On Pipelined GCN with Communication-Efficient Sampling and Inclusion-Aware Caching

Shulin Wang*, Qiang Yu*, Xiong Wang*, Yuqing Li†, Hai Jin*
* National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
† School of Cyber Science and Engineering, Wuhan University, Wuhan, China
E-mail: *{shulin, qiangyu, xiongwang, hjin}@hust.edu.cn, †li.yuqing@whu.edu.cn

IEEE INFOCOM 2024 - IEEE Conference on Computer Communications | 979-8-3503-8350-8/24/$31.00 ©2024 IEEE | DOI: 10.1109/INFOCOM52122.2024.10621386

***Abstract*—Graph convolutional network (GCN) has achieved enormous success in learning structural information from unstructured data. As graphs become increasingly large, distributed training for GCNs is severely prolonged by frequent *cross-worker communications*. Existing efforts to improve the training efficiency often come at the expense of GCN performance, while the communication overhead *persists*. In this paper, we propose PSC-GCN, a *holistic pipelined framework* for distributed GCN training with communication-efficient sampling and inclusion-aware caching, to address the communication bottleneck while ensuring satisfactory model performance. Specifically, we devise an *asynchronous pre-fetching* scheme to retrieve stale statistics (features, embedding, gradient) of boundary nodes in advance, such that the embedding aggregation and model update are pipelined with statistics transmission. To alleviate communication volume and staleness effect, we introduce a *variance-reduction* based sampling policy, which *prioritizes* inner nodes over boundary ones for reducing the access frequency to remote neighbors, thus mitigating cross-worker statistics exchange. Complementing graph sampling, a feature caching module is *co-designed* to buffer hot nodes with high inclusion probability, ensuring that frequently sampled nodes will be available in local memory. Extensive evaluations on real-world datasets show the superiority of PSC-GCN over state-of-the-art methods, where we can reduce training time by *72%-80%* without sacrificing model accuracy.**

I. Introduction

Recent years have witnessed the pervasiveness of graph-structured data in many real-world applications, such as social network analysis [1], molecular protein discovery [2], road traffic prediction [3], and knowledge graph construction [4]. To handle such unstructured data, substantial efforts have been devoted to extending deep neural networks to learn the structural representation on graphs, giving rise to the paradigm of graph neural networks [5]. Among these approaches, graph convolutional networks (GCNs) have emerged as a promising method for dealing with various graph-related situations [6], and achieved immense success in handling general tasks including the node classification [7], link prediction [8], and graph classification [9].

GCNs extract structural information by leveraging traditional convolution operations and iterative neighbor aggregations. To learn the embedding of a specific graph node, GCNs aggregate the features and embedding of its neighbors, and perform convolutional matrix multiplication layer by layer. While this unique operation facilitates GCNs to effectively express the graph structure, it comes at the expense of *training efficiency* [10], [11]. At a high level, the edge connections between nodes impose dependencies among graph data, which can hinder the efficient learning of node representations, or the embedding, during GCN training. Even worse, real-world graphs can be huge, may contain million or billion nodes and edges [12], [13]. Such large graph size far exceeds the GPU memory capacity (at tens of GB), making it impossible to load the entire data into a single worker device. As such, GCN training on large graphs is commonly conducted in a *distributed* manner, where the graph is partitioned into various subgraphs among multiple workers and processed in parallel. Although distributed GCN relieves the memory constraint on an individual worker, it introduces *significant communication overhead* due to iterative neighbor aggregations. To elaborate, statistics of boundary nodes (neighbor nodes required for local embedding learning) are stored in remote workers, and as GCN models become deeper, the number of boundary nodes can increase exponentially. Consequently, cross-worker transmission also grows drastically, leading to communication becoming the *major bottleneck* that impairs the training efficiency.

Considerable efforts have been devoted to taming the communication bottleneck for expediting GCN training. In particular, various sampling-based techniques are proposed to reduce the communication overhead by down sampling the graphs [1], [14], [15]. Besides, discarding unimportant features of sampled nodes also results in a communication mitigation, especially the cross-worker transmission [16]. Though effective, sampling-based distributed GCN often incurs *inaccurate embedding estimation* which harms the learning performance, yet the communication overhead still remains a significant factor and *dominates* the overall training time. An alternative approach is to leverage the full graph and use

This research was supported in part by National Key Research and Development Program of China under Grant 2022ZD0115301; in part by the National Natural Science Foundation of China under Grant 62202185 and Grant 62302343; in part by the Hubei Provincial Natural Science Foundation of China under Grant 2022CFB611; and in part by the Guangdong Basic and Applied Basic Research Foundation under Grant 2022A1515110396. (Corresponding author: Xiong Wang.)

asynchronous GCN training by retrieving stale statistics of boundary nodes [17]. This method is restricted to the *one-step asynchronization*. Caching a small portion of graph data helps reduce the feature transmission frequency [18]. Nonetheless, existing caching policies typically *operate standalone* without the coordination with the training process, often resulting in a poor cache hit rate. A critical question remaining suspended is *how to fully pipeline the distributed GCN training while ensuring favorable learning performance*. To cope with this problem, researchers face the following challenges.

First and foremost, addressing the communication bottleneck in distributed GCN requires a *holistic design* in the training framework, rather than only focusing on specific acceleration techniques like the asynchronous scheme. To improve the efficiency of learning GCN models, we need to carefully orchestrate the workflow of each training subroutine with exquisitely-designed *distributed implementations*, which is still unresolved. Second, graph sampling has been recognized as an effective approach to extend the scalability of handling large graphs. However, aside from the non-negligible additional overhead, sampling policy should also strike a delicate balance between the *incurred errors* in embedding estimation and *transmission cost* in cross-worker communication, which can not be accommodated by existing sampling models. Third, pipelining the communication and computation processes in distributed GCN needs to align the GPU execution speed with the network transmission rate. In practice, cutting-edge GPU workers often run significantly faster than statistics exchange via CPU and NIC. This speed mismatch imposes unique challenges on achieving a seamlessly pipelined training *without compromising* the learning performance.

In this paper, we propose a comprehensive *pipelined framework* with one-hop neighbor *sampling* and node feature *caching* to minimize the communication overhead in distributed GCN training. Concretely, we first devise a *general* asynchronous scheme to pre-fetch stale statistics (features, embedding, intermediate gradient) of sampled boundary nodes in advance. By doing this, we aim to hide the cross-worker statistics transmission into local embedding aggregation and model update. Considering that large statistics staleness for pipelining can harm the GCN performance, we then design a communication-efficient sampling policy to alleviate the transmission volume as well as the staleness effect. Our designed sampling can select the best one-hop neighbor nodes to ensure a *full-graph* training and *minimize embedding estimation error*, leading to enhanced learning performance compared to vanilla schemes. Complementing the graph sampling, we further co-design a caching policy to locally buffer hot boundary vertices with high *inclusion probabilities*, which significantly improves the cache hit rate. As a result, our framework ensures fairly low statistics staleness, while effectively pipelining the distributed GCN training. The main contributions are summarized.

- We propose PSC-GCN, a *holistic pipelined* framework with communication-efficient sampling and inclusion-aware caching, to *fully mitigate* the communication overhead in distributed GCN training. Specifically, we characterize a *general* asynchronous scheme to retrieve sampled boundary nodes in advance, which can tame the communication bottleneck and enjoy an $O(T^{-2/3})$ training convergence.
- We design a *variance-reduction* based graph sampling policy that *prioritizes* the inner nodes over boundary ones. As a result, cross-worker statistics exchange is significantly reduced, while low embedding estimation error is achieved simultaneously. By employing a *semi-dynamic* sampling execution, we can greatly promote GCN performance without introducing extra sampling computations to each worker.
- To alleviate the communication and staleness impact in pipelined GCN training, we incorporate graph sampling to devise a light-weight caching policy for buffering boundary nodes in local GPU memory. In particular, cache update is implemented based on the *inclusion probability* that a node will be sampled, thus *optimizing* the potential cache hit rate.
- Extensive experiments on real-world datasets are carried out, which corroborate the superiority of PSC-GCN over state-of-the-art approaches by reducing *72%-80%* training time and achieving *4.6-6.3* times faster per model update.

II. Preliminary and System Overview

We start by introducing the basics of GCNs, and then provide a high-level system overview of PSC-GCN.

