Scaling Graph Neural Networks: Innovations in Distributed and Decentralized Training for Billion-Scale Graphs

Yuyang Liang
The Chinese University of Hong Kong, Shenzhen
Shenzhen, China
119010174@link.cuhk.edu.cn

Abstract

Graph neural networks (GNNs) have emerged due to their success at modeling graph data. Yet, it is challenging for GNNs to efficiently scale to large graphs. To tackle this challenge, we develop DistDGL, a system for training GNNs in a mini-batch fashion on a cluster of machines. DistDGL distributes the graph and its associated data (initial features and embeddings) across the machines and uses this distribution to derive a computational decomposition by following an owner-computer rule. Furthermore, to reduce the communication cost and improve the CPU utilization and end-to-end performance of distributed GNN training systems, we propose ByteGNN. ByteGNN provides an abstraction of graph sampling to support high parallelism, applies a two-level scheduling strategy to improve resource utilization, and designs a graph partitioning algorithm tailored for GNN. At last, to avoid communication caused by expensive data movement between workers, we propose Sancus, a staleness-aware communication-avoiding decentralized GNN system. By introducing a set of novel bounded embedding staleness metrics and adaptively skipping broadcasts, Sancus abstracts decentralized GNN processing as sequential matrix multiplication and uses historical embeddings via cache. Experiment results show that the proposed methods effectively address the challenges of scalability of GNNs.

ACM Reference Format:
Yuyang Liang. 2024. Scaling Graph Neural Networks: Innovations in Distributed and Decentralized Training for Billion-Scale Graphs. In *Proceedings of Shenzhen, Guangdong (Conference).* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Graph Neural Networks (GNNs) have shown success in learning from graph-structured data and have been applied to many graph applications in social networks, recommendations, knowledge graphs, etc. In these applications, graphs are usually huge, in the order of many millions of nodes or even billions of nodes. For instance, Facebook social network graph contains billions of nodes. Amazon is selling billions of items and has billions of users, which forms a giant bipartite graph for its recommendation task. Natural language processing tasks take advantage of knowledge graphs, such as Freebase [29] with 1.9 billion triples.

It is challenging to train a GNN model on a large graph. Unlike domains such as computer vision and natural language processing, where training samples are mutually independent, graph inherently represents the dependencies among training samples (i.e., vertices). Hence, mini-batch training on GNNs is different from the traditional deep neural networks; each mini-batch must incorporate those depending samples. The number of depending samples usually grows exponentially when exploring more hops of neighbors. This leads to many efforts in designing various sampling algorithms to scale GNNs to large graphs [7, 8, 31, 35, 78]. The goal of these methods is to prune the vertex dependency to reduce the computation while still estimating the vertex representation computed by GNN models accurately.

It gets even more challenging to train GNNs on giant graphs when scaling beyond a single machine. For instance, a graph with billions of nodes requires memory in the order of terabytes attributing to large vertex features and edge features. Due to the vertex dependency, distributed GNN training requires to read hundreds of neighbor vertex data to compute a single vertex representation, which accounts for majority of network traffic in distributed GNN training. This is different from traditional distributed neural network training, in which majority of network traffic comes from exchanging the gradients of model parameters. In addition, neural network models are typically trained with synchronized stochastic gradient descent (SGD) to achieve good model accuracy. This requires the distributed GNN framework to generate balanced mini-batches that contain roughly the same number of nodes and edges as well as reading the same account of data from the network. Due to the complex subgraph structures in natural graphs, it is difficult to generate such balanced mini-batches.

Unfortunately, current systems cannot effectively address the challenges of distributed GNN training. Previous distributed graph analytical systems [27, 58, 69] are designed for full graph computation expressed in the vertex-centric program paradigm, which is not suitable for GNN mini-batch training. Existing domain-specific frameworks for training GNNs, such as DGL [75] and PyTorch-Geometric [20], cannot scale to giant graphs. They were mainly developed for training on a single machine. Although there have been some efforts in building systems for distributed GNN training, they either focus on full batch training by partitioning graphs to fit the aggregated memory of multiple devices [38, 57, 70] or suffer from the huge network traffic caused by fetching neighbor node data [1, 80, 86]. System architectures [12, 49, 63] proposed for training neural networks for computer vision and natural language

Conference, Dec. 6th,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

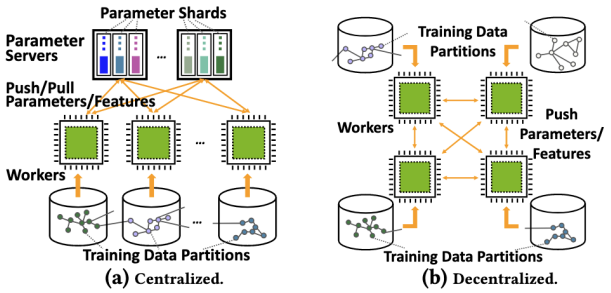


Figure 1: Centralized vs. decentralized GNN processing example. In the centralized scheme, workers periodically send updates to a parameter server. In the decentralized scheme, workers exchange them directly.

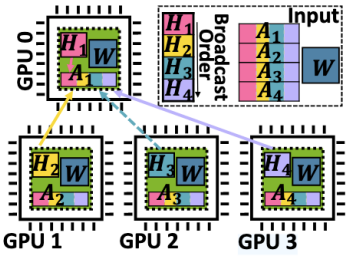


Figure 2: A toy 2-layer GNN example on SANCUS: GPU_{i-1} keeps its shards of H_i and A_i , with a full W ; H_i are sent to all GPUs in order via one-to-all broadcast (arrows omitted without loss of generality). After 4 sequential broadcasts →,→,→,→ and on-device computation, since the broadcasts are in parallel, all H_i updates for GPU_{i-1}. Next, SANCUS may tolerate H_3 to skip 1 broadcast as shown by - - → . In total, only 4 + 3 = 7 broadcasts are needed.

processing are not directly applicable because one critical bottleneck in GNN training is the network traffic of fetching neighbor node data due to the vertex dependencies, while previous systems majorly focuses on network traffic from exchanging the gradients of model parameters.

In this work, we first develop DistDGL on top of DGL to perform efficient and scalable mini-batch GNN training on a cluster of machines. It provides distributed components with APIs compatible to DGL's existing ones. As such, it requires trivial effort to port DGL's training code to DistDGL. Internally, it deploys multiple optimizations to speed up computation. It distributes graph data (both graph structure and the associated data, such as node and edge features) across all machines and run trainers, sampling servers (for sampling subgraphs to generate mini-batches), and in-memory KVStore servers (for serving node data and edge data) all on the same set of machines. To achieve good model accuracy, DistDGL follows a synchronous training approach and allows ego networks forming the mini-batches to include non-local nodes. To reduce network communication, DistDGL adopts METIS [43] to partition a graph with minimum edge cut and co-locate data with training computation.

We conduct comprehensive experiments to evaluate the efficiency of DistDGL and effectiveness of the optimizations. Overall, DistDGL achieves 2.2× speedup over Euler [1] on a cluster of four CPU machines. The main performance advantage comes from the efficient feature copy with 5× data copy throughput. DistDGL speeds up the training linearly without compromising model accuracy as the number of machines increases in a cluster of 16 machines and easily scales the GraphSage [31] model to a graph with 100 million nodes and 3 billion edges. It takes 13 seconds per epoch to train on such a graph in a cluster of 16 machines.

While distributed mini-batch sampling has become the default method for GNN training on a large graph (for which full-batch training and full mini-batch training are not practical), existing distributed GNN training systems suffer from a number of performance problems. One main problem is that sampling can take significantly longer time to complete than training, due to large amounts of random data access and remote data fetching involved in the sampling phase. The imbalance between the sampling and training phases also leads to the under-utilization of computing resources and the problem is worsen if GPUs are used for training (which further widens the gap between the sampling and training time) [66]. To address this imbalanced computing pattern in mini-batch GNN training, existing systems have attempted to apply neighborhood caching [53] and fixed size prefetching [83] to shorten the sampling time. However, it is difficult to set the right hyper-parameters (i.e., cache ratio and prefetching number) for training different GNN models on different graphs. Graph partitioning has also been applied to reduce the cost of remote data fetching [86]. However, existing graph partitioning algorithms are designed for traditional graph workloads (e.g., distributed PageRank) and they do not consider the data access pattern and load balancing in GNN training.

To address these problems, we further propose ByteGNN, a distributed GNN training framework to support fast end-to-end GNN training in large graphs. To improve the efficiency of sampling, we abstract the sampling phase of a mini-batch as a directed acyclic graph (DAG) of small tasks, so that we can run DAGs and tasks within each DAG in parallel. The fine-grained task abstraction in DAG modeling also leads to the design of a two-level scheduling. First, coarse-grained scheduling determines how much resources should be used for minibatch sampling, in order to dynamically adjust the computation loads between the sampling and training phases to avoid resource contention and maximize CPU utilization. Then, fine-grained scheduling decides the execution order of tasks in the DAGs in order to pipeline the sampling outputs to be consumed by the training phase at the right pace. The two scheduling strategies work together to minimize the end-to-end GNN training time. We also propose an effective graph partitioning algorithm tailored for mini-batch graph sampling, which maintains the data locality according to the data access pattern of mini-batch sampling and balances the computation loads in the training, validation and testing stages.

We implemented ByteGNN based on GraphLearn [86]. Our performance evaluation shows that ByteGNN achieves significantly higher training throughput and is more scalable than the state-of-the-art distributed GNN systems. Experimental results show that

ByteGNN achieves up to 23.8x speedup over GraphLearn and 3.5x over DistDGL. The results verify that our system designs lead to efficient GNN training.

In distributed training, the underlying system architecture of how workers communicate is crucial, especially for GNNs with substantial communication overhead. As illustrated in Figure 1, two approaches exist: centralized and decentralized. Though most distributed GNN systems [23, 39, 57, 83, 86] work in the popular centralized parameter server (PS) scheme in Figure 1(a), they often pay the price of heavy preprocessing and complex workflow, in pursuit of efficiency and scalability. By nature of GNNs, the intensive communication between all the workers and the central PS plus the waiting time for stragglers may lead to high communication overhead [5]. Decentralized architectures, however, can be more robust and easier to deploy, by avoiding the inconvenience in implementing and tuning a PS, centralized bottleneck bandwidth, and single point of failure [48]. Especially for large neural networks, the decentralized scheme is proven to be more superior theoretically [52].

To fill this gap in efficient GNN processing, we propose Sancus, a staleness-aware communication-avoiding decentralized GNN training system via adaptively skipping broadcast and caching historical embeddings with bounded staleness. To bypass the irregular data communication between GPUs, we **firstly** revisit the parallel algorithms to distribute GNNs [70] and decrease communication overhead in a fundamentally distinct way. As Figure 2 shows, by regarding the GNN processing purely as a sequence of matrix multiplication operations in a decentralized scheme, each GPU loads the split submatrices without taking the semantic meaning into account. The excessively large adjacency and embedding matrices are sliced into $\mathbf{A}_i$ and $\mathbf{H}_i$ $i \in [1, 4]$ and distributed to GPU_{i-1} with the full weight matrix W . Then $\mathbf{H}_1$ to $\mathbf{H}_4$ are sequentially one-to-all broadcast to all GPUs in parallel. After 4 broadcasts, the whole embedding matrix $\mathbf{H}$ is updated for a layer. Thus, by moving intact matrix blocks, Sancus takes advantage of data parallelism to avoid communication caused by intensive neighbor fetching. **Secondly,** to further **avoid communication** under a decentralized scheme, we propose to cache and skip-broadcast the historical embeddings We define historical embeddings as the embedding submatrices from earlier epochs in each distributed process, i.e., the sub-matrix Hi individually computed on GPU_{i-1} in Figure 2. We utilize caching and design a novel skip-broadcast operator to support historical embeddings in Sancus. **Thirdly,** to manage the system staleness caused by using mixed-version embeddings on each GPU, we propose the generalization of the widely-used bounded gradient staleness in centralized schemes [13], to historical embeddings. We introduce a set of novel bounded embedding staleness metrics in decentralized GNNs. Particularly, Sancus adaptively skip-broadcasts embeddings within bounds and automatically reuses cached historical embeddings to directly avoid communication; otherwise, if the embeddings become too stale, the results are broadcast and updated in cache among GPUs to keep the **system staleness** within bounds.

2 Related Works

2.1 Distributed Deep Neural Networks

There are many system-related works to optimize distributed deep neural network (DNN) training. The parameter server [50] is designed to maintain and update the sparse model parameters. Horovod [67] and Pytorch distributed [51] uses allreduce to aggregate dense model parameters but does not work for sparse model parameters. BytePs [63] adopts more sophisticated techniques of overlapping model computation and gradient communication to accelerate dense model parameter updates. Many works reduce the amount of communication by using quantization [65] or sketching [37]. Several recent work focuses on relaxing the synchronization of weights [32, 56] in case some workers run slower than others temporally due to some hardware issues. GNN models are composed of multiple operators organized into multiple graph convolution network layers shared among all nodes and edges. Thus, GNN training also has dense parameter updates. However, the network traffic generated by dense parameter updates is relatively small compared with node/edge features. Thus, reducing the network traffic of dense parameter updates is not our main focus for distributed GNN training.

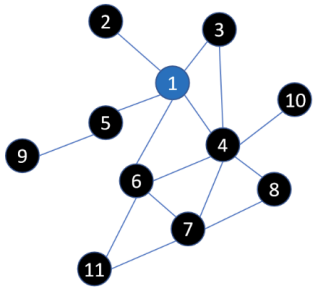
2.2 Distributed Graph Neural Networks

The distributed GNNs are still in its infancy [2], with a few prior works on GPU-based systems. Compared to distributed systems for large graph analysis [10, 18, 19, 58, 68], the communication overhead in distributed GNNs is even more challenging since the intensive data movement among workers to fetch neighbor embeddings is expensive. Currently, most existing systems utilize a centralized architecture. For example, NeuGraph [57] proposes the GNN training framework on a multi-GPU single machine with METIS [18] partitioning and specialized optimization in scheduling and pipelining. However, it is not released for public access. RoC [38] dynamically partitions the graph with an online regression method and proposes a sophisticated memory management method among workers, at the cost of complex workflow. PaGraph [53] exploits static caching of nodes with a higher degree in the GPU memory, leveraging a nontrivial partitioning algorithm to balance the workload and reduce the data movement in cross-device visits. G^3 [54] leverages parallel graph optimizations to improve graph operations in GPU systems, and Zhou et al. [85] utilize channel pruning to accelerate GNN inference, while Grain [82] focuses on GNN data selection via social influence maximization and RDD [81] uses unlabeled data. Yet, their evaluation does not focus on distributed training. AliGraph [86] utilizes static cache as well but only supports CPU servers, while AGL [80] uses MapReduce and optimizes both training and inference. To reduce and balance the communication, DistDGL [83] leverages partitioning with load balancing, while Min et al. [60] present a GPU-oriented communication reduction via zero-copy access. These specialized systems often come with heavy preprocessing and complex workflow. Moreover, such newly proposed frameworks often pose challenges in their deployment and extension. All of aforementioned distributed GNN systems adopt a centralized design, which may lead to centralized communication, high communication overhead, and a single point of failure. Also, it should be noted that only NeuGraph and Roc

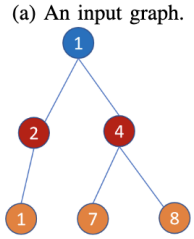
support full GNN processing, while all others need sampling. More recently, DGCL [5], a communication library for distributed full-GNN training, tries to reduce communication by finding optimal communication routes in specific system topology for every node in the entire graph. Regardless of the substantial overhead caused by planning communication for each node before every execution, it still follows the conventional message-passing paradigm for vertex-centric computation. Tripathy et al. [70] incorporate matrix blocking techniques into a set of parallel algorithms [24], to suit the sparse matrix and dense matrix operations in distributed GNNs. Though being a general implementation of distributed GNNs with high extensibility, their proposed CAGNET still struggles in scalability, owing to the communication bottleneck. Thus, we adapt the powerful parallel algorithm [71] and abstract the GNN processing as sequential matrix multiplication so that its intermediate historical embeddings are cached and re-utilized to reduce the communication overhead further in a system environment with staleness for the first time.

