fanout of 20 for the DistDGL sampler following the popular setting. For a fair comparison, we trained them for an equal number of parameter updates (1000 times) to our training method. Note that the comparison with MB is also conducted using the same four-server cluster. For a fair comparison with no advantage from using a pipelining, we used the vanilla version of [58], which is essentially a refactored version of existing full-graph training.

In Figure 9, we break down the training time of MB and GraNNDis into four parts: compute, internal communication, sampler time, and gradient synchronization time. In all cases, the gradient synchronization time among the GPUs and servers is almost negligible because the model size of GNNs is relatively smaller than the traditional DNNs. Cooperative batching provides superior speedup over the baseline DistDGL from 14.38× to 136.24× speedup. The speedup mainly comes from the sampler overhead of the DistDGL, which still cannot be overlapped even when we enable the prefetching feature of the newest version of the DistDGLv2. It aligns with the prior works [24, 68] that mention the huge overhead of the sampler in mini-batch training. Additionally, it is partly because the sampling ability of the 16-core 32-thread server cannot sufficiently fulfill the computational throughput of the four RTX A6000 GPUs in a server. One interesting aspect of training DistDGL is that it shows slower training time for the Product dataset compared to the Reddit dataset, unlike the full-graph training. The sampler overhead is proportional to the total number of vertices because it searches the layerwise dependency of all vertices. On the other hand, the main bottleneck of GraNNDis and the full-graph training is dependency communication which is proportional to the total number of edges. Therefore, the Product dataset, which has a much larger total number of vertices than the Reddit dataset, takes more time to train with DistDGL. As a result, cooperative batching provides much more speedup in the Product dataset than the Reddit dataset, 136.24× speedup in the shallow 3-layer GNN. In Figure 9b, we can find another drawback of the DistDGL sampler. The memory overhead of the sampler is also proportional to the total number of vertices, so in our setting with 256GB DRAM, the sampler gives an out-of-memory error. Additionally, both GraNNDis and DistDGL are sampling-based methods, so we provide direct time-to-accuracy comparison in Section 8.8.

8.3 Unified Comparison using Deeper Networks

In this section and Figure 10, we demonstrate how much benefit GraNNDis bring by providing a unified framework between full-graph training and mini-batch training. We trained deeper GNNs of 56- and 112-layers GNN, with the hidden dimension size from 128 to 512. Surprisingly, in our cluster environment, both the full-graph training (FG) and the mini-batch training (MB) baselines often fail



	|V|	2|V|	4|V|	8|V|	16|V|	32|V|
GRD	1	1	1	1	1	1
FG	2.16	2.24	2.14	OOM	OOM	OOM
MB	38.83	35.69	OOM	OOM	OOM	OOM

Figure 11: (a) Memory consumption of the baseline full-graph training, shared preloading, and expansion-aware sampling. (b) Memory consumption of the baselines (FG, MB) and GraNNDis (GRD) when the graph size becomes large.

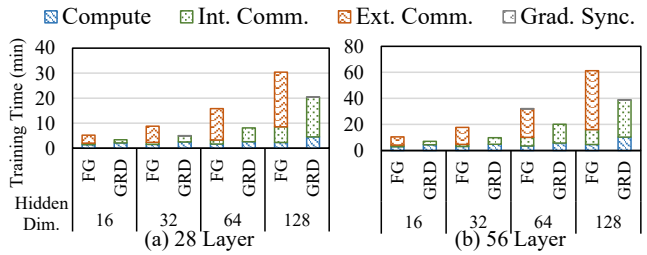


Figure 12: Training time breakdown of GraNNDis (GRD) and the full-graph training (FG) on Reddit dataset with various hidden dimension sizes.

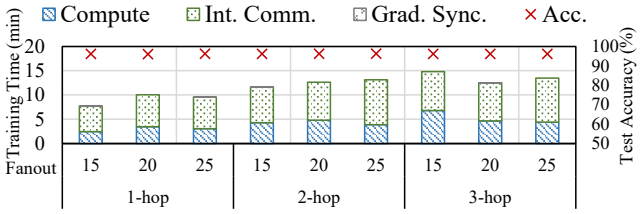


Figure 13: Sensitivity of expansion-aware sampling on sampling hyper-parameter with Reddit dataset using 28-layer and 64 hidden dimension size GNN.

from out-of-memory errors. In full-graph training, it is a natural phenomenon because it requires enough GPU memory to store all vertex feature for all layers. The mini-batch training is expected to run by forming smaller mini-batches. However, MB failed to train them even at the mini-batch size of 1, despite our effort on enabling checkpointing [46] for saving memory space. It is mainly because the neighbor explosion over hundred layers causes too much redundancy, making the size of the mini-batch almost equal to that of the entire graph.