A. GCNs and Distributed Training

1) Graph Convolutional Networks: GCNs are designed for graph-structured data to learn the feature vector (embedding) for each node of the graph [1], [6]. Denote a graph as $\mathcal{G} = (\mathcal{V}, \mathcal{E}, X)$, where $\mathcal{V}$ and $\mathcal{E}$ are node and edge sets, respectively, with $X \in R^{|\mathcal{V}| \times D}$ representing the node feature matrix. To learn node embedding, GCN performs *neighbor aggregation and node update* in each layer:

$$Z^{(l+1)} = PH^{(l)}W^{(l+1)},\ H^{(l+1)} = \sigma(Z^{(l+1)}). \tag{1}$$

Here $P = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ is the propagation matrix, where $\tilde{A} = A + I_{|\mathcal{V}|}$ is the adjacency matrix of $\mathcal{G}$ including self-connections and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ denotes the degree matrix. Besides, $W^{(l)}$ is the trainable weight matrix and $\sigma(\cdot)$ indicates the activation function, like ReLU, to introduce non-linearity. The embedding $H^{(l+1)}$, or intermediate embedding $Z^{(l+1)}$, in the $(l+1)$-th layer is characterized by aggregating the embedding vectors of neighbors from previous layer, and often we mark $H^{(0)} = X$ for ease of exposition. Taking a node $i \in \mathcal{V}$ as an example, whose one-hop neighbor set is $\mathcal{N}_i$, then $Z_i^{(l+1)} = (PH^{(l)})_i W^{(l+1)} = (\sum_{j \in \mathcal{N}_i} P_{ij} H_j^{(l)})W^{(l+1)}$.

2) Sampling-based GCN Training: Training an L-layer GCN is to learn the weight parameter $\Theta = [W^{(1)}, \cdots, W^{(L)}]$. This can be achieved by minimizing the loss function:

$$\mathcal{L}(\Theta) = \frac{1}{|\mathcal{V}|}\sum_{i \in \mathcal{V}} \mathcal{L}(y_i, Z_i^{(L)}), \tag{2}$$

and performing iterative gradient descent on Θ, where y_i denotes the node label and $Z_i^{(L)}$ is the *output* intermediate embedding for label prediction.

TABLE I: Average inner/boundary nodes based on METIS partition.

Dataset	Partitions	Inner nodes	Boundary nodes
Reddit	4	58.2K	94.9K
	6	38.8K	89.4K
	8	29.1K	90.7K
Ogbn-products	4	612.3K	271.3K
	6	408.2K	245.3K
	8	306.1K	208.8K
Yelp	4	179.2K	120.6K
	6	119.5K	124.7K
	8	89.60K	115.5K

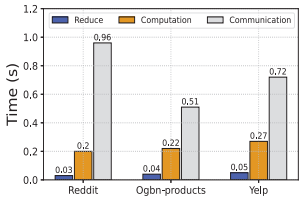


Fig. 1: Training time.

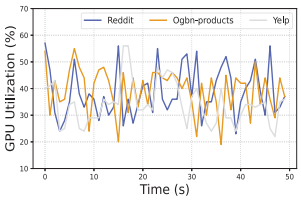


Fig. 2: GPU utilization.

Real-world graphs can be very large [19], far exceeding the capability of vanilla full-graph training. As such, sampling-based techniques, like node-wise GraphSAGE [1], layer-wise LADIES [14] or sub-graph sampling GraphSAINT [15], have been devised to down-sample a mini-batch neighbors for GCN training, thereby extending the scalability that a single device can handle. In the process, propagation matrix $P^{(l)}$ in the l-th layer will be rescaled by the sampling probability. While sampling approaches effectively reduce the training graph size, they also bring in *inaccurate embedding aggregation* [20].

3) Distributed GCN Training: To build GCN models on giant graphs, distributed training complements graph sampling by using multiple workers (GPUs). In general, the full graph is initially divided into several partitions, each stored in a different worker, employing methods like METIS [21]. During training, each worker samples a mini-batch nodes and fetches their features from local or remote partitions in every iteration, and then performs forward pass to learn node embedding and backward pass to update weight parameters. That is, distributed GCN training involves *frequent statistics transmission* (embedding, gradient, etc.) through PCIe or LAN.

B. Motivation

Despite the benefits of distributed GCN, communication has been recognized as the primary bottleneck that impairs training efficiency [11]. Table I provides the statistical information of inner and boundary nodes conditioned on various partitions, which shows that boundary nodes are at the same-level as inner nodes, implying high cross-worker communications during GCN training. To elaborate, we train GCN with 0.3 node-wise sampling rate on Reddit [1], Ogbn-products [19], and Yelp [15] using DGL [22], where we employ six NVIDIA T4 GPUs linked by 15Gpbs network. We display the training time in Fig. 1, as well as the GPU utilization in Fig. 2. One can observe that the communication time *significantly surpasses* computation overhead, occupying around 66%-81% of the total training time. Also, GPU utilization fluctuates at a *low level*, with only 35%-39% computing resource being used.

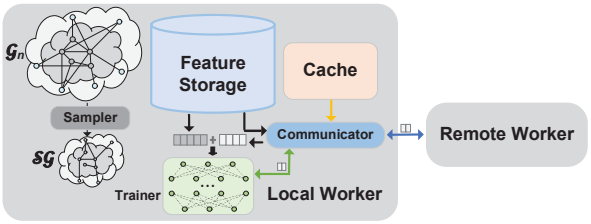


Fig. 3: PSC-GCN overview.

There is a clear gap between the network transmission rate and GPU execution speed. Although recent advancements have been made in sampling-based training [23] or communication computation pipeline [17], the communication bottleneck still persists with substantial statistics exchange cost. This highlights the urgency to co-design a distributed GCN framework to fully eliminate the communication overhead while ensuring model performance.

C. System Overview

In this paper, we propose PSC-GCN, a pipelined framework with communication-efficient sampling and inclusion-aware caching, to mitigate communications in learning GCN models.

1) PSC-GCN Workflow: At a high level, PSC-GCN is a full-graph approach with one-hop neighbor sampling to balance the training efficiency and learning performance. In this framework, each worker *asynchronously* retrieves the statistics (features, embedding, intermediate gradient) of sampled boundary neighbors, as they are stored in remote workers. Fig. 3 illustrates the system workflow of PSC-GCN, which is mainly composed of the trainer, sampler, and cache modules.

PSC-GCN *trainer* is responsible for scheduling the training graph to GPU for processing. In the t-th iteration of model update, trainer uses stale information in the $(t-\tau)$-th iteration of sampled boundary nodes and the latest information of inner nodes (co-located at the same worker) in forward and backward pass. Such asynchronous scheme pipelines the communications with local computations. However, it is essential to ensure that the staleness τ is not too large, as excessive staleness may adversely affect the GCN performance. *Sampler* module is designed to sample one-hop neighbors so as to reduce the communication volume, or equivalently the staleness τ. Sampler requires to determine the selected neighbors to form the training graph, so that trainer can fetch the boundary nodes' statistics in the t-th iteration, for use in the $(t+\tau)$-th model update accordingly. *Cache* module complements the sampler by caching features of hot boundary nodes. This way, trainer could directly access the node features from local memory if they are sampled and cached, thereby further mitigating the cross-worker transmissions. *Communicator* is invoked when we need to transfer the statistics during training.

2) Challenge: The implementation of PSC-GCN is confronted with three primary challenges. First, PSC-GCN is a general asynchronous framework for distributed GCN training. How to *fully pipeline* the communication under the premise of *favorable convergence* is still unresolved. Second, neighbor sampling in a distributed fashion needs to *balance the embedding estimation error and communication overhead*, which has

not been extensively examined before. Last, feature caching should be light-weight enough while ensuring high hit rate of sampled boundary nodes. This requires a *co-design* of sampling and caching polices instead of separate explorations.

III. Pipelining Distributed GCN Training

In this section, we will illustrate the framework of PSC-GCN and analyze its convergence accordingly.

A. PSC-GCN Design

1) Full-graph Training Description: For ease of exposition, we first elucidate full-graph based synchronous distributed GCN training. The whole graph $\mathcal{G}$ will be divided into N partitions $\{\mathcal{G}_0, \ldots, \mathcal{G}_{N-1}\}$ among N workers in the pre-training phase, where partition $\mathcal{G}_n$ holds the *inner nodes* $\mathcal{I}_{\mathcal{G}_n}$ and *boundary nodes* $\mathcal{B}_{\mathcal{G}_n}$ from other graph partitions. In general, boundary nodes are required in *embedding aggregation* for inner nodes across neighbor partitions. In the t-th iteration, worker n will attain the embedding of $\mathcal{B}_{\mathcal{G}_n}$, that is concatenated with the inner node embedding to form:

$$H_{\mathcal{G}_n}^{(t,l)} = \text{concat}\left(H_{\mathcal{I}_{\mathcal{G}_n}}^{(t,l)}, H_{\mathcal{B}_{\mathcal{G}_n}}^{(t,l)}\right). \tag{3}$$