3 Preliminary

In this section, we review the related concepts of our target problem, and introduce the necessary equations to set the background, then formally define the problem. The key notations are listed in Table 1.



(a) An input graph.



(b) A sampled graph for computing one target vertex representation with a two-layer GNN model. Messages flow from leaves to root.

Figure 3: One sampled mini-batch in GNN training.

3.1 Graph Neural Networks

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph of order N with a set of edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ and nodes $\mathcal{V} = \{v_1, v_2, \ldots, v_N\}$. Consider the graph adjacency matrix $\mathbf{A}$, where the element A_{ij} in the matrix specifies the relation between the nodes v_i and v_j with $A_{ij} = 1$ if there exists an edge $(v_i, v_j) \in \mathcal{E}$ or otherwise $A_{ij} = 0$. $\mathbf{A}$ is

Notation	Description
$\mathbf{A}$	Adjacency matrix of the graph ($N \times N$)
$\hat{\mathbf{A}}$	Adjacency matrix after symmetric normalization ($N \times N$)
$\mathbf{W}^{(\ell)}$	Weight matrix at the ℓ^{th} layer ($F \times F$)
$\mathbf{H}^{(\ell)}$	Embedding matrix at the ℓ^{th} layer ($N \times F$)
$\mathbf{Z}^{(\ell)}$	Input matrix to activation function at the ℓ^{th} layer ($N \times F$)
$\mathbf{T}^{(\ell)}$	Intermediate result matrix of the multiplication $\hat{A}\mathbf{H}^{(\ell)}$
$\nabla_{\mathbf{Z}^{(\ell)}} \mathcal{L} = \delta^{(\ell)}$	Gradient matrix of the loss $\mathcal{L}$ with respect to $\mathbf{Z}^{(\ell)}$ at the ℓ^{th} layer
$\nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}$	Gradient matrix of the loss $\mathcal{L}$ with respect to $\mathbf{W}^{(\ell)}$ at the ℓ^{th} layer
$\mathcal{L}$	Loss of the GNN
η	Learning rate
ϵ	Staleness bound
$P(i)$	i-th process in distributed GNN

Table 1: Summary of the key symbols and notations

symmetric since $\mathcal{G}$ is undirected. Denote $\hat{\mathbf{A}} = \bar{\mathbf{D}}^{-\frac{1}{2}} \bar{\mathbf{A}} \bar{\mathbf{D}}^{-\frac{1}{2}}$ as the adjacency matrix after symmetric normalization in GCN [20], where $\bar{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$ denotes the adjacency matrix with self-connections and $\bar{\mathbf{D}} \in \mathbb{R}^{N \times N} = \mathbf{D} + \mathbf{I}_N$ denotes the diagonal node degree matrix.

Without loss of generality, the ℓ-th layer propagation process of such GNNs [2, 76] can be formulated in matrix form as:

$$\mathbf{H}^{(\ell)} = \sigma\left(\mathbf{H}^{(\ell-1)}, \hat{\mathbf{A}}; \mathbf{W}^{(\ell-1)}\right) \tag{1}$$

where σ denotes the activation function such as ReLU and $\mathbf{W} \in \mathbb{R}^{F \times F}$ denotes the weight matrix. Initially, $\mathbf{H}^{(0)} = \mathbf{X}$ where $\mathbf{X} \in \mathbb{R}^{N \times F}$ is the node embedding matrix whose i-th row represents the length- F embedding vector of node v_i. For convenience, the superscript $^{(\ell)}$ notation is omitted when it is clear from context.

Forward Propagation. Specifically, the neighbors' $(\ell-1)$-th embedding vectors are combined for each node. Based on the iterative scheme of GNNs, the computation process is given:

$$\mathbf{Z}^{(\ell)} = \hat{\mathbf{A}}\mathbf{H}^{(\ell-1)}\mathbf{W}^{(\ell-1)} \tag{2}$$

$$\mathbf{H}^{(\ell)} = \sigma\left(\mathbf{Z}^{(\ell)}\right) \tag{3}$$

Backpropagation. Firstly we derive the recurrence to backpropagate the gradient. Let $\nabla_{\mathbf{Z}^{(\ell)}} \mathcal{L}$ denote the gradient of the loss $\mathcal{L}$ with respect to $\mathbf{Z}^{(\ell)}$. To simplify, we define a factor $\delta^{(\ell)} = \nabla_{\mathbf{Z}^{(\ell)}} \mathcal{L}$. By the chain rule, the relation between $\delta^{(\ell-1)}$ and $\delta^{(\ell)}$ is:

$$\delta^{(\ell-1)} = \frac{\partial \mathcal{L}}{\partial \mathbf{Z}^{(\ell-1)}} = \delta^{(\ell)} \hat{\mathbf{A}} \left(\mathbf{W}^{(\ell-1)}\right)^{\top} \odot \sigma'\left(\mathbf{Z}^{(\ell-1)}\right), \tag{4}$$

where $\sigma'(\cdot)$ is the derivative of the activation function $\sigma(\cdot)$. Then, the gradient $\nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}$ of the loss $\mathcal{L}$ with respect to $\mathbf{W}^{(\ell)}$ is:

$$\nabla_{\mathbf{W}^{(\ell-1)}} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell-1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Z}^{(\ell)}} \frac{\partial \mathbf{Z}^{(\ell)}}{\partial \mathbf{W}^{(\ell-1)}} = \delta^{(\ell)} \hat{\mathbf{A}} \left(\mathbf{H}^{(\ell-1)}\right)^{\top} \tag{5}$$

In GNNs, one training epoch consists of a forward propagation and a backpropagation pass, with subsequent weight update:

$$\mathbf{w}^{(\ell-1)} = \mathbf{w}^{(\ell-1)} - \eta \nabla_{\mathbf{w}^{(\ell-1)}} \mathcal{L} \tag{6}$$

where η represents the learning rate. The GNN trains a number of epochs until the accuracy saturates and the model converges.

3.2 Mini-batch training

GNN models on a large dataset can be trained in a mini-batch fashion just like deep neural networks in other domains like computer vision and natural language processing. However, GNN mini-batch training is different from other neural networks due to the data dependency between vertices. Therefore, we need to carefully sample subgraphs that capture the data dependencies in the original graph to train GNN models.

A typical strategy of training a GNN model [31] follows three steps: (i) sample a set of N vertices, called *target vertices*, uniformly at random from the training set; (ii) randomly pick at most K (called *fan-out*) neighbor vertices for each target vertex; (iii) compute the target vertex representations by gathering messages from the sampled neighbors. When the GNN has multiple layers, the sampling is repeated recursively. That is, from a sampled neighbor vertex, it continues sampling its neighbors. The number of recursions is determined by the number of layers in a GNN model. This sampling strategy forms a computation graph for passing messages on. Figure 6(b) depicts such a graph for computing representation of one target vertex when the GNN has two layers. The sampled graph and together with the extracted features are called a mini-batch in GNN training.

There have been many works regarding to the different strategies to sample graphs for mini-batch training [7, 8, 11, 36, 89]. Therefore, a GNN framework needs to be flexible as well as scalable to giant graphs.

4 Propose Method

4.1 DistDGL

4.1.1 Distributed Training Architecture. DistDGLdistributes the mini-batch training process of GNN models to a cluster of machines. It follows the synchronous stochastic gradient descent (SGD) training; each machine computes model gradients with respect to its own mini-batch, synchronizes gradients with others and updates the local model replica. At a high level, DistDGLconsists of the following logical components (Figure 4):

- A number of *samplers* in charge of sampling the mini-batch graph structures from the input graph. Users invoke DistDGLsamplers in the trainer process via the same interface in DGL for neighbor sampling, which internally becomes a remote process call (RPC). After mini-batch graphs are generated, they are sent back to the trainers.
- A *KVStore* that stores all vertex data and edge data distributedly. It provides two convenient interfaces for pulling the data from or pushing the data to the distributed store. It also manages the vertex embeddings if specified by the user-defined GNN model.
- A number of *trainers* that compute the gradients of the model parameters over a mini-batch. At each iteration, they first fetch the mini-batch graphs from the samplers and the corresponding vertex/edge features from the KVStore. They then run the forward and backward computations on their own mini-batches in parallel to compute the gradients. The gradients of dense parameters are dispatched to the *dense model update component* for synchronization, while the gradients of sparse embeddings are sent back to the KVStore for update.
- A *dense model update component* for aggregating dense GNN parameters to perform synchronous SGD. DistDGLreuses the existing components depending on DGL's backend deep learning frameworks (e.g., PyTorch, MXNet, and TensorFlow). For example, DistDGLcalls the all-reduce primitive when the backend framework is PyTorch [51], or resorts to parameter servers [49] for MXNet and TensorFlow backends.

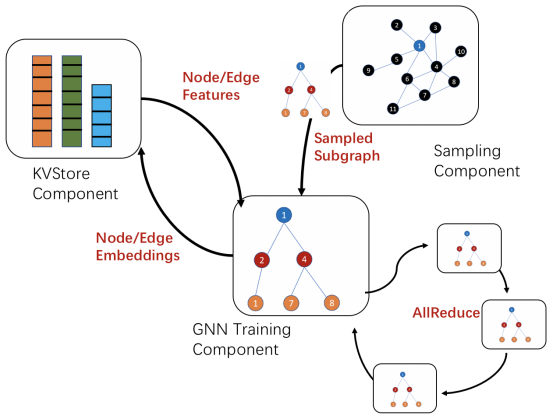


Figure 4: DistDGL's logical components.

When deploying these logical components to actual hardware, the first consideration is to reduce the network traffic among machines because graph computation is data intensive [17]. DistDGLadopts the owner-compute rule (Figure 5). The general principle is to dispatch computation to the data owner to reduce network communication. DistDGLfirst partitions the input graph with a light-weight min-cut graph partitioning algorithm. It then partitions the vertex/edge features and co-locates them with graph partitions. DistDGLlaunches the sampler and KVStore servers on each machine to serve the local partition data. Trainers also run on the same cluster of machines and each trainer is responsible for the training samples from the local partition. This design leverages data locality to its maximum. Each trainer works on samples from the local partition so the mini-batch graphs will contain mostly local vertices and edges. Most of the mini-batch features are locally available too via shared memory, reducing the network traffic significantly. In the following sections, we will elaborate more on the design of each component.

4.1.2 Graph Partitioning. The goal of graph partitioning is to split the input graph into multiple partitions with a minimal number of edges across partitions. Graph partitioning is a preprocessing step before distributed training. A graph is partitioned once and used for many distributed training runs, so its overhead is amortized.

DistDGLadopts METIS [43] to partition a graph. This algorithm assigns densely connected vertices to the same partition to reduce the number of edge cuts between partitions (Figure 6(a)). After assigning some vertices to a partition, DistDGLassigns all incident

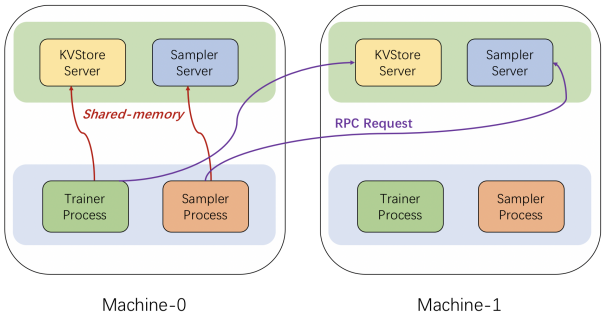


Figure 5: The deployment of DistDGL's logical components on a cluster of two machines.

edges of these vertices to the same partition. This ensures that all the neighbors of the local vertices are accessible on the partition so that samplers can compute locally without communicating with each other. With this partitioning strategy, each edge has a unique assignment while some vertices may be duplicated (Figure 6(b)). We refer to the vertices assigned by METIS to a partition as *core vertices* and the vertices duplicated by our edge assignment strategy as *HALO vertices*. All the core vertices also have unique partition assignments.

While minimizing edge cut, DistDGLdeploys multiple strategies to balance the partitions so that mini-batches of different trainers are roughly balanced. By default, METIS only balances the number of vertices in a graph. This is insufficient to generate balanced partitions for synchronous mini-batch training, which requires the same number of batches from each partition per epoch and all batches to have roughly the same size. We formulate this load balancing problem as a multi-constraint partitioning problem, which balances the partitions based on user-defined constraints [45]. DistDGLtakes advantage of the multi-constraint mechanism in METIS to balance training/validation/test vertices/edges in each partition as well as balancing the vertices of different types and the edges incident to the vertices of different types.

METIS' partitioning algorithms are based on the multilevel paradigm, which has been shown to produce high-quality partitions. However, for many types of graphs involved in learning on graphs tasks (e.g., graphs with power-law degree distribution), the successively coarser graphs become progressively denser, which considerably increases the memory and computational complexity of multilevel algorithms. To address this problem, we extended METIS to only retain a subset of the edges in each successive graph so that the degree of each coarse vertex is the average degree of its constituent vertices. This ensures that as the number of vertices in the graph reduces by approximately a factor of two, so do the edges. To ensure that the partitioning solutions obtained in the coarser graphs represent high-quality solutions in the finer graphs, we only retain the edges with the highest weights in the coarser graph. In addition, to further reduce the memory requirements, we use an out-of-core strategy for the coarser/finer graphs that are not being processed currently. Finally, we run METIS by performing a single initial partitioning (default is 5) and a single refinement iteration (default is 10) during each level. For power-law degree graphs, this optimization leads to a small increase in the edge-cut

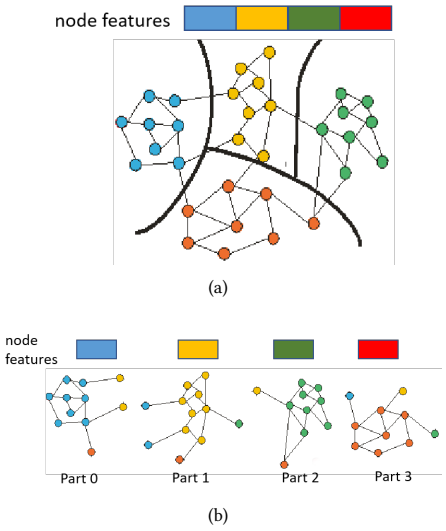


Figure 6: Graph partitioning with METIS in DistDGL. (a) Assign vertices to graph partitions. (b) Generate graph partitions with HALO vertices (the vertices with different colors from majority of the vertices in the partition).