On the other hand, GraNNDis uses a fetch-then-split strategy in cooperative batching. Because it generates a batch at a server level with aggregated GPUs, heavy memory overhead from deeper and wider GNNs can be mitigated. Expansion-aware sampling also strongly supports the training of deep GNNs with limited resources. As a result, GraNNDis is the only choice to train the Reddit graph dataset in a 112-layer and 256-hidden-dimension with the cluster under test. Additionally, in the settings where the full-graph training can be conducted, GraNNDis provides 1.41× and 1.64× speedup in 56- and 112-layer GNNs, respectively.

8.4 Analysis on Memory Consumption

Figure 11a shows the memory consumption of the baselines, shared preloading, and expansion-aware sampling with a representative deep GNN setting of 56 layers and 256 hidden dimensions. In deep-layer settings, neighbor explosion incurs redundant memory usage (shaded orange), but expansion-aware sampling successfully mitigates this issue.

Figure 11b gives further insights about the memory usage of the baselines and GraNNDis. We generated random synthetic graphs of various numbers of vertices, following the average degree (around 492) of the Reddit dataset ($|V|$=the total number of vertices in Reddit). We fixed the model as a 28-layer with a hidden size 64. We indicated the maximum batch sizes for GraNNDis and mini-batch training by colored numbers ($n|V|$). The memory usage gap between the full-graph training and GraNNDis is small, but GraNNDis consumes slightly more memory due to shared preloading. When the graph becomes larger, the full-graph training cannot handle it due to the memory limit. In contrast, GraNNDis keeps training by reducing the bath size to fit the memory. On the other hand, mini-batch training consumes significantly large memory compared to GraNNDis with the same batch size due to the redundant memory usage. Surprisingly, mini-batch training fails to train large graphs over $4|V|$ because of its inefficiency in handling dependency graphs or the sampler out-of-memory, as discussed in Section 8.2 and Section 8.3. Even when the baselines can handle the training, GraNNDis provides a stable speedup of at least 2.14× and 35.69× over full-graph training and mini-batch training, respectively.

8.5 Sensitivity Studies

Hidden dimensions. Figure 12 shows the hidden dimension size sensitivity of GraNNDis on the Reddit dataset with 28- and 56-layer GNNs. In overall, GraNNDis provides consistent 1.49-1.95× speedup over the baseline full-graph training. As the hidden dimension size grows up, the portion of external server communication increases. This translates to slightly more advantage to GraNNDis who provide more speedup when the hidden dimension is wider.

Sampling hyperparameters. Figure 13 illustrates how further hop and larger fanout setting changes training time and accuracy. We tested the 1- to 3-hop setting with a 15- to 25-fanout setting because the 15-25 fanout is widely used in existing sampling techniques. As we sample further hops, the training time increases, however, after 2-hop with a 25-fanout setting, the training time does not increase because we sampled an almost full graph due to neighbor explosion. On the other hand, all settings show similar accuracy, which means that GraNNDis can reach full accuracy even with 1-hop with a 15-fanout setting. For this reason, we set this as the default setting for all experiments in other sections.

Partitioning algorithm. In all types of distributed GNN training, graph partitioning is essential for distributing the workload

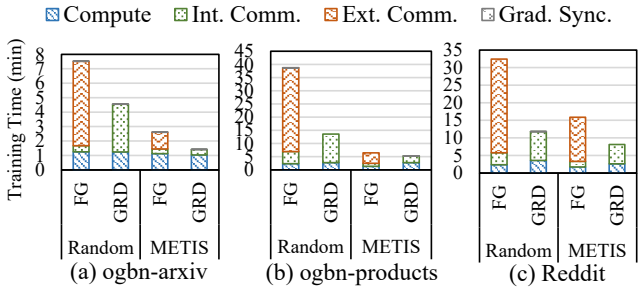


Figure 14: Sensitivity of the full-graph training (FG) and GraNNDis (GRD) on partitioning algorithm.