Then, the $(l+1)$-th layer embedding $H_{\mathcal{I}_{\mathcal{G}_n}}^{(l+1)}$ of $\mathcal{I}_{\mathcal{G}_n}$ is obtained according to Eq. (1). Regarding an L-layer GCN model, we have to iterate such neighbor aggregation for L times until the forward pass completes. As a result, we can acquire the training loss $\mathcal{L}(\Theta)$ based on Eq. (2) using the final outputs $Z_{\mathcal{I}_{\mathcal{G}_n}}^{(L)}$ and labels $Y_{\mathcal{I}_{\mathcal{G}_n}}$. Gradient with respect to (w.r.t.) the predicted labels, also denoted as the last embedding $H_{\mathcal{I}_{\mathcal{G}_n}}^{(L)}$ for brevity, is attained immediately:

$$J_{\mathcal{I}_{\mathcal{G}_n}}^{(t,L)} = \frac{\partial \mathcal{L}(\Theta)}{\partial H_{\mathcal{I}_{\mathcal{G}_n}}^{(t,L)}}. \tag{4}$$

In the backward pass, each worker calculates the gradient $G_n^{(t,l+1)}$ corresponding to each layer weight $W^{(l+1)}$, which follows the chain rule:

$$G_n^{(t,l+1)} = \left[P_{\mathcal{G}_n} H_{\mathcal{G}_n}^{(t,l)}\right]^\top \left(J_{\mathcal{I}_{\mathcal{G}_n}}^{(t,l+1)} \circ \sigma'\left(P_{\mathcal{G}_n} H_{\mathcal{G}_n}^{(t,l)} W^{(t,l+1)}\right)\right), \tag{5}$$

where $\top$ means matrix transpose. One can see that the weight gradient depends on the gradient w.r.t. layer embedding, i.e.,

$$J_{\mathcal{G}_n}^{(t,l)} = P_{\mathcal{G}_n}^\top \left(J_{\mathcal{I}_{\mathcal{G}_n}}^{(t,l+1)} \circ \sigma'\left(P_{\mathcal{G}_n} H_{\mathcal{G}_n}^{(t,l)} W^{(t,l+1)}\right)\right) \left[W^{(t,l+1)}\right]^\top, \tag{6}$$

where $J_{\mathcal{G}_n}^{(t,l)} = \text{concat}\left(J_{\mathcal{I}_{\mathcal{G}_n}}^{(t,l)}, J_{\mathcal{B}_{\mathcal{G}_n}}^{(t,l)}\right)$. Owing to the connection between neighbors, embedding gradient of inner nodes and that of boundary nodes need to be merged if they intersect. More concretely, consider $\mathcal{IC}_{\mathcal{G}_n} \subseteq \mathcal{I}_{\mathcal{G}_n}$ as the inner nodes which are also boundary nodes in remote partitions. Then

$$J_{\mathcal{I}_{\mathcal{G}_n}}^{(t,l)} = \text{accumulate}(J_{\mathcal{I}_{\mathcal{G}_n}}^{(t,l)}, J_{\mathcal{IC}_{\mathcal{G}_n}}^{(t,l)}). \tag{7}$$

Here $J_{\mathcal{IC}_{\mathcal{G}_n}}^{(t,l)}$ is calculated by remote workers and added to $J_{\mathcal{I}_{\mathcal{G}_n}}^{(t,l)}$ computed by worker n. In other words, statistics transmission is also demanded in the backward pass.

Finally, workers perform all-reduce on all GCN layers:

$$G^{(t,l)} = \text{AllReduce}\left(G_n^{(t,l)}\right), \forall l = 1, \cdots, L, \tag{8}$$

Algorithm 1: PSC-GCN training (worker view)

Input: Partition $\mathcal{G}_n$, inner node set $\mathcal{I}_{\mathcal{G}_n}$, features $X_{\mathcal{I}_{\mathcal{G}_n}}$, labels $Y_{\mathcal{I}_{\mathcal{G}_n}}$, learning rate η, initial model W_0, total iterations T, staleness τ, sample rate ε

Output: Weight parameter W_T

Initialize $H_{\mathcal{I}_{\mathcal{G}_n}}^{(t,0)} = X_{\mathcal{I}_{\mathcal{G}_n}}$ and sample graph queue $Q = \emptyset$;
for τ *iterations* **do**
  $\mathcal{SG} = \text{SampleGraph}(\mathcal{G}_n, \mathcal{I}_{\mathcal{G}_n}, \varepsilon)$, $Q.\text{push}(\mathcal{SG})$;
for $t = 0, \cdots, T-1$ **do**
  $\mathcal{TG} = Q.\text{pop}()$;
  with $thread_s$
    $\mathcal{SG} = \text{SampleGraph}(\mathcal{G}_n, \mathcal{I}_{\mathcal{G}_n}, \varepsilon)$, $Q.\text{push}(\mathcal{SG})$;
  for $l = 0, \cdots, L-1$ **do**
    if $t > \tau$ **then**
      Wait for $thread_f^{t-\tau}$ to complete;
      $H_{\mathcal{TG}}^{(t,l)} = \text{concat}(H_{\mathcal{I}_{\mathcal{TG}}}^{(t,l)}, H_{\mathcal{B}_{\mathcal{TG}}}^{(t-\tau,l)})$;
    with $thread_f^t$
      Fetch $H_{\mathcal{B}_{\mathcal{SG}}}^{(t,0)}$ from cache or remote workers;
      Fetch $H_{\mathcal{B}_{\mathcal{SG}}}^{(t,l)}, l > 0$ from remote workers;
    $H_{\mathcal{TG}}^{(t,l+1)} = \sigma\left(P_{\mathcal{TG}} H_{\mathcal{TG}}^{(t,l)} W^{(t,l+1)}\right)$;
  $\mathcal{L}(\Theta^{(t)}) = \mathcal{L}\left(Z_{\mathcal{I}_{\mathcal{TG}}}^{(t,L)}, Y_{\mathcal{I}_{\mathcal{TG}}}\right)$, $J_{\mathcal{I}_{\mathcal{TG}}}^{(t,L)} = \frac{\partial \mathcal{L}(\Theta^{(t)})}{\partial H_{\mathcal{I}_{\mathcal{TG}}}^{(t,L)}}$;
  for $l = L-1, \cdots, 0$ **do**
    $G_n^{(t,l+1)} = \left[P_{\mathcal{TG}} H_{\mathcal{TG}}^{(t,l)}\right]^\top \left(J_{\mathcal{I}_{\mathcal{TG}}}^{(t,l+1)} \circ \sigma'\left(P_{\mathcal{TG}} H_{\mathcal{TG}}^{(t,l)} W^{(t,l+1)}\right)\right)$;
    if $l > 0$ **then**
      $J_{\mathcal{TG}}^{(t,l)} = P_{\mathcal{TG}}^\top \left(J_{\mathcal{I}_{\mathcal{TG}}}^{(t,l+1)} \circ \sigma'\left(P_{\mathcal{TG}} H_{\mathcal{TG}}^{(t,l)} W^{(t,l+1)}\right)\right) \left[W^{(t,l+1)}\right]^\top$;
      if $t > \tau$ **then**
        Wait for $thread_b^{t-\tau}$ to complete;
        $J_{\mathcal{I}_{\mathcal{TG}}}^{(t,l)} = \text{accumulate}(J_{\mathcal{I}_{\mathcal{TG}}}^{(t,l)}, J_{\mathcal{IC}_{\mathcal{TG}}}^{(t-\tau,l)})$;
      with $thread_b^t$
        Fetch $J_{\mathcal{IC}_{\mathcal{SG}}}^{(t,l)}$ from remote workers;
  $G^{(t)} = \text{AllReduce}\left(G_n^{(t)}\right)$;
  $\Theta^{(t+1)} = \Theta^{(t)} - \eta G^{(t)}$;

and update weight parameters:

$$W^{(t+1,l)} = W^{(t,l)} - \eta G^{(t,l)}, \forall l = 1, \cdots, L. \tag{9}$$

Since GCN model often includes only 3 or 4 layers [1], [6], the all-reduce communication volume is negligible compared to the statistics exchange, as also verified in Fig. 1.

2) Asynchronous PSC-GCN: Synchronous distributed GCN encounters frequent statistics exchange, which forces the local embedding and gradient computation to stop and wait. To tame this communication bottleneck, PSC-GCN implements an asynchronous training to pre-fetch essential statistics, like embedding and intermediate gradient, of boundary nodes $\mathcal{B}_{\mathcal{G}_n}$

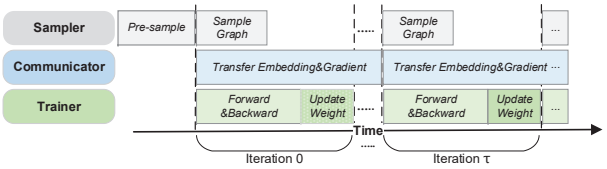


Fig. 4: Pipelined computation and communication.

ahead of $\tau, \tau \geq 1$ iterations before the t-th model update. In this respect, Eq. (3) will involve $H_{\mathcal{B}_{\mathcal{G}_n}}^{(t-\tau,l)}$ and Eq. (7) uses $J_{\mathcal{IC}_{\mathcal{G}_n}}^{(t-\tau,l)}$, so that the *cross-worker communication is overlapped with local computation*. As large staleness τ may damage GCN performance, we incorporate graph sampling and caching into PSC-GCN to reduce the communication volume and statistics staleness, which will be elaborated in subsequent sections.