(2%-10%) but considerably reduces its runtime. Overall, the set of optimizations above compute high-quality partitionings requiring 5× less memory and 8× less time than METIS' default algorithms.

After partitioning the graph structure, we also partition vertex features and edge features based on the graph partitions. We only assign the features of the *core vertices* and edges of a partition to the partition. Therefore, the vertex features and edge features are not duplicated.

After graph partitioning, DistDGLmanages two sets of vertex IDs and edge IDs. DistDGLexposes global vertex IDs and edge IDs for model developers to identify vertices and edges. Internally, DistDGLuses local vertex IDs and edge IDs to locate vertices and edges in a partition efficiently, which is essential to achieve high system speed as demonstrated by previous works [87]. To save memory for maintaining the mapping between global IDs and local IDs, DistDGLrelabels vertex IDs and edge IDs of the input graph during graph partitioning to ensure that all IDs of core vertices and edges in a partition fall into a contiguous ID range. In this way, mapping a global ID to a partition is a binary lookup in a very small array and mapping a global ID to a local ID is a simple subtraction operation.

4.1.3 Distributed Key-Value Store. The features of vertices and edges are partitioned and stored in multiple machines. Even though DistDGLpartitions a graph to assign densely connected vertices to a partition, we still need to read data from remote partitions. To simplify the data access on other machines, DistDGLdevelops a distributed in-memory key-value store (KVStore) to manage the vertex and edge features as well as vertex embeddings, instead of using an existing distributed in-memory KVStore, such as Reddis, for *(i)* better co-location of node/edge features in KVStore and graph

partitions, *(ii)* faster network access for high-speed network, *(iii)* efficient updates on sparse embeddings.

DistDGL's KVStore supports flexible partition policies to map data to different machines. For example, vertex data and edge data are usually partitioned and mapped to machines differently as shown in Section 4.1.2. DistDGLdefines separate partition policies for vertex data and edge data, which aligns with the graph partitions in each machine.

Because accessing vertex and edge features usually accounts for the majority of communication in GNN distributed training, it is essential to support efficient data access in KVStore. A key optimization for fast data access is to use shared memory. Due to the co-location of data and computation, most of data access to KVStore results in the KVStore server on the local machine. Instead of going through Inter-Process Communication (IPC), the KVStore server shares all data with the trainer process via shared memory. Thus, trainers can access most of the data directly without paying any overhead of communication and process/thread scheduling. We also optimize network transmission of DistDGL's KVStore for fast networks (e.g., 100Gbps network). We develop an optimized RPC framework for fast networking communication, which adopts a zero-copy mechanism for data serialization and a multi-thread send/receive interface.

In addition to storing the feature data, we design DistDGL's KVStore to support sparse embedding for training transductive models with learnable vertex embeddings. Examples are knowledge graph embedding models [84]. In GNN mini-batch training, only a small subset of vertex embeddings are involved in the computation and updated during each iteration. Although almost all deep learning frameworks have off-the-shelf sparse embedding modules, most of them lack efficient support for distributed sparse updates. DistDGL's KVStore shards the vertex embeddings in the same way as vertex features. Upon receiving the embedding gradients (via the PUSH interface), KVStore updates the embedding based on the optimizer the user registered.

4.1.4 Distributed Sampler. DGL has provided a set of flexible Python APIs to support a variety of sampling algorithms proposed in the literature. DistDGLkeeps this API design but with a different internal implementation. At the beginning of each iteration, the trainer issues sampling requests using the target vertices in the current mini-batch. The requests are dispatched to the machines according to the core vertex assignment produced by the graph partitioning algorithm. Upon receiving the request, sampler servers call DGL's sampling operators on the local partition and transmit the result back to the trainer process. Finally, the trainer collects the results and stitches them together to generate a mini-batch.

DistDGLdeploys multiple optimizations to effectively accelerate mini-batch generation. DistDGLcan create multiple sampling worker processes for each trainer to sample mini-batches in parallel. By issuing sampling requests to the sampling workers, trainers overlap the sampling cost with mini-batch training. When a sampling request goes to the local sampler server, the sampling workers to access the graph structure stored on the local sampler server directly via shared memory to avoid the cost of the RPC stack. The sampling workers also overlaps the remote RPCs with local sampling computation by first issuing remote requests asynchronously. This

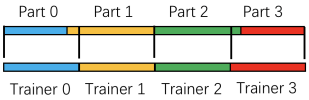


Figure 7: Split the workloads evenly to balance the computation among trainer processes.

effectively hides the network latency because the local sampling usually accounts for most of the sampling time. When a sampler server receives sampling requests, it only needs to sample vertices and edges from the local partition because our graph partitioning strategy (Section 4.1.2) guarantees that the core vertices in a partition have access to the entire neighborhood.

4.1.5 Mini-batch Trainer. Mini-batch trainers run on each machine to jointly estimate gradients and update the parameters of users' models. DistDGLprovides utility functions to split the training set distributedly and generate balanced workloads between trainers.

Each trainer samples data points uniformly at random to generate mini-batches independently. Because DistDGL generates balanced partitions (each partition has roughly the same number of nodes and edges) and uses synchronous SGD to train the model, the data points sampled collectively by all trainers in each iteration are still sampled uniformly at random across the entire dataset. As such, distributed training in DistDGL in theory does not affect the convergence rate or the model accuracy.

To balance the computation in each trainer, DistDGLuses a two-level strategy to split the training set evenly across all trainers at the beginning of distributed training. We first ensure that each trainer has the same number of training samples. The multi-constraint algorithm in METIS (Section 4.1.2) can only assign roughly the same number of training samples (vertices or edges) to each partition (as shown by the rectangular boxes on the top in Figure 7). We thus evenly split the training samples based on their IDs and assign the ID range to a machine whose graph partition has the largest overlap with the ID range. This is possible because we relabel vertex and edge IDs during graph partitioning and the vertices and edges in a partition have a contiguous ID range. There is a small misalignment between the training samples assigned to a trainer and the ones that reside in a partition. Essentially, we make a tradeoff between load balancing and data locality. In practice, as long as the graph partition algorithm balances the number of training samples between partitions, the tradeoff is negligible. If there are multiple trainers on one partition, we further split the local training vertices evenly and assign them to the trainers in the local machine. We find that random split in practice gives a fairly balanced workload assignment.

In terms of parameter synchronization, we use synchronous SGD to update dense model parameters. Synchronous SGD is commonly used to train deep neural network models and usually leads to better model accuracy. We use asynchronous SGD to update the sparse vertex embeddings in the Hogwild fashion [61] to overlap communication and computation. In a large graph, there are many vertex embeddings. Asynchronous SGD updates some of the embeddings in a mini-batch. Concurrent updates from multiple trainers rarely result in conflicts because mini-batches from different trainers run

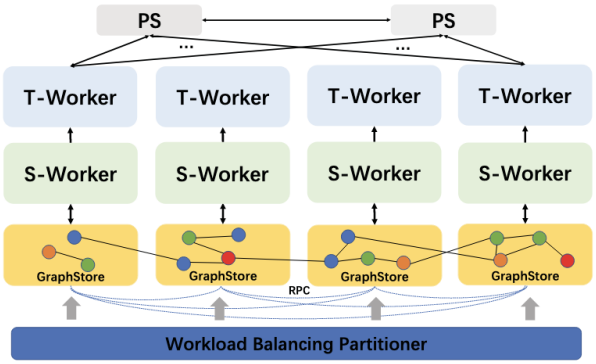


Figure 8: System architecture of ByteGNN

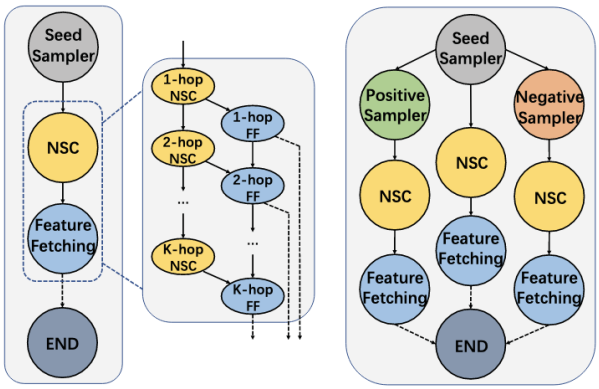


Figure 9: The DAG of the sampling workflow

on different embeddings. Previous study [84] has verified that asynchronous update of sparse embeddings can significantly speed up the training with nearly no accuracy loss.

For distributed CPU training, DistDGLparallelizes the computation with both multiprocessing and multithreading. Inside a trainer process, we use OpenMP to parallelize the framework operator computation (e.g., sparse matrix multiplication and dense matrix multiplication). We run multiple trainer processes on each machine to parallelize the computation for non-uniform memory architecture (NUMA), which is a typical architecture for large CPU machines. This hybrid approach is potentially more advantageous than the multiprocessing approach for synchronous SGD because we need to aggregate gradients of model parameters from all trainer processes and broadcast new model parameters to all trainers. More trainer processes result in more communication overhead for model parameter updates.

4.2 ByteGNN

Figure 8 shows the architecture of ByteGNN, which consists of four main components in each machine where ByteGNNis deployed. **Graph Store** stores a partition of the input graph data and the Graph Stores of all machines form a distributed Graph Store. **PS** is a parameter server that stores the model parameters. **Sampling Worker (S-Worker)**, handles the sampling phase and constructs sampled neighborhood subgraphs for sampled seed vertices. **Training Worker (T-Worker)**, handles the training phase, which computes model gradients on the sampled neighborhood subgraphs constructed by the S-Worker in the same machine and synchronizes the gradients with PS to update the model parameters.

4.2.1 Abstraction of Mini-Batch Graph Sampling. The sampling process in existing GNN systems [21, 83, 86] is not well-organized as the tasks in each sampling phase are executed without overlapping, which often leads to CPU under-utilization. In addition, sampling is conducted for one iteration (i.e., one minibatch) after another, even though different mini-batches are independent of each other. To support parallel sampling within a mini-batch and among mini-batches, so as to maximize CPU utilization, we model the sampling process as a DAG of tasks. Then, we can execute the DAGs of sampling multiple mini-batches in parallel. We also introduce a scheduler to effectively utilize the computing resources for both intra-DAG and inter-DAG parallelization, while balancing the loads between sampling and training so that one is not waiting for the other to finish in order to continue.

To construct this DAG for a general GNN model, we analyzed the sampling phase of a broad range of existing GNN models, which cover most of the widely-adopted models such as GCN [46], GAT[73], GraphSAGE[31], PinSAGE[78], and GraphSAINT[79]. We provide a common abstraction for the sampling phase of these GNN models with a set of five operators: (1) **Seed Sampler**: sampling a set of vertices as seeds from the local graph store; (2) **Positive Sampler**: sampling vertices from the direct neighbors of each seed; (3) **Negative Sampler**: sampling vertices from those that are not the direct neighbors of each seed; (4) **Neighborhood Subgraph Construction (NSC)**: sampling vertices from the multi-hop neighborhood of a given vertex and constructing the sampled neighborhood subgraph; (5) **Feature Fetching**: fetching the attributes of a given vertex/edge to construct its feature vector.

With the above five operators, we can present the workflow of the sampling phase as a DAG, as shown in Figure 9. The DAG on the left of Figure 9 models supervised training, which consists of three tasks: Seed Sampling, NSC, and Feature Fetching. For unsupervised training, we also need to construct the neighborhood subgraphs of each positively and negatively sampled vertices of the seed vertices, as shown in the DAG on the right of Figure 9. The three branches in the DAG for unsupervised training can be executed in parallel, and the results are then collected in the "End" node to be fed into a T-Worker for training.

To enable higher parallelism for both supervised and unsupervised training, we create an instance of the two dominating operations (i.e., NSC and Feature Fetching, as they access multi-hop neighbors and their attributes) for each sampled vertex and execute these instances in parallel. In addition, as NSC (along with Feature Fetching) is executed repeatedly for each hop of neighborhood expansion, we can break the multi-hop operations into many smaller tasks of one-hop operations. As shown in Figure 9, each small task of Feature Fetching can start immediately when the corresponding small NSC task finishes. The more fine-grained task abstraction

results in higher parallelism and better resource utilization (e.g., less head-of-line blocking and stragglers, less fragmentation in resource utilization).

To construct a DAG, users only need to specify the customized sampling functions in Seed Sampler, Positive Sampler, Negative Sampler, and also in NSC (e.g., how and how many neighbors in each hop should be sampled). This design also leaves space for researchers and engineers to explore new, high-quality sampling strategies using the framework. Note that the logical DAG is created only once and physical instances are generated and executed for each mini-batch by the S-Workers.

4.2.2 Two-Level Scheduling. ByteGNNadopts a two-level scheduling strategy to improve CPU utilization and reduce the end-to-end GNN training time. Although many scheduling strategies have been proposed, they are mostly for job scheduling at the cluster level [14–16, 25, 30, 64] or heterogeneous jobs/tasks in dataflow systems [42], which are over-complicated and incur extra overheads for scheduling the simple tasks in our system (note that for the training of a GNN model, we only need to schedule instances of the same DAG instead of many different DAGs).

Coarse-grained scheduling. The S-Worker in each machine executes multiple DAGs in parallel to increase throughput and reduce the end-to-end GNN training time. The first question we need to answer is how many DAGs should be launched in a machine. If we launch too many DAGs, which means more resource is needed by sampling, then resource contention becomes a problem. Resource contention does not just occur among the DAGs, but also between sampling (i.e., DAG execution) and training (i.e., model computation). The training time increases significantly when too many DAGs are launched. On the other hand, if too few DAGs are running, the resource is under-utilized. The training phase finishes quickly and the next iteration's training waits for the neighborhood subgraphs to be produced by the DAGs.

To control resource utilization, we need to decide when to launch a DAG. We can model this problem as a variation of the classical Job-Shop Scheduling Problem (JSP) [4]. Each DAG can be regarded as a job, where a set of operations (tasks) in each job need to be processed in a specific order, and we have a set of jobs that are to be processed on a given set of workers. Knowing the best timing for DAG launching is equivalent to getting the earliest starting time of each job in the solution to this special Job-Shop Scheduling Problem. The Job-Shop Scheduling Problem has been well studied and finding a schedule that minimizes the makespan or minimizes the sum of the job completion time was

proved to be strongly NP-hard [6]. Some new research also shows that the currently best approximation algorithms have worse than logarithmic performance guarantee [26].

We propose a heuristic strategy to decide when to launch a DAG based on three runtime measures: C_{util}, Q_{size}, and T_{gap}.
C_{util} is the CPU utilization rate. If C_{util} is low, we may launch a new DAG; otherwise, we may wait until C_{util} drops to a suitable level. Note that high C_{util} does not necessarily result in better performance because there could be much contention and switching among DAGs and between sampling and training.