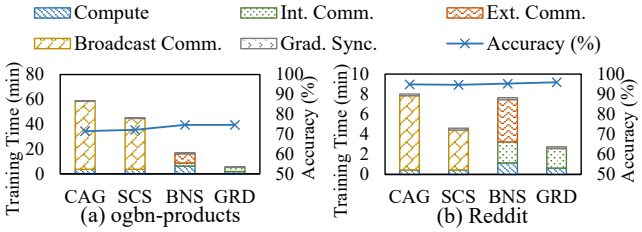


Figure 15: Training time breakdown and accuracy of GraNNDis (GRD) compared with sampling-based (BNS-GCN, BNS) and staleness-based (Sancus, SCS) methods.

to all workers. GraNNDis adopts min-cut-based partitioning of METIS [25]. Figure 14 shows the sensitivity on partitioning algorithms. In random partition cases, external communication consumes up to 93.2% of the total training time. By adopting GraNNDis in such cases, external communication is greatly reduced, and it provides 1.65×, 2.85×, and 2.75× speedup over FG for Arxiv, Products, and Reddit datasets, respectively. This shows that GraNNDis does not rely on a specific partitioning algorithm.

8.6 Other Sampling and Staleness Methods

Some prior works introduce sampling- [57] or staleness- [47] based methods to address the massive communication overhead of full-graph training. Therefore, we directly compare the training time and accuracy of GraNNDis with previous methods in Figure 15. CAGNET [55] (CAG) reduces communication with partitioning strategies and broadcast communications, and Sancus [47] (SCS) uses staled historical embeddings. In addition, BNS-GCN [57] (BNS) samples the communication to be around 10% and uses stale activations. We directly compared GraNNDis with the above three prior works on the Products and Reddit dataset while following the open-sourced code of [47]. We used three layers containing two GCN layers and a single FC layer with a hidden size of 256.

As displayed in Figure 15, the communication volume of broadcast-based methods (CAG and SCS) is large because it is proportional to the square of the total number of vertices. In contrast, the communication volume of all-to-all-based methods (BNS and GRD) is proportional to the total number of edges. Therefore, BNS-GCN and GraNNDis provide more speedup over CAGNET on the Product dataset, which has a lower graph density (#Edges/#Vertices)

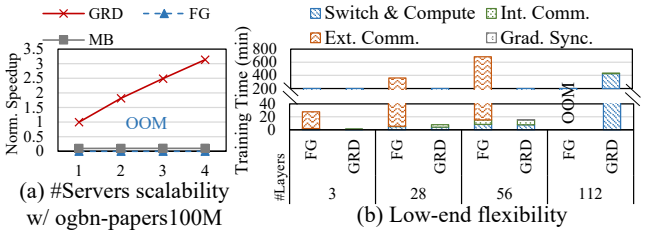


Figure 16: Experiments of cluster and dataset scalability. (a) Scalability of GraNNDis with the hyper-scale ogbn-papers100M dataset. (b) Training time breakdown on a low-end cluster with Reddit dataset.

than the Reddit dataset. GraNNDis shows sufficient speedup over all prior works from 1.68× to 10.74×. In addition, GraNNDis provide higher and more stable accuracy compared to the baselines.

8.7 Scalability and Flexibility

To observe the scalability of GraNNDis, we use ogbn-papers100M (Papers hereafter), a well-known hyperscale dataset. The results are shown in Figure 16a. We used three layers and a hidden dimension size 64 for training. Because full-graph training fails to train from out-of-memory issues, we compared GraNNDis with the mini-batch-based baseline (MB) using DistDGL. We normalized all results to the training time of GraNNDis running on a single server. Following [69, 70], we weak-scaled DistDGL with each GPU processing 1024 batch and trained until it convergence, around 25 epochs. GraNNDis provides a huge 10.29× to 31.26× speedup compared to the baseline and shows great scalability when the number of servers increases. In addition, GraNNDis achieves 63.14%, similar to the accuracy reported in [20] with SGC [61] of wider hidden size than our setting. Unfortunately, the weak-scaled DistDGL training often fails to achieve such accuracy, so we trained it separately with a total batch size of 1024 and got 60.30%. Since GraNNDis only takes around 16 minutes to train hyper-scale Papers dataset with four servers to achieve such accuracy, it is highly efficient for limited computing resources with hyper-scale datasets.

GraNNDis supports transitioning from full-graph training to mini-batch training. This enables training even in memory-limited environments, which provides flexibility on lower-end clusters. We equipped a lower-end cluster with two servers, each with four RTX 2080Ti (11GB device memory), connected with 1GbE external bandwidth. In various layer settings with a hidden size 64, GraNNDis provides 29.07× to 46.64× speedup on the Reddit dataset compared to the full-graph training baseline, which shows the importance of considering external server communication in distributed GNN training. As illustrated in Figure 16b, the baseline full-graph training (FG) cannot handle Reddit dataset with 112 layers. GraNNDis (GRD) can mitigate such cases by allowing five updates per epoch.