B. Algorithm Description

In general, we present our PSC-GCN in Algorithm 1 and show the pipelining process in Fig. 4. At the beginning, we initialize $H_{\mathcal{I}_{\mathcal{G}_n}}^{(t,0)}$ as the inner node features $X_{\mathcal{I}_{\mathcal{G}_n}}$, and push the sampled graph $\mathcal{SG}$ (inner nodes with their sampled one-hop inner and boundary neighbors) into queue Q, where $\mathrm{SampleGraph}()$ in Line 3 implies the communication-efficient sampling policy, to be explained in the next section. In iteration t, every worker n first constructs the training graph $\mathcal{TG}$ by popping the head out of queue Q, and also samples new graph ($thread_s$) for future use due to the τ-step asynchronization (Lines 5-7). Based on $\mathcal{TG}$, worker n performs the forward pass to aggregate embedding and compute loss, and backward pass to derive the weight gradient $G_n^{(t)}$.

The forward pass propagates from the first layer to the last layer. We obtain layer embedding $H_{\mathcal{TG}}^{(t,l)}$ by concatenating the stale embedding of sampled boundary nodes $H_{\mathcal{B}_{\mathcal{TG}}}^{(t-\tau,l)}$ which were transmitted asynchronously ($thread_f^{t-\tau}$) in the $(t-\tau)$-th iteration, and the latest embedding of inner nodes $H_{\mathcal{I}_{\mathcal{TG}}}^{(t,l)}$ according to Eq. (3) (Lines 9-11). Also, the worker will pre-retrieve ($thread_f^t$) the embedding of the sampled boundary neighbors in Lines 12-14, which are used in the $(t+\tau)$-th iteration for embedding aggregation. Specifically, the first layer embedding $H_{\mathcal{B}_{\mathcal{SG}}}^{(t,0)}$ (features) can be obtained from local cache if the boundary node is sampled and cached. Besides, we have to fetch the embedding of subsequent layers $H_{\mathcal{B}_{\mathcal{SG}}}^{(t,l)}, l > 0$ from remote workers, which can not be cached in local memory due to their dynamically changing values. Using the outputs $Z_{\mathcal{I}_{\mathcal{TG}}}^{(t,L)}$ and labels $Y_{\mathcal{I}_{\mathcal{TG}}}$, we derive the loss value in Line 16. Note that the embedding transmission (communication) and aggregation (computation) *run concurrently* as they are executed by CPU together with NIC, and GPU, respectively.

Backward pass is conducted to acquire the gradient w.r.t. weight parameter from the end to front (Lines 17-25). In particular, worker n needs to calculate both the weight and embedding gradients alternatively (Lines 18-20), and also retrieves the gradient $J_{\mathcal{IC}_{\mathcal{CG}}}^{(t-\tau,l)}$ of sampled boundary nodes ahead of τ steps in line with Eq. (7) (Lines 21-23). At the same time, the worker will fetch $J_{\mathcal{IC}_{\mathcal{SG}}}^{(t,l)}$ from remote partitions in Lines 24-25, which is prepared for gradient calculation in the $(t+\tau)$-th iteration considering asynchronous training. All-reduce is performed on the gradients across all workers, and finally the weight parameter is updated (Lines 26-27).

C. Convergence Analysis

Now, we characterize the convergence behavior of PSC-GCN. Generally, the main difference between PSC-GCN and synchronous full-graph training lies in the stale statistics and neighbor sampling. In this respect, we first analyze the error and bias introduced by asynchronous training and graph sampling, which will aid PSC-GNN convergence analysis. Due to space limit, we omit the proofs and only show the results.

Lemma 1. *Considering full graph without sampling, the error introduced by asynchronous training is bounded:*

$$\|\tilde{\nabla}\mathcal{L}(\Theta^{(t)}) - \mathcal{L}(\Theta^{(t)})\|^2 \leq \eta^2\tau^2 C_a, \tag{10}$$

where $\tilde{\nabla}\mathcal{L}(\Theta^{(t)})$ denotes the gradient of using τ-step stale statistics and C_a is a finite constant.

Lemma 1 states that the error between asynchronous and synchronous full-graph training is quadratically proportional to the staleness τ. Next, we illustrate the sampling influence.

Lemma 2. *The error incurred by neighbor sampling satisfies*

$$\|G^{(t)} - \tilde{\nabla}\mathcal{L}(\Theta^{(t)})\|^2 \leq C_s, \tag{11}$$

where C_s implies a finite value.

In fact, Eq. (11) can be broken down into the bias $\|\mathbb{E}[G^{(t)}] - \tilde{\nabla}\mathcal{L}(\Theta^{(t)})\|^2$ and variance $\|G^{(t)} - \mathbb{E}[G^{(t)}]\|^2$, which both *depend on the sampling policy* but will not go to infinity [20]. On the basis of asynchronous and sampling errors, we provide the convergence rate of PSC-GCN.

Theorem 1. *Suppose the loss function is K-smooth and $\eta K \leq 1$, then we have:*

$$\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}[\|\nabla\mathcal{L}(\Theta^{(t)})\|^2] \leq \frac{2(\mathcal{L}(\Theta^{(0)}) - \mathbb{E}[\mathcal{L}(\Theta^{(T)})])}{\eta T} + 2\eta^2\tau^2 C_a + 2C_s. \tag{12}$$

Proof. Since the loss satisfies K-smoothness, we attain:

$$\begin{aligned}
&\mathbb{E}[\mathcal{L}(\Theta^{(t+1)})] \leq \mathbb{E}[\mathcal{L}(\Theta^{(t)})] + \mathbb{E}[\langle\nabla\mathcal{L}(\Theta^{(t)}), \Theta^{(t+1)} - \Theta^{(t)}\rangle] \\
&+ \frac{K}{2}\mathbb{E}[\|\Theta^{(t+1)} - \Theta^{(t)}\|^2] \\
&= \mathbb{E}[\mathcal{L}(\Theta^{(t)})] - \eta\mathbb{E}[\langle\nabla\mathcal{L}(\Theta^{(t)}), G^{(t)}\rangle] + \frac{\eta^2 K}{2}\mathbb{E}[\|G^{(t)}\|^2] \\
&= \mathbb{E}[\mathcal{L}(\Theta^{(t)})] + \frac{\eta}{2}\mathbb{E}[\|\nabla\mathcal{L}(\Theta^{(t)}) - G^{(t)}\|^2] \\
&- \frac{\eta}{2}\mathbb{E}[\|\nabla\mathcal{L}(\Theta^{(t)})\|^2] - \Big(\frac{\eta}{2} - \frac{\eta^2 K}{2}\Big)\mathbb{E}[\|G^{(t)}\|^2] \\
&\leq \mathbb{E}[\mathcal{L}(\Theta^{(t)})] + \frac{\eta}{2}\mathbb{E}[\|\nabla\mathcal{L}(\Theta^{(t)}) - G^{(t)}\|^2] - \frac{\eta}{2}\mathbb{E}[\|\nabla\mathcal{L}(\Theta^{(t)})\|^2],
\end{aligned} \tag{13}$$

where the last inequality is owing to $\eta K \leq 1$. Let us focus on the term $\frac{\eta}{2}\mathbb{E}[\|\nabla\mathcal{L}(\Theta^{(t)}) - G^{(t)}\|^2]$.

$$\begin{aligned}
&\frac{\eta}{2}\mathbb{E}[\|\nabla\mathcal{L}(\Theta^{(t)}) - G^{(t)}\|^2] \\
&= \frac{\eta}{2}\mathbb{E}[\|\nabla\mathcal{L}(\Theta^{(t)}) - \tilde{\nabla}\mathcal{L}(\Theta^{(t)}) + \tilde{\nabla}\mathcal{L}(\Theta^{(t)}) - G^{(t)}\|^2] \\
&\overset{a}{\leq} \eta\mathbb{E}[\|\nabla\mathcal{L}(\Theta^{(t)}) - \tilde{\nabla}\mathcal{L}(\Theta^{(t)})\|^2] + \eta\mathbb{E}[\|\tilde{\nabla}\mathcal{L}(\Theta^{(t)}) - G^{(t)}\|^2] \\
&\overset{b}{\leq} \eta^3\tau^2 C_a + \eta C_s,
\end{aligned} \tag{14}$$

in which $\overset{a}{\leq}$ is from Cauchy–Schwarz inequality and $\overset{b}{\leq}$ is according to Lemmas 1-2. Combining Eqs. (13) and (14), we obtain $\mathbb{E}[\mathcal{L}(\Theta^{(t+1)})] \leq \mathbb{E}[\mathcal{L}(\Theta^{(t)})] + \eta^3\tau^2 C_a + \eta C_s - \frac{\eta}{2}\mathbb{E}[\|\nabla\mathcal{L}(\Theta^{(t)})\|^2]$. Arrange the terms, then

$$\frac{\eta}{2}\mathbb{E}[\|\nabla\mathcal{L}(\Theta^{(t)})\|^2] \leq \mathbb{E}[\mathcal{L}(\Theta^{(t)})] - \mathbb{E}[\mathcal{L}(\Theta^{(t+1)})] + \eta^3\tau^2 C_a + \eta C_s.$$

Telescope both sides from $t = 0$ to $T-1$, and divide $\frac{\eta T}{2}$ to yield the final result in Eq. (12). □

Regarding Theorem 1, we have the following observations.