In addition to CPU utilization, We also need to consider the memory footprint. The neighborhood subgraph constructed from each

Algorithm 1 The Coarse-Grained Scheduling Strategy

Variable: $C_{util}, Q_{size}, T_{gap}$
Given: σ = *launch-score*
while *more_dag* **do**
  //*more_dag*=1 when more DAGs can be launched
  $balance = \frac{T_{avg\_sample}}{T_{avg\_train} * Q_{size}}$;
  $f(C_{util}) = (101 - \mathbf{e}^{C_{util}/\mathbf{c}})$, where $\mathbf{c} = \frac{100}{\ln 101}$;
  $launch\_score = T_{gap} * f(C_{util}) * balance$;
  if $launch\_score \geq \sigma$ **then**
    more_dag = *Launch_DAG()*;
    // launches a new DAG; returns 0 when no more DAG to launch;
    launch
    $T_{last\_launch} = Time()$ // used to calculate T_{gap};
  else
    sleep(5ms);
  end if
end while

DAG execution is kept in the DAG output queue in the S-Worker and Q_{size} is the size of this queue. The neighborhood subgraphs are then consumed by the T-Worker for training. Thus, Q_{size} is essentially an indicator of the speed of production (by the S-Worker) and the speed of consumption (by the T-Worker) of the neighborhood subgraphs. If Q_{size} is small, we may launch new DAGs; otherwise, we pause the launching. If Q_{size} is large, it implies an over-supply of neighborhood subgraphs and we may shift more computing resources from sampling to accelerate training. Thus, Q_{size} not only controls the memory usage but also balances the overall resource usage between sampling and training.

We also found that the real-time measure for C_{util} is not sensitive enough since newly launched DAGs may not change the CPU utilization in a short time period and many DAGs may be launched during the period. Later, when the tasks in these DAGs start to run in parallel and use up the computing resources, the system suffers from severe resource contention. To avoid such delayed performance punishments, we introduce T_{gap}, which is the time gap elapsed since the previous DAG launch. If T_{gap} is too small, we may want to wait for a bit longer before we launch a new DAG.

It would be undesirable if users need to set the thresholds for the three measures, as it is hard to determine what values of C_{util}, Q_{size}, and T_{gap} are good and how to relate them to each other. To this end, we integrate them into one single score, launch-score, to decide whether we should launch a new DAG. The idea is to maintain the balance between the production speed and the consumption speed of neighborhood subgraphs while keeping CPU utilization high. Ideally, we hope that the output of each DAG will be consumed immediately by the training phase, which means that Q_{size} should be close to 0 all the time. However, in most cases a very low Q_{size} happens with a very low C_{util}. Thus, we need to consider Q_{size} together with C_{util}.

Algorithm 1 shows the algorithm for coarse-grained scheduling. First, we want to maintain $balance = \frac{T_{avg\_sample}}{T_{avg\_train} * Q_{size}}$, where T_{avg_sample} and T_{avg_train} are the average time for sampling and training a mini-batch. If $balance > 1$, it means that it would take less

Algorithm 2 Block Assignment

Input: List of Blocks $\mathbf{B} = B_1, B_2, \ldots, B_n$
Output: Graph partitions $P_1, P_2, P_3, \ldots, P_k$
for *each block* B_i *in* **B do**
  for $j \leftarrow 1$ *to* k **do**
    CE[j] = |Cross_Edge(P_j, B_i)|/|P_j|
    BD[j] = $1 - \alpha * \frac{|P_j(train)|}{C(train)} - \beta * \frac{|P_j(val)|}{C(val} - \gamma * \frac{|P_j(test)|}{C(test)}$
  end for
  x = **argmax**$_{1 \le t \le k}${CE[t]*BS[t]}
  $P_x = P_x \cup B_i$
end for
return $P_1, P_2, P_3, \ldots, P_k$

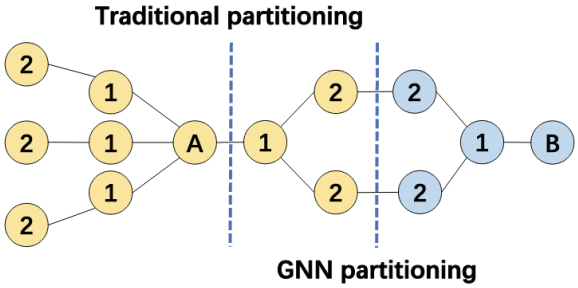


Figure 10: Traditional partitioning vs. GNN partitioning

time to consume the current Q_{size} sampling results than to produce a new sampling result, which is an indicator that a new DAG should be launched. Next, we first attempt to use ($100 - C_{util}$) to give a higher weight to balance if C_{util} is low and penalize balance (i.e., delay new DAG launching) when C_{util} is high. However, simply using ($(100 - C_{util})$ does not work well as it is a linear scale. Instead, we want to quickly increase CPU utilization when C_{util} is low and prevent contention promptly when C_{util} is already very high. Thus, we use an exponential function, $f(C_{util}) = 101 - e^{C_{util}/c}$, where $c = \frac{100}{\ln 101}$ is a constant used to align the range of $f(C_{util})$ with that of C_{util}, i.e., $f(0) = 100, f(100) = 0$, and $0 \le f(C_{util}) \le 100$. Finally, we also put T_{gap} as a weight to reflect the delay in the real-time measurement of C_{util}, which leads to the definition of $launch - score$ in Algorithm 1.

4.2.3 GNN-based Graph Partitioning. We monitor launch-score in real-time and launch a new DAG when $launch - score \ge \sigma$, where σ is a threshold set as follows. As shown in Algorithm 1 and explained above, $launch - score$ connects $balance$, $f(C_{util})$ and T_{gap} together to determine whether a new DAG job should be launched. In practice, there exist reasonable values of balance, $f(C_{util})$ and T_{gap} for which a new DAG should be launched; Note that there are always trade-offs between $balance$ and $f(C_{util})$, e.g., a higher $balance$ and a lower $f(C_{util})$, to achieve a high $launch - score$. Such tradeoffs in runtime allow the system to automatically adjust the resource allocation to balance the sampling and training progress.

Fine-grained scheduling. After new DAGs are launched, the SWorker executes the tasks in the DAGs, in parallel with the tasks in other DAGs. These tasks are put in a queue when their dependency is cleared (i.e., their parent tasks in the DAG are completed) and are handled by a pool of processing threads. If we execute the tasks in a FIFO order, some tasks of newly launched DAGs could be in front of the tasks in those almost-finished DAGs. For example, when the DAG_1 pushes the "END" node in the task queue and there are already "NSC" tasks from DAG_2 and DAG_3 in the queue, the "NSC" tasks will be executed first and the "END" task will be processed later even although the "END" task is the last task in DAG_1, completing which will immediately return the sampled data to the T-Worker for training. Meanwhile, one task may unlock a lot of downstream tasks in the same DAG, and heavy tasks may block many light tasks. Thus, the average completion time of the DAG jobs and hence the end-to-end GNN training time can be significantly increased.

We schedule tasks according to the following orders: (1) tasks in a DAG with a smaller ID will be executed first; (2) tasks in the same DAG will be executed in ascending order of their costs. We assign a smaller ID to a DAG launched earlier to prioritize earlier DAGs to be completed first. We calculate the cost of a task by the data it needs to handle. For example, for sampling tasks in each hop of NSC, the cost is equal to the total number of neighbors of the input vertices; for Feature Fetching, the cost is the number of vertices/edges to be fetched multiplied by the vertex/edge feature dimension. As tasks may require data from remote machines, the S-Worker sends data-fetching requests to the local Graph Store, which communicates with remote Graph Stores to fetch the data. The remote requests are also scheduled in a similar way and the network operations are processed concurrently with the CPU operations.

Existing graph partitioning algorithms [40, 44, 53] are mainly designed to reduce inter-partition edges and balance the workload. They have been widely adopted in distributed graph processing systems [28, 62, 88] to reduce inter-machine communication. However, sample-based GNN training focuses on the K-hop neighborhood of only the vertices in the *training*, *validation* and *test* sets (instead of all vertices). For example, in Figure 10, traditional partitioning strategies cut the graph into two parts by the left dotted line since it not only balances the vertices but also has the least cut edge. But for a 2-layer GNN training, since vertex A and vertex B are the labeled vertices, partitioning by the right dotted line is actually a better choice. Even if this results in two cut edges, it would not cause any data movement in the training process as only the 2-hop neighbors of the labeled vertices are required.

In addition, the ratio of the sizes of the training, validation, and test sets of different real-world graphs may differ significantly. For example, in the Ogbn-Product dataset, the test set size is 11 times the training set size and 56 times the validation set size; while in the Ogbn-Papers dataset, the test set size is only 0.18 times the training set size and 1.7 times the validation set size. Thus, the partitioning algorithm should consider both the special data access pattern of k-layer GNN training and the balanced distribution of the training, validation, and test sets.

It is known that the traditional graph partitioning problem is proved to be APX-hard [3]. Thus, our graph partitioning problem is also APX-hard as it can be reduced to the traditional graph partitioning problem. We propose a heuristic two-step graph partitioning

strategy tailored for GNN sampling workloads. The main idea is to group vertices into multi-hop neighborhood-based blocks and then assign these blocks to partitions by balancing the numbers of *training*, *validation*, and *test* vertices in the partitions.

Step (1) neighborhood block construction. To better preserve the locality of graph data for GNN sampling workloads, we construct a neighborhood block for each vertex in the training, validation, and test sets. We start a K-hop breadth-first search from each vertex v (v is called the block center) and broadcast the unique block ID of v to its K-hop neighbors being visited. Every vertex only keeps the first block ID it receives, except for block centers which keep their own block ID. A block is then formed of all the vertices that keep the same block ID. Figure 10 demonstrates how to construct the blocks.

Step (2) block assignment. Just as existing graph partitioning algorithms aim to balance the number of vertices in the partitions, our objective is to also balance the number of training, validation, and test vertices in the partitions so that the work of training, validation, and test is also balanced among the machines. Algorithm 2 shows how to assign the blocks. For each block B_i, it is assigned to the partition with the highest score. P_j is the set of vertices that have already been assigned to partition j. $CE[j]$ is the number of cross-edges between B_i and P_j, which will be eliminated if B_i is assigned to P_j. Thus, the larger $CE[j]$ is, the more likely B_i is assigned to P_j. As the size of different partitions may vary during the assignment, we normalize $CE[j]$ by $|P_j|$. $BS[j]$ is the balancing score that controls the number of training/validation/test vertices in partition j to be close to the average value. For example, the expected number of training vertices in each partition is C (train) =| V(train) | /N, where V(train) is the set of all training vertices and N is the total number of partitions. Let $P_j(train)$ be the set of training vertices currently in partition j. Thus, a smaller $\frac{|P_j(train)|}{C(train}$) means that more training vertices can be assigned to partition j. The above applies to the validation and test vertices as well. In addition, we also use a weight to put more attention on a specific type of vertices according to the scale of that type in order to obtain a better overall performance. For example, if the number of training vertices is significantly more, we may set a larger α to favor the training process, which can improve the end-to-end processing time.

Before the block assignment, we sort the blocks in descending order of $\max\{|V(\text{train})|, |V(\text{val})|, |V(\text{test})|\}$. Then, we start the block assignment according to this order. In this way, larger blocks are assigned to different partitions first, so that smaller blocks may be used later to fill the partitions more easily when the partitions begin to fill up.

4.3 SANCUS

First, we overview SANCUS step by step in Section 4.3.1. Algorithm 3 introduces the complete staleness-aware communication-avoiding decentralized full-GNN training algorithm. To further elaborate on avoiding communication, we propose historical embeddings and skip-broadcast accordingly in Section 4.3.3 and Section 4.3.4. To manage system staleness caused by historical embeddings, we propose a set of novel metrics on bounded embedding staleness in Section 4.3.5.

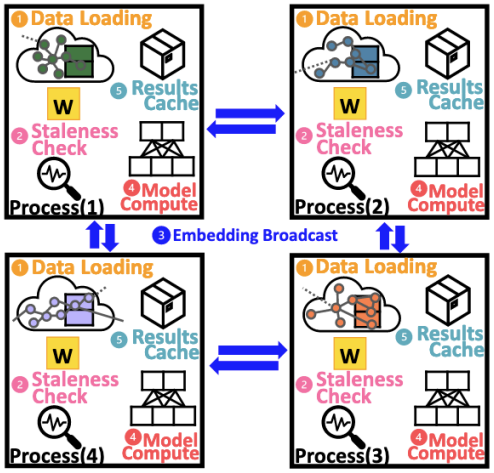


Figure 11: The overall architecture. The full graph and its node features are distributed to each process on the workers. SANCUS has five major steps: (1) Data Loading: load the split graph, embedding matrix blocks, and full model into each GPU; (2) Staleness Check: check if the embeddings become too stale for the root GPU; (3) Embedding Broadcast: if the embeddings is too stale, broadcast the up-to-date results among GPUs, otherwise other GPUs reuse the cached historical results from current root GPU; (4) Model Computation: compute the model either based on the latest or cached stale results; (5) Results Cache: update cache accordingly.

4.3.1 Overview. In this work, we propose SANCUS, an adaptive staleness-aware communication-avoiding decentralized GNN system. Fundamentally, SANCUS is simple yet effective which caches and reuses the stale historical embeddings and skips broadcast accordingly during the decentralized GNN training, based on a general communication-avoiding matrix blocking algorithm for parallel computing.

We provide the overview of SANCUS in Figure 11. Primarily, there are five steps: (1) data loading, (2) staleness bound checking, (3) embedding broadcasting, (4) GNN model computing, and (5) results caching. Here, we briefly clarify these steps: (1) to begin with, the whole sparse adjacency matrix of the full graph and the dense embedding matrix are split into matrix blocks, then loaded to individual workers. Each worker keeps its own replica of the full model; (2) on each GPU, before broadcasting the last computing results, we check whether the staleness of historical embeddings is within proposed bounds. If the staleness is within bounds, the embedding broadcast is skipped and the cached historical embeddings are reused for this iteration's model computing; (3) otherwise, if the staleness exceeds the limit, the latest results are broadcast to all workers and updated in cache; (4) thus either latest embeddings or cached historical embeddings are loaded to the GNN model to compute; (5) Finally, updated embeddings are dispatched to next iteration's staleness check before the broadcast.

Algorithm 3 The decentralized stale parallel GNN training algorithm based on arbitrary general block row decomposition preprocessing strategy with a foward pass procedure to compute $\mathbf{Z}$ in Equation (2) and $\mathbf{H}$ in Equation (3), a backpropagation procedure to compute the gradients δ in Equation (4) and $\nabla_{\mathrm{W}}\mathcal{L}$ in Equation (5), and the final weight update in Equation (6). The matrices $\hat{A}$ and H are distributed on a $p \times p$ process grid, where each process $P(i)$ receives N/p consecutive block rows.