8.8 Accuracy Comparison

Expansion-aware sampling is a sampling technique, so it is essential to compare accuracy with the baselines. Figure 17 is the time-to-accuracy plot of the baselines and GraNNDis while training 28-layer GNN with 64 hidden dimension size. We fixed the total number of

Table 4: Speedup and Accuracy of GraNNDis

Method	ogbn-arxiv						ogbn-products						Reddit					
	3-layer		28-layer		56-layer		3-layer		28-layer		56-layer		3-layer		28-layer		56-layer	
	SPD	Acc. (%)	SPD	Acc. (%)	SPD	Acc. (%)	SPD	Acc. (%)	SPD	Acc. (%)	SPD	Acc. (%)	SPD	Acc. (%)	SPD	Acc. (%)	SPD	Acc. (%)
FG	-	70.72	-	70.44	-	70.40	-	76.01	-	75.50	-	74.58	-	96.20	-	96.13	-	96.39
GraNNDis	1.64×	70.57	1.84×	70.50	1.89×	70.68	1.63×	75.44	1.23×	75.21	1.23×	74.88	1.66×	96.06	1.95×	96.14	1.59×	96.08

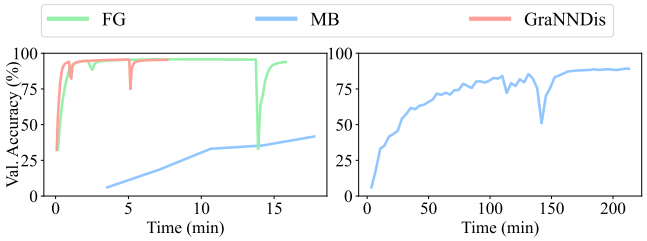


Figure 17: Time-to-accuracy comparison on Reddit dataset training with the baselines and GraNNDis. The mini-batch baseline (MB) is reported separately for visibility (right).

parameter updates as 1000 for a fair comparison. The mini-batch training shows a much slower convergence speed compared to GraNNDis and the full-graph training. GraNNDis achieves the final accuracy 1.95× faster than the full-graph training. We also directly compare the speedup and accuracy of the full-graph training (FG) and GraNNDis in Table 4. It is shown that GraNNDis provides superior speedup from 1.23× to 1.95×, without compromising accuracy.

9 RELATED WORK

Large Graph and Deep Graph Neural Network There have been attempts to increase the layer of GNN by borrowing many techniques of traditional CNN models [18, 19]. [32, 33] applied residual connection and preactivation to deep GNN layers in order to mitigate the over-smoothing issues. These deep GNNs have shown superior accuracy in large graph datasets [20, 53]. In order to meet memory limitation, many previous [3, 10, 17, 65] works use sampling method in GNN training. [24, 39] further tried to mitigate the overhead of sampling by overlapping data transfer time with computation time. On the other hand, [5, 66] use the subgraph structure of graphs and find neighbors within its subgraph. [36, 37, 65] also focused on this overhead of preparing data for GNN training and developed a cache policy in order to reduce feature retrieving costs. [57] noticed this memory access and communication costs mainly come from boundary nodes and applies sampling to those.

Graph Processing and Partitioning Many graph processing frameworks usually take variants of Gather-Apply-Scatter approach [16]. However, this approach does not support some GNN operations such as GAT [56]. On the other hand, existing DNN frameworks [1, 35, 46] assume each input is independent of each other and distribute training samples to workers without considering dependency among samples. In order to achieve load-balancing and consider this dependency, [73] set a heuristic objective which minimizes edges crossing each partition. Furthermore, [15] even partitions feature dimensions across workers and adopts existing parallel approaches [26, 35, 41] to reduce communication. One popular approach to reduce communication is to partition graphs. METIS [25] is widely used due to its superior performance.