- The smoothness assumption of loss function is commonly adopted in previous works [17], [20]. Actually, this prerequisite can be derived from a series of *simpler conditions*.
- If we set the learning rate $\eta = \mathcal{O}(T^{-1/3})$, then the convergence will be $\mathcal{O}(T^{-2/3})$. As for the non-vanishing term $2C_s$, it will exist as long as *neighbor sampling* is leveraged to accelerate GCN training.

IV. Communication-Efficient Graph Sampling

This section is dedicated to design a *communication-efficient* sampling policy for reducing the whole *embedding estimation variance* of all layers.

A. Embedding Aggregation Variance

To begin with, we elaborate the estimation error introduced by graph sampling. Suppose that a boundary or inner node j is a one-hop neighbor of inner node i, i.e., $j \in \mathcal{N}_i$. Graph sampling is to *determine the probability* p_{ij} that neighbor j is sampled by node i in its embedding aggregation. To this end, define a random variable $\chi_j \sim \text{Bernoulli}(p_{ij})$ to indicate whether the neighbor is selected. Hence, we have an unbiased estimation [17] of i's intermediate embedding $Z_i^{(l+1)}$ as:

$$Z_i^{(l+1)} = \sum_{j\in\mathcal{N}_i} \chi_j W_{ij}^{(l+1)} P_{ij}^{(l)} H_j^{(l)} = \sum_{j\in\mathcal{N}_i} \tfrac{1}{p_{ij}}\chi_j W_{ij}^{(l+1)} P_{ij} H_j^{(l)}. \quad (15)$$

Here $P_{ij}^{(l)}$ is the rescaled propagation coefficient by multiplying $\frac{1}{p_{ij}}$, and $W_{ij}^{(l+1)}$ implies the corresponding weight parameter. Note that $Z_i^{(t,l+1)}$ in the t-th iteration will leverage *τ-step ahead* $H_j^{(t-\tau,l)}$ if j is a boundary neighbor and the *latest* embedding $H_j^{(t,l)}$ of inner node due to asynchronous training. In this section, we *omit the time index* t mainly for ease of exposition. As a result, the overall estimation variance of all L layers is obtained:

$$\begin{aligned}
\text{var}_i &= \mathbb{E}\Big[\Big\|\sum_{l=0}^{L-1}\Big(\sum_{j\in\mathcal{N}_i}\Big(\frac{1}{p_{ij}}\chi_j W_{ij}^{(l+1)} P_{ij} H_j^{(l)} - W_{ij}^{(l+1)} P_{ij} H_j^{(l)}\Big)\Big)\Big\|^2\Big] \\
&= \mathbb{E}\Big[\Big\|\sum_{l=0}^{L-1}\Big(\sum_{j\in\mathcal{N}_i}\Big(\frac{1}{p_{ij}}\chi_j - 1\Big) W_{ij}^{(l+1)} P_{ij} H_j^{(l)}\Big)\Big\|^2\Big] \\
&= \mathbb{E}\Big[\Big\|\sum_{j\in\mathcal{N}_i}\Big(\frac{1}{p_{ij}}\chi_j - 1\Big)\sum_{l=0}^{L-1} W_{ij}^{(l+1)} P_{ij} H_j^{(l)}\Big\|^2\Big] \\
&= \mathbb{E}\Big[\Big\|\sum_{j\in\mathcal{N}_i}\Big(\frac{1}{p_{ij}}\chi_j - 1\Big)\Big\|^2 \|\boldsymbol{y}_{ij}\|^2\Big] \\
&= \sum_{j\in\mathcal{N}_i}\Big(\frac{1}{p_{ij}} - 1\Big)\|\boldsymbol{y}_{ij}\|^2,
\end{aligned} \quad (16)$$

where $\boldsymbol{y}_{ij} = \big[W_{ij}^{(1)} P_{ij} H_j^{(0)}, \cdots, W_{ij}^{(L)} P_{ij} H_j^{(L-1)}\big]$ denotes the concatenation of aggregated embedding. Since latest $H_j^{(l)}$ can not be acquired in the sampling phase which is decided before the training in current iteration, a remedy is to use the embedding information from previous iterations, i.e., determine the sampled graph ahead for later GCN model updates.

Conditioned on the number of neighbor nodes $|\mathcal{N}_i|$, we consider $\sum_{j\in\mathcal{N}_i} p_{ij} = B_i$ with $B_i = \varepsilon|\mathcal{N}_i|, \varepsilon \in (0, 1]$, that is, node i will choose neighbors with sampling rate ε on average.

B. Communication-Efficient Variance Reduction

1) Variance Reduction Formulation: Our goal is to find the optimal sampling probabilities for neighbors $\mathcal{N}_i$ to minimize the embedding estimation variance, which can be casted as the following optimization problem:

$$\begin{aligned}
\min \quad & \overline{\text{var}}_i = \sum_{j\in\mathcal{N}_i} a_j\Big(\frac{1}{p_{ij}} - 1\Big)\|\boldsymbol{y}_{ij}\|^2 \\
\text{s.t.} \quad & \sum_{j\in\mathcal{N}_i} p_{ij} = B_i, 0 < p_{ij} \leq 1.
\end{aligned} \quad (17)$$

In particular, we introduce a factor a_j for rescaling var_i in Eq. (16) to $\overline{\text{var}}_i$, which is to *prioritize* local inner neighbors over remote boundary ones in graph sampling. To be more specific, inner nodes are associated with higher factor a_j and boundary nodes have lower a_j, so that we can weigh more on the inner variance. By doing so, the sampling probability of inner neighbors will be adjusted to a larger value, while that of boundary neighbors will be smaller, i.e., *cross-worker communication* volume is reduced.

2) Sampling Probability Solution: To solve Eq. (17), we construct its Lagrange function by combining the constraints:

$$\begin{aligned}
\Phi(\lambda, \boldsymbol{\alpha}, \boldsymbol{\beta}) = &\sum_{j\in\mathcal{N}_i} a_j\Big(\frac{1}{p_{ij}} - 1\Big)\|\boldsymbol{y}_{ij}\|^2 + \lambda\Big(\sum_{j\in\mathcal{N}_i} p_{ij} - B_i\Big) \\
&- \sum_{j\in\mathcal{N}_i}\alpha_j p_{ij} + \sum_{j\in\mathcal{N}_i}\beta_j\left(p_{ij} - 1\right),
\end{aligned} \quad (18)$$

where $\lambda, \alpha_j, \beta_j$ are Lagrange multipliers. Accordingly, the KKT conditions are characterized to derive the solution.

$$\begin{cases}
\frac{\partial\Phi}{\partial p_{ij}} = -\frac{a_j}{p_{ij}^2}\|\boldsymbol{y}_{ij}\|^2 + \lambda - \alpha_j + \beta_j = 0, \forall j \in \mathcal{N}_i, \\
\sum_{j\in\mathcal{N}_i} p_{ij} - B_i = 0, \\
\alpha_j p_{ij} = 0, \forall j \in \mathcal{N}_i, \\
\beta_j\left(p_{ij} - 1\right) = 0, \forall j \in \mathcal{N}_i.
\end{cases}$$

We present the following classified discussions.

- Since the probability $p_{ij} > 0$, it is obvious that $\alpha_j = 0$.
- If $\beta_j > 0$, then $p_{ij} = 1$, and $a_j\|\boldsymbol{y}_{ij}\|^2 = \lambda + \beta_j > \lambda$.
- If $\beta_j = 0$, then we have $p_{ij} = \frac{\sqrt{a_j}\|\boldsymbol{y}_{ij}\|}{\sqrt{\lambda}}$.