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$; Sparse adjacency matrix $\hat{\mathbf{A}}$; Dense feature matrix $\mathbf{H}^{(0)}$; Dense weight matrix $\mathbf{W}$;
Output: Node embedding matrix $\mathbf{H}^{(L)}$;
Preprocessing: Block row partition
for all process $P(i)$ **in parallel do**
  procedure FORWARD PASS
    for $\ell = 1, \ldots, L$ **do**
      for $j = 1$ **to** p **do**
        if F(j)==ACTIVE **then**
          BROADCAST($\mathbf{H}^{(\ell-1)_j}$)
          CACHE($\mathbf{H}^{(\ell-1)_j}$)
          $\mathrm{T}_i^{(\ell-1)} \leftarrow \mathrm{T}_i^{(\ell-1)} + \hat{\mathbf{A}}_{ij}\mathbf{H}_j^{(\ell-1)}$
        else
          $\mathrm{T}_i^{(\ell-1)} \leftarrow \mathrm{T}_i^{(\ell-1)} + \hat{\mathbf{A}}_{ij}\tilde{\mathbf{H}}_j^{(\ell-1)}$
        end if
      end for
      $\mathbf{Z}_i^{(\ell)} \leftarrow \mathrm{T}_i^{(\ell-1)}\mathbf{W}^{(\ell-1)}$
      $\mathrm{H}_i^{(\ell)} \leftarrow \sigma\left(\mathbf{Z}_i^{(\ell)}\right)$
      F(i) $\leftarrow$ STALE $\left(\mathbf{H}_i^{(\ell)}\right)$
    end for
  end procedure
  procedure BACKWARD PASS
    for $\ell = L-1, \cdots, 0$ **do**
      $\delta_i^{(\ell)} \leftarrow$ GRADIENT_ CLIP $\left(\delta_i^{(\ell)}\right)$
      BROADCAST $\left(\delta_i^{(\ell)}\right)$ and Update weights W^{ℓ} by gradients
    end for
  end procedure
end for

4.3.2 Staleness-Aware Communication-Avoiding Decentralized Training. First, we present the comprehensive staleness-aware communication avoiding decentralized full-graph GNN Training in Algorithm 3 and elaborate on its details. There are three keys: **(1)** Worker state flag F(i) is equipped to indicate the worker state. The state is recorded as either ACTIVE or STALE to support the Skip-Broadcast operation in Section 4.3.4; **(2)** cache is utilized to store the historical embeddings from other workers that can be repeatedly utilized for future iterations to avoid communication; **(3)** bounded embedding staleness is tolerated to manage system staleness, where each worker may use embeddings from different iterations.

For the preprocessing, SANCUS supports any partitioning algorithms that split the graph and feature matrices into matrix blocks, such as the classical METIS [43] adopted in most existing distributed systems including DistDGL [83] and AliGraph [86], or efficient random partitioning. As shown in Figure 12, SANCUS treats GNN processing purely as sequential matrix multiplication operations to avoid intensive neighbor fetching during GNN aggregation. To start with, the sparse adjacency matrix $\hat{A}$ and the dense embedding matrix H are distributed to each process $P(i)$ where $i \in [1, p]$ on workers. To further illustrate, recall that N denotes the node number and F denotes the feature embedding length, then the $(N \times N)$ matrix $\hat{\mathbf{A}}$ is computed with p row partitions and p column partitions while the $(N \times F)$ matrix H is computed with p row partitions as shown in Figure 12. The $(F \times F)$ dense weight matrix W, however, is fully replicated throughout every process $P(i)$. Additionally, we define the intermediate results $\mathrm{T}_i^{(\ell)}$ of the matrix multiplication $\hat{A}H^{(\ell)}$ as $\mathbf{T}_i^{(\ell)} = \sum_{j=1}^{p} \hat{\mathbf{A}}_{ij}\mathbf{H}_j^{(\ell)}$ for each process $P(i)$.

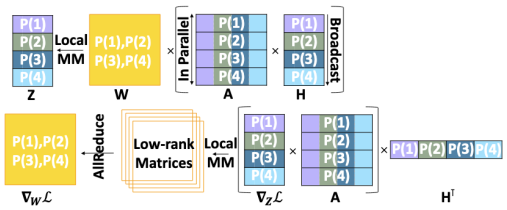


Figure 12: Communication-Avoiding Data-Parallel GNN training, from a sequential matrix multiplication processing perspective.

For each distributed process $P(i)$ in parallel, SANCUS proceeds the forward pass and backpropagation with the help of collective operations such as ring-based pipelined Broadcast and AllReduce. At the beginning, the staleness of the intermediate embedding results $\mathbf{H}_j^{(\ell-1)}$ of process j is checked in Line 6 before broadcasting to other workers. If the process state is ACTIVE, then $\mathrm{H}_j^{(\ell-1)}$ is sent to all workers in Line 7 from the root rank and copied to all ranks via a One-to-All Broadcast sequentially and cached accordingly. Otherwise, if the process state is STALE, SANCUS performs Skip-Broadcast to swap out the process j from the communication topology but leave it in the broadcast graph so that the worker j can still receive updates. The process j stops broadcasting out $\mathbf{H}_j^{(\ell-1)}$ for this round, so all other workers repeatedly use their cached version of stale embeddings $\tilde{\mathbf{H}}_j^{(\ell-1)}$. Thus, either the up-to-date results from the last epoch are used in Line 10 or the cached stale results from earlier epochs are automatically repeated for local computation in Line 13. Next, the intermediate results $\mathrm{T}_i^{(\ell-1)}$ are used to compute the embeddings $\mathbf{H}_i^{(\ell)}$ for each process $P(i)$. After computing the latest embeddings $\mathbf{H}_i^{(\ell)}$ for process i, SANCUS checks whether $\mathbf{H}_i^{(\ell)}$ is within the staleness bound so that the worker state flag F (i) remains STALE to keep Skip-Broadcast $\mathbf{H}_i^{(\ell)}$ or becomes ACTIVE to send out $\mathbf{H}_i^{(\ell)}$. Note that for staleness checking, SANCUS either inspect staleness defined in Definition 1, Definition 2, or Definition 3, respectively.

In the backward pass, gradients δ_j are broadcast similarly. To reduce communication of gradient sending, gradient clipping is performed locally, also as a regularizer of historical embeddings.

Conventionally, gradients are clipped whenever the L2-norm $\|\cdot\|_2$ exceeds a threshold th. With decentralized P workers, denote the local threshold as th_i. Assume i.i.d. gradients on each worker with variance σ^2, then the sum of gradients on all workers has variance $P\sigma^2$. Hence, $\mathbb{E}\,\|\delta_i\|_2 \approx \sigma$ and $\mathbb{E}\|\delta\|_2 \approx P^{\frac{1}{2}}\sigma$. It follows that the local gradient threshold is scaled by $P^{\frac{1}{2}}$, that is, $th_i = P^{\frac{1}{2}} th$. Finally to update the model, an AllReduce operation is performed to combine gradients from all workers and send them to all ranks.

Since the GNN processing is treated purely as sequential matrix multiplication operations as shown in Figure 12, matrix blocks are directly moved among decentralized workers. Thus, SANCUS avoids the irregular and complex request-send communication to fetch neighbors in vertex-centric distributed GNNs. To further avoid communication, we define historical embeddings in Section 4.3.3 so that SANCUS can cache and reuse historical embeddings.

4.3.3 Historical Embeddings. With P decentralized workers, the embedding matrix H split by rows is denoted as $\mathbf{H}_i$ where $i \in [1, p]$ and is distributed to each $P(i)$ process. To compute the embedding $\mathbf{H}^{(\ell)}$ in a general GNN in Equation (1), we need to combine the matrix block $\mathbf{H}_i$ on each GPU. Inspired by historical embeddings $\tilde{\mathbf{h}}^{(\ell)}$ [9, 22], we generalize the idea of historical embeddings as stale intermediate embedding results computed by other workers in distributed GNNs. Thus, the embedding matrix $\mathbf{H}^{(\ell)}$ in Equation (2) consists of two parts the latest embedding submatrices from active workers which just broadcast the results and the historical embedding submatrices from stale workers whose embedding variation is small enough to be neglected. The historical embeddings are stored in the cache on each GPU, only preserving the fresh ones. Let [] denotes the vertical concatenation of matrix blocks. The processor state ACTIVE and STALE denote whether the processor $P(i)$ is active to broadcast the latest result of embedding submatrix $\mathbf{H}_i^{(\ell)}$ or stale so the history $\tilde{\mathbf{H}}_i^{(\ell)}$ is used:

$$\begin{aligned}\mathbf{H}^{(\ell)} &= \sigma\left(\left[\mathbf{H}_i^{(\ell-1)}\right]_{i=1}^{P}, \hat{\mathbf{A}}; \mathbf{W}^{(\ell-1)}\right) \\ &\approx \sigma\left(\left[\mathbf{H}^{(\ell-1)}_{i:P(i)\Leftarrow\text{ACTIVE}} \mid \tilde{\mathbf{H}}^{(\ell-1)}_{i:P(i)\Leftarrow\text{STALE}}\right]_{i=1}^{P}, \hat{\mathbf{A}}; \mathbf{W}^{(\ell-1)}\right).\end{aligned} \tag{7}$$

4.3.4 Skip-Broadcast. With the decentralized scheme, the question is how we can adjust the communication operation such as one-to-all broadcast to support historical embeddings with bounded staleness. Since most implementations of such decentralized scheme [70] are based on bulk synchronism, it is challenging to directly enforce historical embeddings. Thus, we propose a communication primitive that is efficient to implement and requires no centralized parameter servers. Particularly, a Skip-Broadcast scheme is designed, allowing seamless reshaping of the communication topology during training. To realize Skip-Broadcast, SANCUS keeps the state flag Flag(i) on each worker i to indicate the corresponding worker status for the embeddings $\mathbf{H_i}$ computed on that worker, where $i \in [1, p]$. Specifically, Flag(i) == ACTIVE means worker i needs to broadcast its latest version of embeddings $\mathbf{H_i}$. During the broadcast, the latest embeddings $\mathbf{H_i}$ should be sent to all other workers and cached there respectively. If Flag(i) turns to STALE, SANCUS can Skip-Broadcast H_i and let other workers utilize their cached stale embeddings.

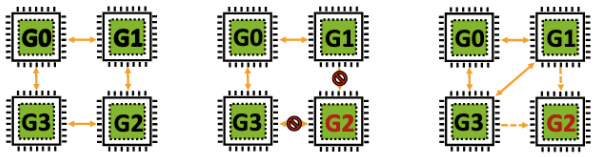


Figure 13: Skip-Broadcast Example, with STALE GPU worker marked red.

Take Figure 13 as an example, GPU 2 is notified with Flag(2) with STALE state so that other workers rely on their cached stale version of the historical embeddings H_2. Then the ring-based communication topology is reshaped seamlessly to skip GPU 2 and connect its neighbors directly. To receive updated embeddings and gradients from other workers, it should be pointed out that GPU 2 is still preserved in the graph. Therefore, the Skip-Broadcast is performed in replacement of the original broadcast operation whenever the portion of embeddings computed by the corresponding worker is within the bounded staleness. By bypassing the STALE worker to broadcast its stale embeddings, SANCUS further reduces the communication overhead. The stale flag Flag(i) is checked in every iteration and updated if needed to help reshape the ring-based communication topology.

4.3.5 Bounded Embedding Staleness. With skip-broadcast to support embedding staleness, workers can result with embeddings of different iterations from others. To manage system staleness with such mixed-version issues, SANCUS supports bounded embedding staleness. Though bounded gradient staleness is deeply investigated [13, 41, 77] in traditional distributed ML for stochastic gradient descent (SGD), its main purpose is to help SGD converge, mitigating negative effects from stale gradients. However, we actively utilize stale embeddings to avoid communication. By introducing a set of novel bounded embedding staleness metrics ϵ, we can control the errors caused by stale embeddings.

Bounded Staleness Definition Variants. We firstly provide the definitions of three staleness of historical embeddings, including one measured by the variation gap of embeddings. The first epoch-fixed embedding staleness of one local update on each processor $P(i)$ is formalized as follows:

Definition 1 (Epoch-Fixed Embedding Staleness). *For all processors in the decentralized GNNs, let $\tilde{e}$ and e denote the epoch number of the intermediate stale embeddings and the current epoch respectively, the maximum number ϵ_E of stale epochs that can be tolerated is defined as the fixed-epoch embedding staleness: $|\tilde{e} - e| \leq \epsilon_E$, where the intermediate embeddings during model computation are only broadcast after every ϵ_E epochs.*

However, all workers can still be regarded as working in a fully synchronous fashion. Thus, we propose two more flexible metrics for the decentralized scheme, so workers may rely on stale embeddings of adaptive iterations from others. Thus, the system can better manage its staleness where workers work at different speeds.

Definition 2 (Epoch-Adaptive Embedding Staleness). *For each processor $P(i)$ in the decentralized GNNs, let the maximum epoch gap ϵ_A of embeddings between $P(i)$ and all its in-coming neighbor processors be the epoch-adaptive embedding staleness. $P(i)$ must broadcast*

its latest results to others after receiving stale embeddings from all its in-coming neighbors at most ϵ_A epochs ago.

Definition 3 (Epoch-Adaptive Variation-Gap Embedding Staleness). *For each processor $P(i)$ in the decentralized GNNs, let the maximum variation gap ϵ_H in the values of the stale embeddings that can be tolerated be defined as the epoch-adaptive variation-gap embedding staleness: $\left\|\tilde{\mathbf{H}}_i^{(\ell)} - \mathbf{H}_i^{(\ell)}\right\| \leq \epsilon_H$, where the intermediate embeddings are adaptively broadcast whenever the embedding variation gap exceeds ϵ_H regardless of the number of epochs skipped.*

Since all the above metrics are defined locally on each worker, we need no centralized or global monitor to break the decentralized scheme. Particularly, one can easily adapt the general definitions above to any specific distributed GNN systems as the metrics to study how the stale results affect the distributed training.

Bounded Staleness Check Procedure. Now, we introduce the procedure to check whether the cached historical embeddings exceed the staleness bound to manage system staleness caused by the mixed-version problem. One crucial distinction from traditional distributed ML is that the bounded staleness is enforced on the intermediate embeddings $\mathbf{H}_i$ instead of gradients, with a new perspective to avoid unnecessary communication in distributed GNNs by taking the initiative to utilize stale embeddings. However, we need to control the errors caused by using stale embeddings. To allow bounded embedding staleness in a decentralized setting, it is natural to design a light-weighted local state tracker on each worker for efficient bounded embedding staleness check-in Sancus.

From the database community, we adapt the idea of version control [55] but in a decentralized approach to monitor the staleness of the system. We use ver to denote the current training epoch number. Then on each worker i, upon the arrival of each latest H_j from the worker j, we stamp $\mathbf{H}_j$ with version $\mathrm{Ver}_i(j)$, i.e., the epoch number where $\mathbf{H}_j$ is computed, correspondingly. We keep track of the version number for all the $\mathbf{H}_j$ where $j \in [1, \cdots, p]$.