Distributed Training in GNN There are many recent works [7, 36, 38, 47, 54, 55, 69, 70, 72] which aim to perform distributed training in a large graph. Most distributed training frameworks focus on how to handle the dependencies. [72] separately stores attributes of graphs in different workers and caches frequent neighbors. [67] designed k-hop neighborhood for each node and therefore enables parameter-server [34] based training. However, [67, 72] only support CPU-based GNN training. [69, 70] try to handle dependency among vertices with caching and therefore suffer from the redundant computation. On the other hand, [7, 22, 47, 54, 55, 58] attempt to solve this dependency with communication. [55] uses sequential broadcast methods to handle the vertex dependency among distributed workers. [47] further accelerates [55] through using historical embeddings to avoid pricy communication. [54] considers the low-end servers, but it is limited to CPU servers. By focusing on the limitation of each approaches, a hybrid approach was proposed [60], but with an unrealistic assumption that each server has only one GPU. [38] combined graph computation optimization with data partitioning and scheduling, and [7] adopted hypergraph partitioning model to encode communication cost and optimized communication operations. However, the prior works [36, 38, 65] do not carefully consider multi-server and multi-GPU environment, which has a large gap between internal and external server bandwidth. To the extent of our knowledge, we propose the first approach to consider the multi-server multi-GPU environment.

10 CONCLUSION

In this paper, we propose GraNNDis, which is a unified framework for distributed GNN training. Because the existing full-graph training framework suffers from slow external-server communication, we introduce novel techniques to minimize it. From those techniques, we found a new chance to transition from full-graph training to mini-batch training, when a graph does not fit into memory. Compared to the existing mini-batch training framework, GraNNDis is more memory efficient and provides superior speedup. As a result, GraNNDis enables faster and much more memory-efficient distributed GNN training compared to the prior art.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for Large-Scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[2] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann Lecun. Spectral networks and locally connected networks on graphs. In *International Conference on Learning Representations (ICLR)*, 2014.

[3] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: Fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations (ICLR)*, 2018.

[4] Zhao-Min Chen, Xiu-Shen Wei, Peng Wang, and Yanwen Guo. Multi-label image recognition with graph convolutional networks. In *IEEE/CVF conference on computer vision and pattern recognition (CVPR)*, 2019.

[5] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2019.

[6] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

[7] Gunduz Vehbi Demirci, Aparajita Haldar, and Hakan Ferhatosmanoglu. Scalable Graph Convolutional Network Training on Distributed-Memory Systems. *Proceedings of the VLDB Endowment (VLDB)*, 2022.

[8] Ailin Deng and Bryan Hooi. Graph neural network-based anomaly detection in multivariate time series. In *AAAI conference on artificial intelligence (AAAI)*, 2021.

[9] Jialin Dong, Da Zheng, Lin F Yang, and George Karypis. Global neighbor sampling for mixed cpu-gpu training on giant graphs. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2021.

[10] Jialin Dong, Da Zheng, Lin F. Yang, and George Karypis. Global neighbor sampling for mixed cpu-gpu training on giant graphs. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2021.

[11] facebook. GLOO, 2023. https://github.com/facebookincubator/gloo, visited on 2023-02-01.

[12] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph Neural Networks for Social Recommendation. In *The World Wide Web Conference (WWW)*, 2019.

[13] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *ICLR Workshop on Representation Learning on Graphs and Manifolds (ICLRW)*, 2019.

[14] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. Protein interface prediction using graph convolutional networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[15] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[16] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[17] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE/CVF conference on computer vision and pattern recognition (CVPR)*, 2016.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision (ECCV)*, 2016.

[20] 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. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[21] Binxuan Huang and Kathleen Carley. Syntax-aware aspect level sentiment classification with graph attention networks. In *Conference on Empirical Methods in Natural Language Processing and International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019.

[22] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Conference on Machine Learning and Systems (MLSys)*, 2020.

[23] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. Hbm (high bandwidth memory) dram technology and architecture. In *IEEE International Memory Workshop (IMW)*, 2017.

[24] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E Leiserson, and Jie Chen. Accelerating training and inference of graph neural networks with fast sampling and pipelining. In *Conference on Machine Learning and Systems (MLSys)*, 2022.

[25] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 1998.

[26] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. Strads: A distributed framework for scheduled model parallel machine learning. In *European Conference on Computer Systems (EuroSys)*, 2016.

[27] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[28] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[29] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2005.

[30] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems (IEEE TPDS)*, 2020.

[31] Guohao Li, Matthias Müller, Bernard Ghanem, and Vladlen Koltun. Training graph neural networks with 1000 layers. In *International Conference on Machine Learning (ICML)*, 2021.

[32] Guohao Li, Matthias Muller, Ali Thabet, and Bernard Ghanem. Deepgcns: Can gcns go as deep as cnns? In *International Conference on Computer Vision (ICCV)*, October 2019.