Putting them all together, there must exist a threshold λ^* that divides neighbor nodes $\mathcal{N}_i$ into two subsets: 1) $\{j : a_j\|\boldsymbol{y}_{ij}\|^2 < \lambda^*\}$ of size k with $p_{ij} = \frac{\sqrt{a_j}\|\boldsymbol{y}_{ij}\|}{\sqrt{\lambda^*}}$; 2) $\{j : a_j\|\boldsymbol{y}_{ij}\|^2 > \lambda^*\}$ of size $(|\mathcal{N}_i| - k)$ with $p_{ij} = 1$. The remaining issue is to find the threshold value, which is computed based on the fact that $\sum_{j=1}^{k}\frac{\sqrt{a_j}\|\boldsymbol{y}_{ij}\|}{\sqrt{\lambda^*}} + |\mathcal{N}_i| - k = B_i$.

Algorithm 2: SampleGraph

Input: Graph $\mathcal{G}$, target nodes $\mathcal{T}_{\mathcal{G}}$, sampling rate ε
Output: Subgraph $\mathcal{SG}$

1 $\boldsymbol{p} = \left[\boldsymbol{p}_1, \ldots, \boldsymbol{p}_{|\mathcal{T}_{\mathcal{G}}|}\right]$ where $\boldsymbol{p}_i = \left[p_{i1}, \ldots, p_{i|\mathcal{N}_i|}\right]$;
2 Initialize $p_{ij} = \varepsilon, \forall p_{ij} \in \boldsymbol{p}$;
3 Estimate $\boldsymbol{y}_{ij}$ based on historical information;
4 **for** $i \in \mathcal{T}_{\mathcal{G}}$ **do**
5 **for** $j \in \mathcal{N}_i$ **do**
6 $B_i = \varepsilon \times |\mathcal{N}_i|$;
7 Assign each sampling probability $p_{ij} \in \boldsymbol{p}_i$ according to Eq. (19);
8 Sample a subgraph $\mathcal{SG}$ obeying to $\boldsymbol{p}$;
9 **return** $\mathcal{SG}$

Theorem 2. *Suppose that the neighbor nodes $\mathcal{N}_i$ are sorted in an ascending order by $a_j \left\|\boldsymbol{y}_{ij}\right\|^2$. Let k be the largest integer that satisfies $B_i + k - |\mathcal{N}_i| \leq \frac{\sum_{j=1}^{k} \sqrt{a_j}\|\boldsymbol{y}_{ij}\|}{\sqrt{a_k}\|\boldsymbol{y}_{ik}\|}$, then we can set $\sqrt{\lambda^*} = \frac{\sum_{j=1}^{k} \sqrt{a_j}\|\boldsymbol{y}_{ij}\|}{B_i + k - |\mathcal{N}_i|}$, and the sampling probability is*

$$p_{ij} = \begin{cases} (B_i + k - |\mathcal{N}_i|)\frac{\sqrt{a_j}\|\boldsymbol{y}_{ij}\|}{\sum_{l=1}^{k} \sqrt{a_l}\|\boldsymbol{y}_{il}\|} & \text{if } j \leq k, \\ 1 & \text{if } j > k. \end{cases} \tag{19}$$

One can substitute the probability in Eq. (19) into the KKT conditions to verify its correctness. Besides, when $B_i \leq \frac{\sum_{j=1}^{|\mathcal{N}_i|} \sqrt{a_j}\|\boldsymbol{y}_{ij}\|}{\sqrt{a_{|\mathcal{N}_i|}}\left\|\boldsymbol{y}_{i|\mathcal{N}_i|}\right\|}$, Eq. (19) will boil down to the importance sampling with $p_{ij} = B_i \frac{\sqrt{a_j}\|\boldsymbol{y}_{ij}\|}{\sum_{l=1}^{|\mathcal{N}_i|} \sqrt{a_l}\|\boldsymbol{y}_{il}\|}$. In this respect, we compare the theoretical rescaled variances of the importance sampling and conventional random sampling to show the superiority of PSC-GCN.

Lemma 3. *Denote $\overline{\mathrm{var}}_i^{im}$ as the rescaled estimation variance of importance sampling, and $\overline{\mathrm{var}}_i^{rd}$ as that of random sampling $\boldsymbol{p}^{rd} = \left[p_1^{rd}, \cdots, p_{|\mathcal{N}_i|}^{rd}\right]$ with $p_j^{rd} = \frac{B_i}{|\mathcal{N}_i|}$. Then, the variance difference satisfies:*

$$\overline{\mathrm{var}}_i^{rd} - \overline{\mathrm{var}}_i^{im} = \frac{|\mathcal{N}_i|}{B_i} \sum_{j \in \mathcal{N}_i} a_j \left\|\boldsymbol{y}_{ij}\right\|^2 - \frac{1}{B_i}\Big(\sum_{j=1}^{|\mathcal{N}_i|} \sqrt{a_j}\left\|\boldsymbol{y}_{ij}\right\|\Big)^2. \tag{20}$$

Since Eq. (20) is non-negative from Cauchy–Schwarz inequality, this lemma implies that our variance-reduction based sampling *yields lower error* than vanilla sampling policy.

C. Sampling Algorithm Description

Algorithm 2 describes the communication-efficient sampling policy for variance reduction. Concretely, inputs for Line 3 in PSC-GCN, i.e., Algorithm 2, are $(\mathcal{G}, \mathcal{T}_{\mathcal{G}}, \varepsilon) = (\mathcal{G}_n, \mathcal{I}_{\mathcal{G}_n}, \varepsilon)$ in line with local partition of any worker n. Moreover, each p_{ij} can be regarded as the probability that an edge from node j to node i is sampled, and hence the computational complexity of Algorithm 2 is $\mathcal{O}(|\mathcal{E}_{\mathcal{G}_n}|)$ for worker n.

To alleviate the computation workload brought by our sampling policy, Algorithm 2 in PSC-GCN could be implemented in a *semi-dynamic fashion*. That is, the sampling probability $\boldsymbol{p}$ is updated every certain iterations (Lines 4-7) to amortize the sampling complexity and ensure fairly low estimation error. Consequently, PSC-GCN will always hinge on the latest probability $\boldsymbol{p}$ to sample subgraph in Line 8.

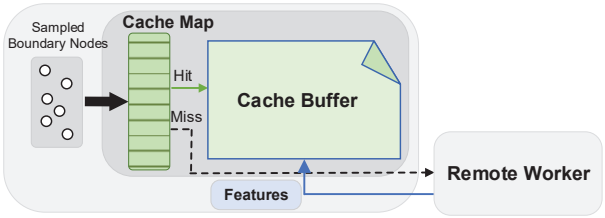


Fig. 5: Cache workflow.

V. Inclusion-aware Feature Caching

In this section, we elucidate the caching policy design.

A. Workflow of Cache Module

PSC-GCN would iteratively sample a subgraph for neighbor aggregation and model update. As cross-worker communication is the major bottleneck in graph sampling, we deploy a cache on each worker GPU to store a small portion of boundary nodes, such that their features are directly accessed without retrieving through LAN if sampled, as shown in Fig. 5. In particular, our cache module consists of a cache map and a cache buffer, where the map is designed to record whether a node is cached in the buffer.

During GCN training, when a worker needs to retrieve the statistics of sampled boundary nodes, it first checks the cache map to verify whether they are stored in the buffer, as described by Line 13 in Algorithm 1. Once the nodes are hit in cache map, the worker fetches their indices and reads the features from local cache buffer. Otherwise, their information is obtained from remote workers. Note that while the cache map introduces additional overhead, it is still lightweight in terms of memory consumption considering that mapping 1M nodes actually requires less than 5MB GPU memory.

B. Cache Update Policy

Caching efficiency is often quantified by the cache hit rate, i.e., the probability that a boundary node is sampled and also buffered in local memory. To improve the efficiency, cache update is desired to be co-designed with the sampling scheme so as to balance the update overhead and node hit rate, which however is largely ignored in existing literatures [18], [24]. In general, dynamically refreshing the cache buffer may promote the hit rate, but often comes at the cost of high computation and communication consumptions as GPUs are less efficient at complex operations. While pure static caching policies may incur low overhead, they usually can not ensure a favorable hit rate, wasting the precious GPU memory. To reconcile their conflicts, we employ a semi-dynamic cache update policy along with the graph sampling as aforementioned.

Consistent with the neighbor sampling probability in Eq. (19), the *inclusion probability* that a boundary node j is picked by worker n for neighbor aggregation is $p_j \triangleq 1 - \prod_{i \in \mathcal{N}_j \cap \mathcal{I}_{\mathcal{G}_n}} (1 - p_{ij})$. Apparently, boundary nodes with

TABLE II: Test accuracy results.

Method	Reddit	Ogbn-products	Yelp
GraphSAGE	95.03%	76.49%	64.13%
GraphSAINT	96.25%	76.49%	64.62%
Full-graph	97.01%	78.94%	65.12%
PipeGCN	97.07%	79.69%	65.14%
PSC-GCN ($\tau = 2, \varepsilon = 1$)	96.96%	79.67%	65.17%
PSC-GCN ($\tau = 1, \varepsilon = 0.1$)	97.15%	80.73%	65.33%
PSC-GCN ($\tau = 2, \varepsilon = 0.1$)	97.21%	80.84%	65.28%
PSC-GCN ($\tau = 2, \varepsilon = 0.01$)	96.80%	75.73%	65.33%

higher inclusion probability will be more likely to be sampled, implying caching those nodes conditioned on the buffer size can result in a higher hit rate. Since the sampling probability will change in a semi-dynamic manner, we synchronize the cache update cycle accordingly to renew the cache buffer.