Algorithm 4 Embedding Staleness Check based on Definition 1.

```
Input: Current epoch number Ver; Cached embedding version Ver(j)
procedure STALE( )
    if Ver − Ver(j) > ϵ_E then
        F(i) ← ACTIVE
    else
        F(i) ← STALE
    end if
end procedure
```

Firstly, we introduce the procedure to check the epoch-fixed embedding staleness in Definition 1. The version (epoch number) of latest broadcast $\mathbf{H}_j^{(\ell-1)}$ is stamped with $\mathrm{Ver}(j)$. After a forward and backward pass are finished, a new epoch is proceeded. The algorithm to check the epoch-fixed embedding staleness is shown in Algorithm 4. Notice that the subscript i for $\mathrm{Ver}_i(j)$ is omitted since this basic staleness bound is unified for all workers. An $\epsilon_E = 1$ example in Figure 14(a) shows the embeddings skip-broadcast every other epoch.

Algorithm 5 Embedding Staleness Check based on Definition 2

```
Input: Current epoch number Ver; Cached embedding version Ver_i(j)
procedure STALE( )
    for all process P(i) in parallel do
        for j = 1 to p do
            if Ver − Ver_i(j) > ϵ_A then
                F(j) ← ACTIVE
                EndProcedure for P(j)
            end if
        end for
    end for
end procedure
```

Secondly, for the epoch-adaptive embedding staleness in Definition 2, the latest broadcast $\mathbf{H}_j^{(\ell-1)}$ is stamped with $\mathrm{Ver}_i(j)$ for each $\mathbf{H}_j^{(\ell-1)}$ on processor i. Algorithm 5 depicts the staleness checking procedure. If the embeddings $\tilde{\mathbf{H}}_j^{\ell-1}$ of any other worker j cached on worker i is from ϵ_A epochs ago, the worker i is running too fast for the worker j and $\tilde{\mathbf{H}}_j$ is too stale, so F(j) is turned to ACTIVE to send out $\mathbf{H}_j$ soon. This ensures that the worker only broadcasts the latest results proceeded in the new epoch when it receives updates from other workers at most ϵ_A epochs ago. An example with $\epsilon_A = 1$ is shown in Figure 14(b). In Epoch 9, the results $\mathbf{H}_{2\_7}$ of worker 2 become too stale since $9 - 7 = 2 > \epsilon_A = 1$, worker 2 becomes ACTIVE and broadcasts updated results to all others as shown in Epoch 10. Then worker 1 becomes ACTIVE because others are using H_{1_8} which is too stale. Similarly, worker 3 and 4 are designated as ACTIVE in Epoch 11.

Algorithm 6 Embedding Staleness Check based on Definition 3

```
Input: Current H_i^ℓ computed in P(i); Cached embeddings H_i^ℓ
procedure STALE( )
    if ‖H_i^ℓ − H̃_i^ℓ‖ > ϵ_H then
        F(i) ← ACTIVE
    else
        F(i) ← STALE
    end if
end procedure
```

Lastly, for the epoch-adaptive variation-gap embedding staleness in Definition 3, the staleness check is purely based on the embedding variation locally on each worker. The checking procedure is elaborated in Algorithm 6. If the embedding variation of $\mathbf{H}_i^{\ell}$ is within the bound, the latest results need no broadcasting, and other workers can use the stale historical embedding $\tilde{\mathbf{H}}_i^{e}$. Otherwise, the embedding variation becomes too large, then the latest $\mathbf{H}_i^{\ell}$ needs broadcasting and caching. It is noticed that no version tracker is required here. As Figure 14(c) shows, since the staleness is only based on the embedding variation gap, workers may become STALE after an adaptive number of epochs.

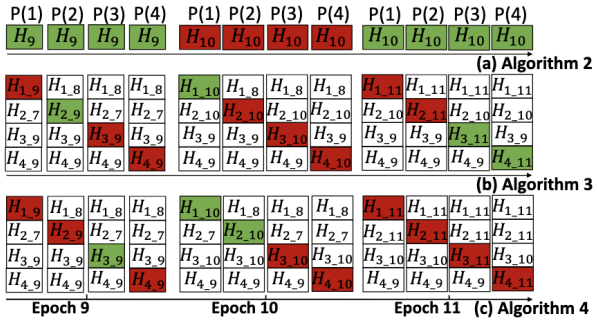


Figure 14: Example training based on each bounded embedding staleness: worker i is denoted by $P(i)$ where $i \in [1, 4]$ and H_{i_e} denotes the results obtained from process i in epoch e; ACTIVE worker is colored green while STALE worker is colored red. For Algorithm 4 and 5 , the staleness bound $\epsilon_E = \epsilon_A = 1$. For Algorithm 6, a toy example is given.

Table 2: Dataset statistics from the Open Graph Benchmark [33].

Dataset	# Nodes	# Edges	Node Features
ogbn-product	2,449,029	61,859,140	100
ogbn-papers100M	111,059,956	3,231,371,744	128

5 Experiments

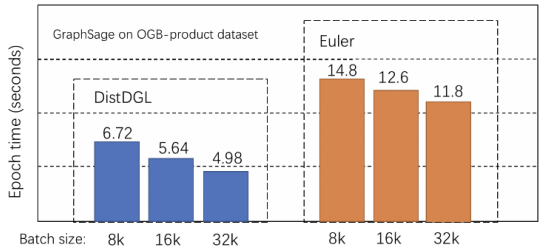
5.1 DistDGL

We focused on the node classification task using GNNs throughout the evaluation. The GNNs for other tasks such as link prediction mostly differ in the objective function while sharing most of the GNN architectures so we omit them in the experiments.

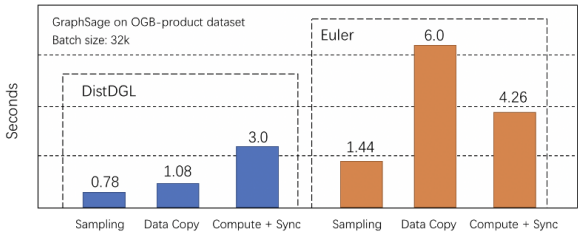
We benchmark the state-of-the-art GraphSAGE [31] model on two Open Graph Benchmark (OGB) datasets [33] shown in Table 2. The GraphSAGE model has three layers of hidden size 256; the sampling fan-outs of each layer are 15, 10 and 5. We use a cluster of eight AWS EC2 m5n.24xlarge instances (96 VCPU, 384GB RAM each) connected by a 100Gbps network.

In all experiments, we use DGL v0.5 and Pytorch 1.5. For Euler experiments, we use Euler v2.0 and TensorFlow 1.12.

5.1.1 DistDGL vs. other distributed GNN frameworks. We compare the training speed of DistDGL with Euler [1], one of the state-of-the-art distributed GNN training frameworks, on four m5n.24xlarge instances. Euler is designed for distributed mini-batch training, but it adopts different parallelization strategy from DistDGL. It parallelizes computation completely with multiprocessing and uses one thread for both forward and backward computation as well as sampling inside a trainer. To have a fair comparison between the two frameworks, we run mini-batch training with the same global batch size (the total size of the batches of all trainers in an iteration) on both frameworks because we use synchronized SGD to train models.



(a) The overall runtime per epoch with different global batch sizes.



(b) The breakdown of epoch runtime for the batch size of 32K.

Figure 15: DistDGLvs Euler on ogbn-product graph on four m5n.24xlarge instances.

DistDGLgets 2.2× speedup over Euler in all different batch sizes (Figure 15(a)). To have a better understanding of DistDGL's performance advantage, we break down the runtime of each component within an iteration shown in Figure 15(b). The main advantage of DistDGL is *data copy*, in which DistDGL has more than 5× speedup. This is expected because DistDGLuses METIS to generate partitions with minimal edge cuts and trainers are co-located with the partition data to reduce network communication. The speed of *data copy* in DistDGL gets close to local memory copy while Euler has to copy data through TCP/IP from the network. DistDGL also has 2× speedup in *sampling* over Euler for the same reason: DistDGL samples majority of vertices and edges from the local partition to generate mini-batches. DistDGL relies on DGL and Pytorch to perform sparse and dense tensor computation in a mini-batch and uses Pytorch to synchronize gradients among trainers while Euler relies on TensorFlow for both mini-batch computation and gradient synchronization. DistDGL is slightly faster in mini-batch computation and gradient synchronization. Unfortunately, we cannot separate the batch computation and gradient synchronization in Pytorch.

5.1.2 Ablation Study. We further study the effectiveness of the main optimizations in DistDGL: 1) reducing network traffic by METIS graph partitioning and co-locating data and computation, 2) balance the graph partitions with multi-constraint partitioning. To evaluate their effectiveness, we compare DistDGL's graph partitioning algorithm with two alternatives: random graph partitioning and default METIS partitioning without multi-constraints. We use a cluster of four machines to run the experiments.

METIS partitioning with multi-constraints to balance the partitions achieves good performance on both datasets (Figure 16). Default METIS partitioning performs well compared with random

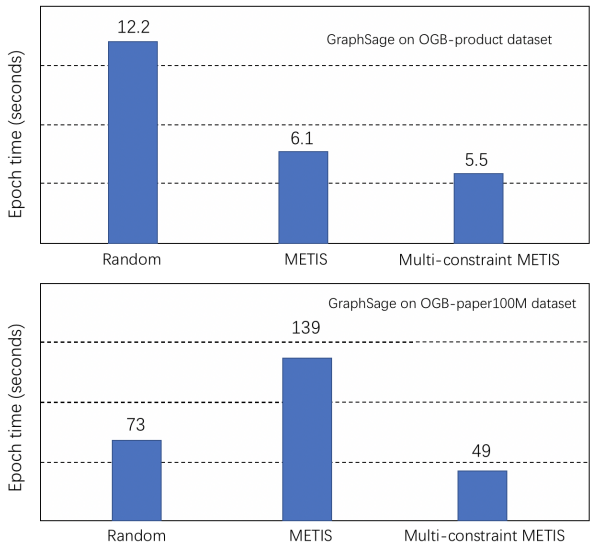


Figure 16: METIS vs Random Partition on four machines

partitioning (2.14× speedup) on the ogbn-product graph due to its superior reduction of network communication; adding multiple constraints to balance partitions gives additional 4% improvement over default METIS partitioning. However, default METIS partitioning achieves much worse performance than random partitioning on the ogbn-papers100M graph due to a high imbalance between partitions created by METIS, even though METIS can effectively reduce the number of edge cuts between partitions. Adding multi-constraint optimizations to balance the partitions, we see the benefit of reducing network communication. This suggests that achieving load balancing is as important as reducing network communication for improving performance.

5.2 ByteGNN

We evaluate the performance of ByteGNN by comparing with Graph-Learn [86], Euler [1], and Distributed DGL (DistDGL) [83]. We also examine the effects of our system designs on the performance.

Testbed. We ran our experiments on a cluster of machines where each machine is equipped with 1T DDR4 main memory and two 2.40GHz Intel(R) Xeon(R) Platinum 8260 CPU (each CPU has 24 cores or 48 virtual cores by hyper-threading). All the machines are connected by a 25 Gbps network and the OS is the Debian 9.13 with Linux kernel 4.19.117.

Datasets. We used three datasets in the evaluation, as shown in Table 3. ogbn-product and ogbn-papers100M are the largest two graphs in the Open Graph Benchmark (OGB). ogbn-product is an undirected and unweighted graph modeling an Amazon product copurchasing network [34]. ogbn-papers100M is a directed citation graph of 111 million papers indexed by MAG [74]. The Social dataset is a directed graph in industry from the social network scenario.

Models. We used three representative GNN models, Graph Convolutional Network (GCN) [47], GraphSAGE [31], and Graph Attention network(GAT) [73], in our evaluation. In order to demonstrate

Dataset	ogbn-product (Product)	ogbn-papers100M (Papers)	Social
Vertices	2, 449, 029	111, 059, 956	66, 351, 656
Edges	123, 718, 280	1, 615, 685, 872	1, 751, 915, 191
Feature	100	128	150
Classes	47	172	2
Training set	196,615	1, 207, 179	6, 631, 989
Validation set	39,323	125,265	19, 908, 461
Test set	2, 213, 091	214,338	39, 811, 206

Table 3: Graph datasets

the expressiveness and efficiency of ByteGNN, we also tested the unsupervised variants of these three models. Although unsupervised

5.2.1 Overall Performance. We evaluated the systems' overall performance by measuring throughput, the number of samples processed per second, a standard metric for training efficiency. Throughput is calculated as the total seed vertices processed divided by the end-to-end GNN training time. Higher throughput indicates shorter training times. We set the hidden size to 32 for GCN and GraphSAGE, and for GAT, used 4 attention heads with a hidden size of 16. Euler was excluded for unsupervised GAT training due to execution failure.

Figure 17 shows ByteGNN outperforming GraphLearn with 7.5 to 16.2x speedup in supervised training and up to 23x in unsupervised training, attributed to the system designs of two-level scheduling. The improvements are more pronounced in unsupervised training due to ByteGNN's parallel sampling capabilities. ByteGNN also achieves up to 4.7x speedup over Euler by enabling concurrent execution of multiple DAGs, which maximizes CPU utilization and avoids convergence issues caused by TensorFlow's computation graphs in Euler.

Against DistDGL, ByteGNN achieves 2.1–3.5x speedup for dense graphs like ogbn-product and outperforms on sparse graphs like ogbn-papers100M and Social, with 1.5x and 1.3x speedups in supervised GCN and GraphSAGE training. The improvements are smaller for sparse graphs, especially with GAT, due to optimized sparse tensor operations in DistDGL, which ByteGNN lacks. In unsupervised training, ByteGNN shows greater gains, achieving 2.4x speedup for GraphSAGE and 1.6x for GAT, thanks to its higher sampling parallelism. Figure 18 reports ByteGNN's CPU utilization, which is 3–6x higher than GraphLearn, Euler, and DistDGL. Lower utilization in supervised GCN and GraphSAGE is due to efficient resource allocation between sampling and training phases and lighter GCN workloads.

5.2.2 Scalability. Figure 19 shows the throughput scalability of systems on the ogbn-papers100M dataset with machine counts ranging from 4 to 64. ByteGNN outperforms all other systems in scalability. Results for other datasets are omitted due to space constraints but follow similar patterns, with ByteGNN performing even better on the dense ogbn-product graph. A hidden size of 256 is used to test performance under different configurations.