[33] Guohao Li, Chenxin Xiong, Ali Thabet, and Bernard Ghanem. Deepergcn: All you need to train deeper gcns. *arXiv preprint arXiv:2006.07739*, 2020.

[34] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[35] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[36] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *ACM Symposium on Cloud Computing (SoCC)*, 2020.

[37] 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. *arXiv preprint arXiv:2112.08541*, 2021.

[38] 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 *USENIX Annual Technical Conference (ATC)*, 2019.

[39] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. Large graph convolutional network training with gpu-oriented data communication architecture. *Proceedings of the VLDB Endowment (VLDB)*, 2021.

[40] Zhewei Wei Ming Chen, Bolin Ding Zengfeng Huang, and Yaliang Li. Simple and Deep Graph Convolutional Networks. In *International Conference on Machine Learning (ICML)*, 2020.

[41] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[42] NVIDIA. InfiniBand Network, 2023. https://docs.nvidia.com/networking/display/MLNXOFEDv493150/InfiniBand+Network,visited on 2023-01-30.

[43] NVIDIA. NCCL, 2023. https://github.com/NVIDIA/nccl,visited on 2023-02-01.

[44] NVIDIA. NVLink Bridge, 2023. https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/,visited on 2023-06-01.

[45] Kenta Oono and Taiji Suzuki. Graph neural networks exponentially lose expressive power for node classification. In *International Conference on Learning Representations (ICLR)*, 2020.

[46] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration, 2017.

[47] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. Sancus: Staleness-Aware Communication-Avoiding Full-Graph Decentralized Training in Large-Scale Graph Neural Networks. *Proceedings of the VLDB Endowment (VLDB)*, 2022.

[48] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. Dropedge: Towards deep graph convolutional networks on node classification. In *International Conference on Learning Representations (ICLR)*, 2020.

[49] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 2008.

[50] Marco Serafini and Hui Guan. Scalable Graph Neural Network Training: The Case for Sampling. *SIGOPS Oper. Syst. Rev.*, 2021.

[51] Jonathan Shlomi, Peter Battaglia, and Jean-Roch Vlimant. Graph neural networks in particle physics. *Machine Learning: Science and Technology*, 2020.

[52] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2014.

[53] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. Arnetminer: Extraction and mining of academic social networks. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2008.

[54] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[55] Alok Tripathy, Katherine Yelick, and Aydın Buluç. Reducing Communication in Graph Neural Network Training. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[56] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[57] Cheng Wan, Youjie Li, Ang Li, Nam Sung Kim, and Yingyan Lin. Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. In *Conference on Machine Learning and Systems (MLSys)*, 2022.

[58] Cheng Wan, Youjie Li, Cameron R. Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. PipeGCN: Efficient Full-Graph Training of Graph Convolutional Networks with Pipelined Feature Communication. In *International Conference on Learning Representations (ICLR)*, 2022.

[59] 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.

[60] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. NeutronStar: Distributed GNN Training with Hybrid Dependency Management. In *International Conference on Management of Data (SIGMOD)*, 2022.

[61] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying Graph Convolutional Networks. In *International Conference on Machine Learning (ICML)*, 2019.

[62] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019.

[63] Pinar Yanardag and SVN Vishwanathan. Deep graph kernels. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2015.

[64] Chaoqi Yang, Ruijie Wang, Shuochao Yao, Shengzhong Liu, and Tarek Abdelzaher. Revisiting over-smoothing in deep gcns. *arXiv preprint*, 2020.

[65] 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 *European Conference on Computer Systems (EuroSys)*, 2022.

[66] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. Graphsaint: Graph sampling based inductive learning method. In *International Conference on Learning Representations (ICLR)*, 2020.

[67] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. Agl: A scalable system for industrial-purpose graph machine learning. *Proceedings of the VLDB Endowment (VLDB)*, 2020.

[68] 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. *Proceedings of the VLDB Endowment (VLDB)*, 2022.

[69] 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. *arXiv preprint arXiv:2010.05337*, 2020.

[70] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2022.

[71] Zhe Zhou, Cong Li, Xuechao Wei, Xiaoyang Wang, and Guangyu Sun. Gnnear: Accelerating full-batch training of graph neural networks with near-memory processing. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2022.

[72] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: A comprehensive graph neural network platform. *Proceedings of the VLDB Endowment (VLDB)*, 2019.

[73] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.