VI. Performance Evaluation

In this section, we evaluate the performance of PSC-GCN, involving the asynchronous training, communication-efficient sampling, and inclusion-aware caching.

A. Evaluation Setup

1) Platform Setup: We conduct PSC-GCN on a GPU cluster with three physical servers, where each is equipped with two NVIDIA T4 GPUs (16GB) and 16vCPU cores (64GB). CPU-GPU and GPU-GPU within a machine are connected by PCIe3x16, while the servers are inter-linked by LAN with 15Gbps bandwidth. PSC-GCN is implemented with over 3000 lines of Python codes based on Pytorch and DGL. Each worker trains GCN on a single GPU.

2) Datasets and GCN Models: We mainly use three datasets to perform the evaluations, i.e., Reddit [1], Ogbn-products [19], and Yelp [15]. Following the convention, we leverage a 4-layer GCN model with 256 and 512 hidden units in each layer on Reddit and Yelp, respectively, and a 3-layer model with 128 hidden units on Ogbn-products. Besides, the scaling factor is set to $a_j = 1.1$ for inner nodes and $a_j = 1$ for boundary nodes to distinguish them in the graph sampling.

3) Baselines: For comparisons, we introduce benchmarks regarding asynchronous, sampling and caching schemes.

- PipeGCN: A one-step asynchronous training algorithm on full graphs without the neighbor sampling [17].
- GraphSAGE: Random node-wise sampling policy to generate the node embedding [1].
- GraphSAINT: Sub-graph sampling based inductive learning method to ensure dense node connections [15].
- PaGraph: Degree-based static feature caching policy [18].

B. Basic Results

First, we show the basic results of PSC-GCN and baselines.

1) Model Accuracy: To verify the performance, we compare the test accuracy of PSC-GCN and benchmarks on Reddit, Ogbn-product and Yelp in Table II. We can observe that *small-step* asynchronous training only results in a *negligible* accuracy decline when using full graphs, roughly by 0.05% on Reddit. Even with graph sampling, PSC-GCN with staleness $\tau = 1, 2,$ and sampling rate $\varepsilon = 0.1$ still achieves a similar performance to synchronous full-graph training on all datasets.

TABLE III: Training time (s) per iteration and for convergence.

Method	Time: per-iteration and convergence		
	Reddit	Ogbn-products	Yelp
GraphSAGE	0.9023	0.7978	0.9749
	902.32	279.23	1657.30
Full-graph	1.5974	1.3005	1.8973
	1517.53	487.69	2845.95
PipeGCN	1.1434	0.7163	1.238
	1257.74	286.52	1858.05
PSC-GCN ($\tau = 2, \varepsilon = 1$)	1.0730	0.7043	1.2293
	1233.95	281.72	1843.95
PSC-GCN ($\tau = 1, \varepsilon = 0.1$)	0.6259	0.3637	0.5496
	876.26	163.67	824.40
PSC-GCN ($\tau = 2, \varepsilon = 0.1$)	0.6141	0.3416	0.4776
	859.74	136.64	716.40
PSC-GCN ($\tau = 2, \varepsilon = 0.01$)	0.2521	0.2414	0.4148
	302.52	168.98	622.20

Sampling rate ε also affects the model accuracy, which decreases sharply by 5% on Ogbn-products when ε ranges from 0.1 to 0.01. Nonetheless, the model performance will be at the same-level as synchronous full-graph training when ε *reaches 0.1*, as PSC-GCN ($\tau = 1, 2, \varepsilon = 0.1$) even leads to a higher model accuracy. Besides, PSC-GCN outperforms other sampling-based benchmarks, with 1%-4% higher accuracy than GraphSAGE and GraphSAINT on three datasets, since we can effectively mitigate the embedding estimation error.

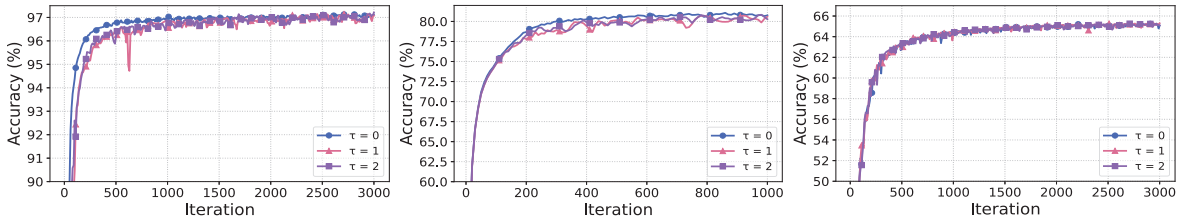
2) Training Acceleration: One of the main targets of PSC-GCN is to expedite GCN training, and hence we present the per-iteration and convergence time in Table III. Our proposed PSC-GCN significantly outperforms benchmarks by achieving *4.6-6.3 times* per-iteration acceleration compared to vanilla full-graph scheme, and *2.4-4.5 times* compared to PipeGCN and GraphSAGE (GraphSAINT is unsuitable for distributed implementation). Compared to one-step asynchronous PSC-GCN ($\tau = 1, \varepsilon = 0.1$), the two-step case ($\tau = 2$) reduces per-iteration time by 6%-13%. Moreover, the convergence time verifies that PSC-GCN can *speed up training* by 3.6-5.0 times, i.e., reducing wall-clock time for convergence by 72%-80%.

C. Ablation Study

We continue to validate PSC-GCN through ablation study.

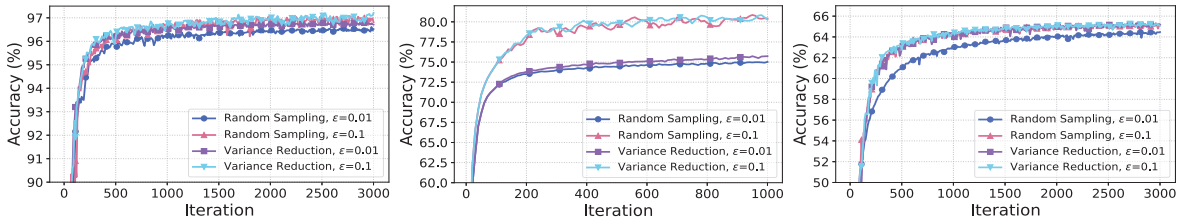
1) Asynchronous Training: Fig. 6 illustrates asynchronous training with a fixed sampling rate 0.1 while various statistics staleness τ. Although the synchronous case ($\tau = 0$) results in faster statistical convergence regarding training iterations, the final test accuracy given different τ would approach a *similar value*, i.e., around 97%, 80%, 65% on three datasets, respectively. That is, small staleness has negligible impact on model accuracy, but can significantly mitigate the training time by 33%-38% on Ogbn-products and Yelp, respectively, suggesting the robustness of PSC-GCN.

2) Graph Sampling: Fig. 7 displays the test accuracy under different sampling policies and rates, given two-step asynchronization. Notably, our variance-reduction based sampling leads to a *1.23 times faster* convergence than random sampling in GraphSAGE. Also, we can achieve 0.5%-1.5% higher



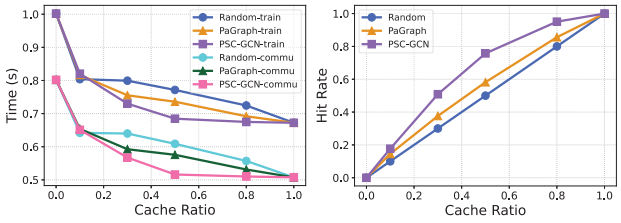
(a) Reddit (b) Ogbn-products (c) Yelp

Fig. 6: Comparison results of different asynchronization staleness τ on Reddit, Ogbn-products, and Yelp.



(a) Reddit (b) Ogbn-products (c) Yelp

Fig. 7: Comparison results of sampling polices with different sampling rate ε on Reddit, Ogbn-products and Yelp.



(a) Training time (b) Cache hit rate

Fig. 8: Comparison results of different caching policies.

accuracy under the same sampling rate. Both corroborate the effectiveness of our proposed sampling policy, making it a favorable choice for efficient GCN training.

3) Feature Caching: At last, we discuss the caching results on Reddit (Ogbn-products and Yelp are similar) compared with random caching and PaGraph, provided 0.1 sampling rate. As depicted in Fig. 8(a), we can reduce the training time by *10% (4%)* compared to random policy (PaGraph) when buffering 30% boundary nodes, as cross-worker transmission is alleviated. Besides, Fig. 8(b) shows cache hit rate, where we always outperform baselines with *7%-25% and 3%-17%* higher hit, respectively, i.e., our caching policy is more efficient.