Distributed GNN training typically has sub-linear scalability due to synchronization overhead in the BSP model and high network I/O. GraphLearn and Euler scale poorly, with low throughput caused by the lack of an effective graph partitioning algorithm,

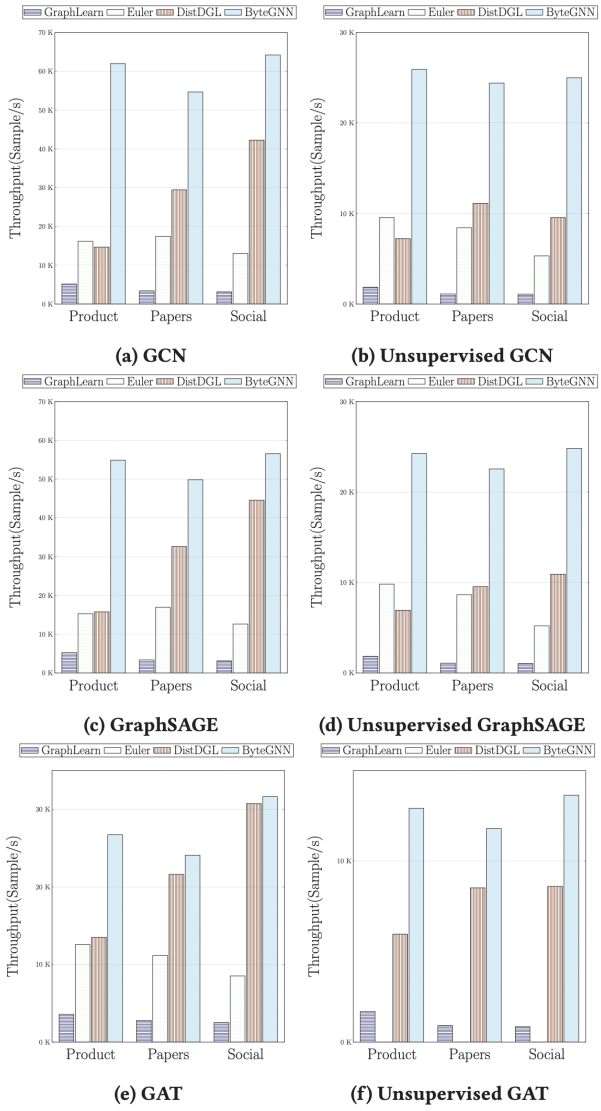


Figure 17: The throughput of GraphLearn, Euler, DistDGL, and ByteGNN for training different models on 4 machines.

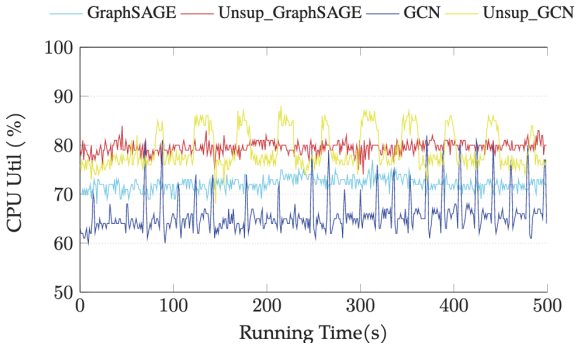


Figure 18: The throughput of GraphLearn, Euler, DistDGL, and ByteGNN for training different models on 4 machines.

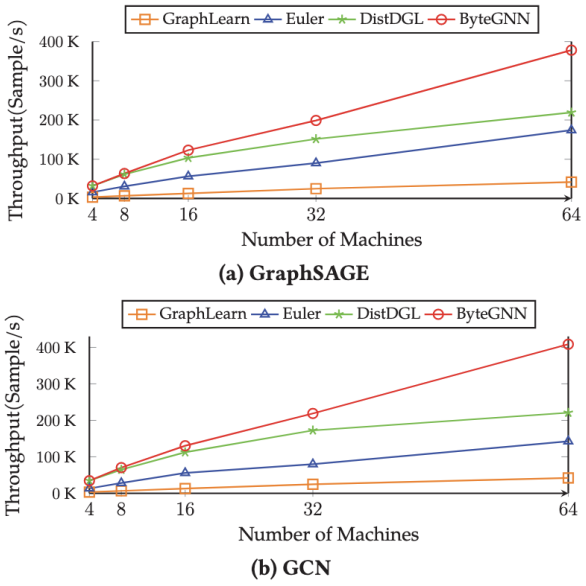


Figure 19: The throughput of GraphLearn, Euler, DistDGL, and ByteGNN for training different models on 4 machines.

leading to significant network communication overhead. Although they use TensorFlow's asynchronous gradient update to minimize synchronization overhead, it risks accuracy loss. DistDGL, on the other hand, suffers from synchronization delays during gradient updates, as trainers must wait for mini-batch sampling outputs. Prefetching helps but doesn't scale well with more machines.

ByteGNN addresses these issues with pipelined sampling and training, ensuring better resource utilization. Its GNN-specific graph partitioning reduces network communication overhead as machine numbers increase, enabling superior scalability compared to other systems.

5.2.3 Model Accuracy. We also report the correctness of ByteGNN by evaluating the test accuracy of the GraphSAGE model on the ogbn-product dataset, comparing with GraphLearn and DistDGL. Euler has similar accuracy as GraphLearn. In Figure 20, we report the test accuracy of different systems at every epoch until the training converges. The result shows that the systems achieve similar or the same accuracy eventually, but ByteGNN converges the fastest, in both the single-machine setting (1M) and distributed 4-machine setting (4M). We also note that as the mini-batch training can update the model many times in one epoch, the accuracy increases quickly in the first several epochs. The single-machine accuracy of GraphLearn can also be seen as the baseline to demonstrate that our code changes to GraphLearn do not affect the semantics of the GNN models. And as ByteGNN uses BSP to ensure model convergence in distributed training, it achieves approximately the same accuracy as DistDGL but uses less time.

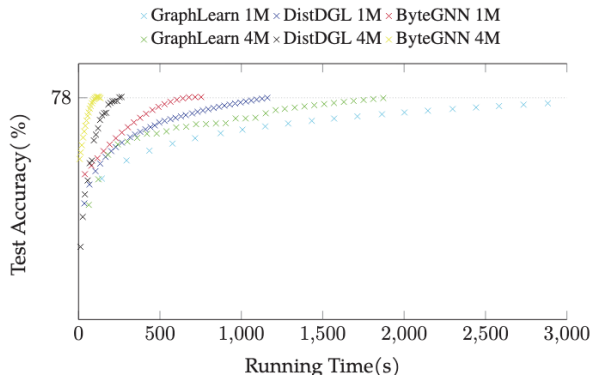


Figure 20: The throughput of GraphLearn, Euler, DistDGL, and ByteGNN for training different models on 4 machines.

5.3 SANCUS

We evaluate SANCUS on five commonly-used [2] large-scale benchmark datasets [34, 79], listed in Table 4. The task on Flickr and Reddit is single-class node classification, while on Amazon, ogbn-products, and ogbn-papers100M is multi-class classifications. Specifically, Flickr models the relations between images uploaded with common properties. Reddit dataset consists of posts and user comments to predict the topical communities that the posts belong to. On Amazon and ogbn-product datasets with node representing product and edge representing products purchased by one customer, we need to categorize the product nodes with multiple labels. Ogbn-papers100M is a citation graph to predict subject areas of papers. All datasets follow the "fixed-partition" splits [34, 79].

Dataset	$\|\mathcal{V}\|$	$\|\mathcal{E}\|$	$\|F\|$	# Class	Byte size
Flickr	89,250	899,756	500	7	529 MB
Reddit	232,965	11, 606, 919	602	41	3.53 GB
Amazon	1, 598, 960	132, 169, 734	200	107(m)	2.34 GB
ogbn-products	2, 449, 029	61, 859, 140	100	47	1.38 GB
ogbn-papers100M	111, 059, 956	1, 615, 685, 872	200	172	56.2 GB

Table 4: Summary of the graph data statistics used in our experiments to evaluate the proposed framework("m": multi-class classification).

We implement SANCUS using a PyTorch-based adaptation of classical parallel algorithms for distributed GNN training. Unlike existing methods, SANCUS adaptively avoids communication by skipping broadcasts of cached historical embeddings, leveraging bounded embedding staleness. We implement ϵ_E (Definition 2) with Algorithms 4 and 2 as **SCS-E**, which skips broadcasts and reuses cached embeddings for ϵ_E epochs. Similarly, ϵ_A (Definition 3) and ϵ_H (Definition 4) are implemented as **SCS-A** and **SCS-H** using Algorithms 5, and Algorithms 6, respectively, allowing adaptive skip-broadcasting. We also implement **SkipG**, based on bounded gradient staleness, to demonstrate the superiority of bounded embedding staleness.

5.3.1 SANCUS is effective. First, we demonstrate the communication reduction effectiveness of SANCUS across benchmark datasets

Config	Operation	CAGNET	SCS-A1	SCS-E	SCS-H
(1) **8*1**	**compute**	0.365	0.359	0.359	0.343
p2p: no	**communicate**	1	0.687	0.697	0.675
(2) **4*2**	**compute**	0.093	0.093	0.092	0.090
p2p: no	**communicate**	1	0.717	0.714	0.698
(3) **4*1**	**compute**	0.437	0.431	0.431	0.425
p2p: yes	**communicate**	1	0.703	0.708	0.692

Table 5: Summary of detailed time breakdown (second) measured in seconds in one epoch with SCS-A/E/H compared to CAGNET, on Reddit dataset.

of increasing scale to show scalability. Figure 21 highlights results with accuracy loss within 0.01. Compared to SOTA methods like CAGNET and SkipG, our SANCUS variants reduce communication by 35% to 74% using bounded embedding staleness. To demonstrate generality, we evaluate the least communication-reducing baselines, SCS-E1/A1/H1, focusing on system configurations. Figure 22 examines GPU configurations, number of GPUs, GNN layers, and hidden feature sizes across ogbn-products, Flickr, Reddit, and ogbn-papers100M. Training time breakdowns of computation and communication are shown in Table 5.

Figure 22(a) shows that communication time varies with GPU configurations, but SANCUS consistently reduces communication across all setups, including single-machine multi-GPU and multi-server environments. Table 3 confirms this across configurations (1)(2)(3). SCS-A1 achieves minimal overhead in all settings, including multi-server setups. As Figure 22(b) shows, increasing GPUs reduces total costs compared to CAGNET. With SCS-H, communication cost using 8 GPUs approaches CAGNET's cost with 2 GPUs, alongside a 67% reduction in computation cost. Additionally, avoided communication proportion increases with GPU count, which centralized PS architectures [52] struggle to achieve. Figure 22(c) demonstrates effective communication reduction across varying GNN depths. Figure 22(d) shows that with increasing hidden feature size, SANCUS's epoch-adaptive strategy results in much slower communication cost growth compared to CAGNET. These results confirm the robustness and generality of our framework across diverse system configurations.

5.3.2 SANCUS converges fast and reserves the GNN performance. With repeated usage of historical embeddings to skip-broadcast adaptively, SANCUS avoids communication while preserving the GNN accuracy. In Figure 23 and 24(b), all proposed variants converge to a very close ($\leqslant$ 0.005), even the same, sometimes the better accuracy results with communication avoiding, compared to CAGNET. Besides, the convergence time to reach satisfying accuracy is much faster. We also show that the traditional bounded gradient method SkipG suffers far more accuracy loss ($\leqslant$ 0.02) compared to our skip-broadcast historical embeddings. In general, we draw the conclusion on the extremely close proximity of the model performance between the proposed training framework and the original GNN training.

5.3.3 SANCUS is staleness-aware. We show the adaptivity in the number of epochs that SANCUS skips when using cached historical embeddings to manage system staleness. As illustrated in Figure

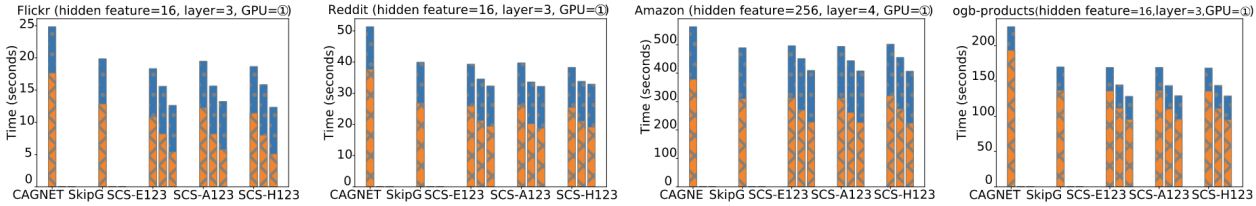
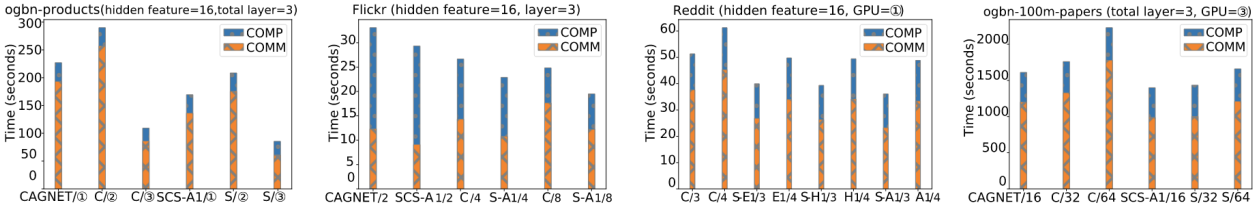
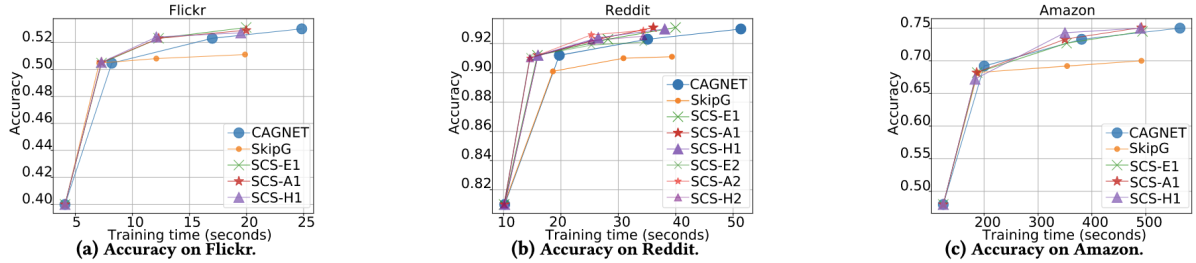


Figure 21: Communication-avoiding performance using all 8 GPUs on Flickr/Reddit/Amazon/ogbn-products datasets. In each subplot, x-axis denotes the methods compared: CAGNET[72], SkipG[59], SCS-E1/E2/E3 with $\epsilon_E = \{1, 2, 3\}$, SCS-A1/A2/A3 with $\epsilon_A = \{1, 2, 3\}$, SCS-H1/H2/H3 with $\epsilon_H = \{0.01, 0.02, 0.03\}$; y-axis denotes the time proportion during training, where blue bar denotes model computation cost and orange bar denotes communication cost.



(a) GPU Configurations. (b) Number of GPUs. (c) Number of Total GNN Layers. (d) Size of Hidden Feature.

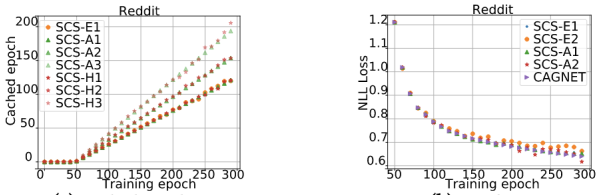
Figure 22: The performance with system architecture variants (GPU configurations/number of GPUs/GNN layer number/GNN hidden feature size) compared to CAGNET on ogbn-products/Flickr/Reddit/ogbn-papers100M datasets, respectively. In each subplot, y-axis denotes the time proportion during training, where blue bar denotes model computation cost and orange bar denotes communication cost. For the x-axis, we denote the method/GPU configurations in Figure 22(a), method/number of GPUs in Figure 22(b), method/layer number in Figure 22(c), and method/hidden feature size in Figure 22(d).