VII. Related Work

GCN training. GCNs have emerged as state-of-the-art approaches for many graph-related learning tasks [25]. However, training GCN models are time-consuming due to the high communication overhead [26]. Considering that real-world graph can be huge, DistDGL is developed to build graph models in a distributed manner [27]. To handle full graphs, Wan *et al*. design BNS-GCN to randomly sample boundary nodes, which enhances the GCN training efficiency and scalability [28]. Besides, PipeGCN is proposed to asynchronously retrieve boundary nodes one-step ahead for communication mitigation [17]. These works have made great progresses in distributed GCN training, while the communication is still high when workers are linked by slow-rate network.

Graph sampling. Sampling a subgraph in centralized or distributed GCN can reduce the communication and computation workload [1]. To avoid neighbor explosion, layer-wise sampling is proposed to choose nodes in each layer conditioned on the top one, so as to allow for a fixed-sized sampling [29]. Moreover, subgraph-wise sampling is designed to tackle the connectivity problem encountered by layer-wise methods [15]. Since gradient variance and error are introduced by graph sampling, a variance-reduction guided sampling is characterized for centralized training [30]. Nonetheless, how to explicitly achieve low embedding estimation error while catering for the communication cost is still unexplored.

Feature caching. Recent attentions have been paid to caching node features to reduce the communication cost [24]. PaGraph is a static caching policy that buffers nodes with high out-degrees to mitigate data fetching time [18]. To improve cache hit rate, Liu *et al*. develop a distributed training system with dynamic caching and graph partition scheme to alleviate data preparation cost [31]. Furthermore, a two-level caching policy by using CPU and GPU memory to buffer node features is proposed in [32], where nodes are dynamically evicted from local memories. Existing caching policies often work standalone without a tight co-design with the graph sampling.

VIII. Conclusion

In this paper, we propose pipelined PSC-GCN to address the communication bottleneck in distributed GCN training. Specifically, we fetch the sampled boundary nodes in an asynchronous manner which can hide the statistics transmission into local model computation. To alleviate the adverse impact of asynchronous staleness, we design an efficient sampling policy to reduce the communication volume and embedding estimation error. In addition, we enhance graph sampling by caching a small portion of boundary nodes in local memory, further optimizing the data preparation time. Real-world experiments show that PSC-GCN outperforms existing benchmarks by accelerating the training convergence by 3.6-5.0 times.

REFERENCES

[1] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," in *Proc. NIPS*, 2017.

[2] A. Fout, J. Byrd, B. Shariat, and A. B. Hur, "Protein Interface Prediction Using Graph Convolutional Networks," in *Proc. NIPS*, 2017, pp. 6530–6539.

[3] Z. Cui, K. Henrickson, R. Ke, and Y. Wang, "Traffic Graph Convolutional Recurrent Neural Network: A Deep Learning Framework for Network-Scale Traffic Learning and Forecasting," *IEEE TITS*, vol. 21, no. 11, pp. 4883-4894, 2019.

[4] H. Wang, F. Zhang, M. Zhang, J. Leskovec, M. Zhao, W. Li, and Z. Wang, "Knowledge-aware Graph Neural Networks with Label Smoothness Regularization for Recommender Systems," in *Proc. ACM KDD*, 2019, pp. 968–977.

[5] Z. Wu, S. Pan, F. Chen, G. Long, and C. Zhang, and P. S. Yu, "A Comprehensive Survey on Graph Neural Networks," *IEEE TNNLS*, vol. 32, no. 1, pp. 4-24, 2021.

[6] T. N. Kipf, and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," in *Proc. ICLR*, 2017.

[7] S. A. E. Haija, A. Kapoor, B. Perozzi, and J. Lee, "N-GCN: Multi-scale Graph Convolution for Semi-supervised Node Classification," *PMLR*, vol. 115, pp. 841-851, 2020.

[8] K. Lei, M. Qin, B. Bai, G. Zhang, and M. Yang, "GCN-GAN: A Non-linear Temporal Link Prediction Model for Weighted Dynamic Networks," in *Proc. IEEE INFOCOM*, 2019, pp. 388-396.

[9] Y. Ma, S. Wang, C. C. Aggarwal, and J. Tang, "Graph Convolutional Networks with EigenPooling," in *Proc. ACM KDD*, 2019, pp. 723–731.

[10] C. Zheng, H. Chen, Y. Cheng, Z. Song, Y. Wu, C. Li, J. Cheng, H. Yang, and S. Zhang, "ByteGNN: Efficient Graph Neural Network Training at Large Scale," *ACM VLDB*, vol. 15, no. 6, pp. 1228–1242, 2022.

[11] S. Gandhi, and A. P. Iyer, "P3: Distributed Deep Graph Learning at Scale," in *Proc. USENIX OSDI*, 2021, pp. 551-568.

[12] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *ACM VLDB*, vol. 11, no. 4, pp. 420–431, 2017.

[13] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph Convolutional Neural Networks for Web-Scale Recommender Systems," in *Proc. ACM KDD*, 2018, pp. 974-983.

[14] D. Zou, Z. Hu, Y. Wang, S. Jiang, Y. Sun, and Q. Gu, "Layer-Dependent Importance Sampling for Training Deep and Large Graph Convolutional Networks," in *Proc. NeurIPS*, 2019.

[15] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graph-SAINT: Graph Sampling Based Inductive Learning Method," in *Proc. ICLR*, 2020.

[16] X. Wan, K. Chen, and Y. Zhang, "DGS: Communication-Efficient Graph Sampling for Distributed GNN Training," in *Proc. IEEE ICNP*, 2022, pp. 1-11.

[17] C. Wan, Y. Li, C. R. Wolfe, A. Kyrillidis, N. S. Kim, and Y. Lin. "PipeGCN: Efficient Full-Graph Training of Graph Convolutional Networks with Pipelined Feature Communication," in *Proc. ICLR*, 2022.

[18] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, "PaGraph: Scaling GNN Training on Large Graphs via Computation-aware Caching," in *Proc. ACM SoCC*, 2020, pp. 401–415.

[19] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open Graph Benchmark: Datasets for Machine Learning on Graphs," in *Proc. NeurIPS*, 2020, pp. 22118-22133.

[20] W. Cong, M. Ramezani, and M. Mahdavi, "On the Importance of Sampling in Training GCNs: Tighter Analysis and Variance Reduction," 2021, [Online]. Available: https://arxiv.org/abs/2103.02696

[21] G. Karypis, and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graph," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359-392, 1998.

[22] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks," 2019, [Online]. Available: https://arxiv.org/abs/1909.01315

[23] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling," in *Proc. ICLR*, 2018.

[24] J. Yang, D. Tang, X. Song, L. Wang, Q. Yin, R. Chen, W. Yu, and J. Zhou, "GNNLab: a factored system for sample-based GNN training over GPUs," in *Proc. ACM EuroSys*, 2022, pp. 417-434.

[25] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Berg, I. Titov, and M. Welling, "Modeling Relational Data with Graph Convolutional Networks," in *Proc. ESWC*, 2018, pp. 593-607.

[26] D. Zhang, X. Huang, Z. Liu, J. Zhou, Z. Hu, X. Song, Z. Ge, L. Wang, Z. Zhang, and Y. Qi, "AGL: A Scalable System for Industrial-purpose Graph Machine Learning," *ACM VLDB*, vol. 13, no. 12, pp. 3125–3137, 2020.

[27] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, "DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs," in *Proc. IEEE/ACM IA3*, 2020, pp. 36-44.

[28] C. Wan, Y. Li, A. Li, N. S. Kim, and Y. Lin, "BNS-GCN: Efficient Full-Graph Training of Graph Convolutional Networks with Partition-Parallelism and Random Boundary Node Sampling," in *Proc. MLSys*, 2022, pp. 673-693.

[29] W. Huang, T. Zhang, Y. Rong, and J. Huang, "Adaptive Sampling Towards Fast Graph Representation Learning," in *Proc. NeurIPS*, 2018.

[30] W. Cong, R. Forsati, M. Kandemir, and M. Mahdavi, "Minimal Variance Sampling with Provable Guarantees for Fast Training of Graph Neural Networks," in *Proc. ACM KDD*, 2020, pp. 1393-1403.

[31] T. Liu, Y. Chen, D. Li, C. Wu, Y. Zhu, J. He, Y. Peng, H. Chen, H. Chen, and C. Guo, "BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing," in *Proc. USENIX NSDI*, 2023, pp. 103-118.

[32] Z. Zhang, Z. Luo, and C. Wu, "Two-level Graph Caching for Expediting Distributed GNN Training," in *Proc. IEEE INFOCOM*, 2023.