(a) Accuracy on Flickr. (b) Accuracy on Reddit. (c) Accuracy on Amazon.

Figure 23: The accuracy results using all eight GPUs on Flickr/Reddict/Amazon datasets, respectively. In each subplot, x-axis denotes the methods compared: CAGNET[72], SkipG [59], SCS-E1/E2 denotes $\epsilon_E = \{1, 2\}$, SCS-A1/A2 $\epsilon_A = \{1, 2\}$, SCS − H1/H2$\epsilon_H = \{0.01, 0.02\}$; y-xis denotes the time proportion.

24(a), we plot the corresponding epochs that actually perform broadcasting in the training of the proposed framework. As the control group, epoch-fixed SCS-E1 in Figure 24(a) broadcast the latest results in every 2 iterations. Thus its cached epochs - the orange dots, increase regularly. As for epoch-adaptive schemes SCS-A and SCS-H, we find the broadcasts exhibiting irregular patterns, by observing that the interspaces between dots are at varying distances, especially for SCS-H. The explanation is that the staleness in epoch number is adaptively controlled by the staleness tolerance ϵ_H, also shown by the earlier example in Figure 14. In accordance with Figure 21, SCS-E1/A1/H1 cache the least epochs thus avoid the least communication as Figure 21 shows. Compared to the epoch-fixed SCS-E, epoch-adaptive methods provide higher accuracy results (Figure 23). It shows that managing system staleness can lead to better preservation of effectiveness. Notably, though SCS-E3 caches similar epochs with similar communication avoiding in Figure 21, it suffers from the most accuracy loss in Figure 23. Taking all above into account, we conclude that the adaptive strategy with staleness-awareness is more robust and advantageous in communication avoiding with little and even no loss of accuracy, sometimes even better accuracy.



(a) Cached epochs **(b) NLL loss**

Figure 24: The scatter plots with x-axis denoting epochs performing broadcast on Reddit, to show adaptive staleness-awareness. SCS-E1/E3 denotes $\epsilon_E = \{1, 2\}$; SCS-A1/A2/a3 denotes $\epsilon_A = \{1, 2, 3\}$; SCS-H1/H2/H3 denotes $\epsilon_H = \{0.01, 0.02, 0.03\}$.

Dataset	2 GPUs	4 GPUs	8 GPUs
Flickr	170.2 MB	255.3 MB	297.9 MB
Reddit	0.5 GB	0.8 GB	0.9 GB
Amazon	1.2 GB	1.8 GB	2.1 GB
ogbn-products	0.9 GB	1.4 GB	1.6 GB

Table 6: Cache Memory Footprints with configuration(1). Ogbn-paper100M is omitted from the table since it only runs on 4 GPUs with 31GB cache size.

5.3.4 Sancus is memory-efficient. Due to the GPU memory constraints, the scalability of such distributed GNN training is instructed by the memory cost on GPUs. During Sancus training, there are three parts of GPU memory footprints: local data (i.e., embeddings, local adjacency matrix, and full weight matrix), memory for matrix operation, and cache of historical embeddings. Compared to CAGNET, the only extra memory Sancus consumes is the cache of historical embeddings. Thus, we show the cache memory footprints in Table 6. Generally, the cache memory cost is a particularly small proportion for modern GPUs on most datasets.

6 Conclusion

The convergence of distributed and decentralized strategies has transformed the scalability of GNN training on billion-scale graphs. DistDGL's owner-compute principle and partitioning optimizations reduce computational bottlenecks while preserving memory efficiency. ByteGNN brings system-level innovations like mixed-precision computation and GPU-centric hierarchical memory usage, tackling real-world industrial-scale workloads with remarkable throughput improvements. Sancus demonstrates the viability of decentralized training by embracing staleness-aware updates and communication-avoiding techniques, balancing model accuracy and communication overheads. These systems collectively redefine the scalability limits of GNNs, paving the way for future innovations in large-scale graph processing and enabling practical applications across diverse domains, from social networks to biological systems.

References

[1] 2020. Euler Github. https://github.com/alibaba/euler.
[2] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. 2021. Computing Graph Neural Networks: A Survey from Algorithms to Accelerators. *ACM Comput. Surv.* 54, 9, Article 191 (Oct. 2021), 38 pages. https://doi.org/10.1145/3477141
[3] Konstantin Andreev and Harald Räcke. 2004. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Barcelona, Spain) *(SPAA '04)*. Association for Computing Machinery, New York, NY, USA, 120–124. https://doi.org/10.1145/1007912.1007931
[4] David Applegate and William Cook. 1991. A computational study of the job-shop scheduling problem. *ORSA Journal on computing* 3, 2 (1991), 149–156.
[5] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: An efficient communication library for distributed GNN training. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 130–144.
[6] Bo Chen, Chris N Potts, and Gerhard J Woeginger. 1998. A review of machine scheduling: Complexity, algorithms and approximability. *Handbook of Combinatorial Optimization: Volume1–3* (1998), 1493–1641.
[7] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations*.
[8] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction *(Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, Stockholmsmässan, Stockholm Sweden, 942–950.
[9] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 942–950. https://proceedings.mlr.press/v80/chen18p.html
[10] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)* 5, 3 (2019), 1–39.
[11] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Jul 2019).
[12] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO, 571–582.
[13] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. 2013. Solving the straggler problem with bounded staleness. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*.
[14] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 153–167.
[15] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices* 48, 4 (2013), 77–88.
[16] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and qos-aware cluster management. *ACM Sigplan Notices* 49, 4 (2014), 127–144.
[17] Stijn Eyerman, Wim Heirman, Kristof Du Bois, Joshua B. Fryman, and Ibrahim Hur. 2018. Many-Core Graph Workload Analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Article 22, 11 pages.
[18] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, and Jiaxin Jiang. 2017. GRAPE: Parallelizing sequential graph computations. In *43rd International Conference on Very Large Data Bases*. Very Large Data Base Endowment Inc., 1889–1892.
[19] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. 2018. Parallelizing Sequential Graph Computations. *ACM Trans. Database Syst.* 43, 4, Article 18 (Dec. 2018), 39 pages. https://doi.org/10.1145/3282488
[20] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).
[21] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. arXiv:1903.02428 [cs.LG] https://arxiv.org/abs/1903.02428
[22] Matthias Fey, Jan E. Lenssen, Frank Weichert, and Jure Leskovec. 2021. GNNAutoScale: Scalable and Expressive Graph Neural Networks via Historical Embeddings. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 3294–3304. https://proceedings.mlr.press/v139/fey21a.html
[23] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 551–568. https://www.usenix.org/conference/osdi21/presentation/gandhi
[24] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluc. 2018. Integrated Model, Batch, and Domain Parallelism in Training Neural Networks. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures* (Vienna, Austria) *(SPAA '18)*. Association for Computing Machinery, New York, NY, USA, 77–86. https://doi.org/10.1145/3210377.3210394

[25] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. 2016. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 99–115.

[26] Leslie Ann Goldberg, Mike Paterson, Aravind Srinivasan, and Elizabeth Sweedyk. 1997. Better approximation guarantees for job-shop scheduling. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms* (New Orleans, Louisiana, USA) *(SODA '97)*. Society for Industrial and Applied Mathematics, USA, 599–608.

[27] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*.

[28] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[29] Google. [n. d.]. Freebase Data Dumps. https://developers.google.com/freebase/data.

[30] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 455–466.

[31] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. 1025–1035.

[32] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*. 1223–1231.

[33] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687* (2020).

[34] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 22118–22133. https://proceedings.neurips.cc/paper_files/paper/2020/file/fb60d411a5c5b72b2e7d3527cfc84fd0-Paper.pdf

[35] Wen-bing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive Sampling Towards Fast Graph Representation Learning. *CoRR* abs/1809.05343 (2018).

[36] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive Sampling Towards Fast Graph Representation Learning. *arXiv preprint arXiv:1809.05343* (2018).

[37] Nikita Ivkin, Daniel Rothchild, Enayat Ullah, Vladimir Braverman, Ion Stoica, and Raman Arora. 2019. Communication-efficient distributed SGD with Sketching. *arXiv preprint arXiv:1903.04488* (2019).

[38] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.). Vol. 2. 187–198.

[39] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.

[40] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of the Third Conference on Machine Learning and Systems, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2020/hash/91fc23ceccb664ebb0cf4257e1ba9c51-Abstract.html

[41] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware Distributed Parameter Servers. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 463–478. https://doi.org/10.1145/3035918.3035933

[42] Tatiana Jin, Zhenkun Cai, Boyang Li, Chengguang Zheng, Guanxian Jiang, and James Cheng. 2020. Improving resource utilization by timely fine-grained scheduling. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 20, 16 pages. https://doi.org/10.1145/3342195.3387551

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

[44] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[45] George Karypis and Vipin Kumar. 1998. Multilevel Algorithms for Multi-Constraint Graph Partitioning. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing* (San Jose, CA). USA, 1–13.

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

[47] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907* (2016).

[48] Hao Li, Asim Kadav, Erik Kruus, and Cristian Ungureanu. 2015. Malt: distributed data-parallelism for existing ml applications. In *Proceedings of the tenth european conference on computer systems*. 1–16.

[49] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 583–598.

[50] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598.

[51] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *arXiv preprint arXiv:2006.15704* (2020).

[52] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. 2017. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. *Advances in neural information processing systems* 30 (2017).

[53] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 401–415. https://doi.org/10.1145/3419111.3421281

[54] Husong Liu, Shengliang Lu, Xinyu Chen, and Bingsheng He. 2020. G3: when <u>g</u>raph neural networks meet parallel <u>g</u>raph processing systems on <u>G</u>PUs. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 2813–2816. https://doi.org/10.14778/3415478.3415482

[55] Frederick H Lochovsky and Won Kim. 1989. *Object-Oriented Concepts, Databases and Applications*. ACM Press.

[56] Qinyi Luo, Jinkun Lin, Youwei Zhuo, and Xuehai Qian. 2019. Hop: Heterogeneity-aware decentralized training. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 893–907.

[57] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA, 443–458.

[58] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. New York, NY, USA, 135–146.

[59] Xupeng Miao, Xiaonan Nie, Yingxia Shao, Zhi Yang, Jiawei Jiang, Lingxiao Ma, and Bin Cui. 2021. Heterogeneity-Aware Distributed Machine Learning Training via Partial Reduce. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2262–2270. https://doi.org/10.1145/3448016.3452773

[60] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021. Large graph convolutional network training with GPU-oriented data communication architecture. *Proc. VLDB Endow.* 14, 11 (July 2021), 2087–2100. https://doi.org/10.14778/3476249.3476264

[61] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. 2011. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. *arXiv preprint arXiv:1106.5730* (2011).

[62] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu. 2017. Do We Need Specialized Graph Databases? Benchmarking Real-Time Social Networking Applications. In *Proceedings of the Fifth International Workshop on Graph Data-Management Experiences & Systems* (Chicago, IL, USA) *(GRADES'17)*. Association for Computing Machinery, New York, NY, USA, Article 12, 7 pages. https://doi.org/10.1145/3078447.3078459

[63] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. New York, NY, USA, 16–29.

[64] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski,

Steven Hand, and John Wilkes. 2020. Autopilot: workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 16, 16 pages. https://doi.org/10.1145/3342195.3387524

[65] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*.

[66] Marco Serafini and Hui Guan. 2021. Scalable Graph Neural Network Training: The Case for Sampling. *SIGOPS Oper. Syst. Rev.* 55, 1 (June 2021), 68–76. https://doi.org/10.1145/3469379.3469387

[67] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[68] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating dynamic graph analytics on GPUs. *Proc. VLDB Endow.* 11, 1 (Sept. 2017), 107–120. https://doi.org/10.14778/3151113.3151122

[69] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *SIGPLAN Not.* 48, 8 (Feb. 2013), 135–146.

[70] Alok Tripathy, Katherine Yelick, and Aydin Buluc. 2020. Reducing Communication in Graph Neural Network Training. *arXiv preprint arXiv:2005.03300* (2020).

[71] Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2020. Reducing communication in graph neural network training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 70, 17 pages.

[72] Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2020. Reducing Communication in Graph Neural Network Training. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1109/SC41405.2020.00074

[73] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[74] Kuansan Wang, Zhihong Shen, Chiyuan Huang, Chieh-Han Wu, Yuxiao Dong, and Anshul Kanakia. 2020. Microsoft Academic Graph: When experts are not enough. *Quantitative Science Studies* 1, 1 (02 2020), 396–413. https://doi.org/10.1162/qss_a_00021 arXiv:https://direct.mit.edu/qss/article-pdf/1/1/396/1760880/qss_a_00021.pdf

[75] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[76] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

[77] Lintao Xian, Bingzhe Li, Jing Liu, Zhongwen Guo, and David HC Du. 2021. Hps: A heterogeneous-aware parameter server with distributed neural network training. *IEEE Access* 9 (2021), 44049–44058.

[78] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. *CoRR* abs/1806.01973 (2018).

[79] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931* (2019).

[80] Dalong Zhang, Xin Huang, Ziqi Liu, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Zhiqiang Zhang, Lin Wang, Jun Zhou, Yang Shuang, and Yuan Qi. 2020. AGL: a Scalable System for Industrial-purpose Graph Machine Learning. *arXiv preprint arXiv:2003.02454* (2020).

[81] Wentao Zhang, Xupeng Miao, Yingxia Shao, Jiawei Jiang, Lei Chen, Olivier Ruas, and Bin Cui. 2020. Reliable data distillation on graph convolutional network. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1399–1414.

[82] Wentao Zhang, Zhi Yang, Yexin Wang, Yu Shen, Yang Li, Liang Wang, and Bin Cui. 2021. Grain: Improving data efficiency of graph neural networks via diversified influence maximization. *arXiv preprint arXiv:2108.00219* (2021).

[83] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2021. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. arXiv:2010.05337 [cs.LG] https://arxiv.org/abs/2010.05337

[84] Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis. 2020. DGL-KE: Training Knowledge Graph Embeddings at Scale. *arXiv preprint arXiv:2004.08532* (2020).

[85] Hongkuan Zhou, Ajitesh Srivastava, Hanqing Zeng, Rajgopal Kannan, and Viktor Prasanna. 2021. Accelerating large scale real-time GNN inference using channel pruning. *Proc. VLDB Endow.* 14, 9 (May 2021), 1597–1605. https://doi.org/10.14778/3461535.3461547

[86] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *arXiv preprint arXiv:1902.08730* (2019).

[87] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[88] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A {Computation-Centric} distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 301–316.

[89] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. 2019. Layer-Dependent Importance Sampling for Training Deep and Large Graph Convolutional Networks. *arXiv preprint arXiv:1911.07323* (2019).