Journal of Parallel and Distributed Computing 147 (2021) 166–183

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Accurate, efficient and scalable training of Graph Neural Networks

Hanqing Zeng [a,*,1], Hongkuan Zhou [a,1], Ajitesh Srivastava [a], Rajgopal Kannan [b], Viktor Prasanna [a]

[a] University of Southern California, Los Angeles, CA, United States of America
[b] US Army Research Lab, Los Angeles, CA, United States of America

ARTICLE INFO

Article history:
Received 18 March 2020
Received in revised form 1 August 2020
Accepted 25 August 2020
Available online 16 September 2020

Keywords:
Graph representation learning
Graph Neural Networks
Graph sampling
Graph partitioning
Memory optimization

ABSTRACT

Graph Neural Networks (GNNs) are powerful deep learning models to generate node embeddings on graphs. When applying deep GNNs on large graphs, it is still challenging to perform training in an efficient and scalable way. We propose a novel parallel training framework. Through sampling small subgraphs as minibatches, we reduce training workload by orders of magnitude compared with state-of-the-art minibatch methods. We then parallelize the key computation steps on tightly-coupled shared memory systems. For graph sampling, we exploit parallelism within and across sampler instances, and propose an efficient data structure supporting concurrent accesses from samplers. The parallel sampler theoretically achieves near-linear speedup with respect to number of processing units. For feature propagation within subgraphs, we improve cache utilization and reduce DRAM traffic by data partitioning. Our partitioning is a 2-approximation strategy for minimizing the communication cost compared to the optimal. We further develop a runtime scheduler to reorder the training operations and adjust the minibatch subgraphs to improve parallel performance. Finally, we generalize the above parallelization strategies to support multiple types of GNN models and graph samplers. The proposed training outperforms the state-of-the-art in scalability, efficiency and accuracy simultaneously. On a 40-core Xeon platform, we achieve 60× speedup (with AVX) in the sampling step and 20× speedup in the feature propagation step, compared to the serial implementation. Our algorithm enables fast training of deeper GNNs, as demonstrated by orders of magnitude speedup compared to the Tensorflow implementation. We open-source our code at https://github.com/GraphSAINT/GraphSAINT

1. Introduction

Graph embedding is a powerful dimensionality reduction technique to facilitate downstream graph analytics. The embedding process converts graph nodes with unstructured neighbor connections into points in a low-dimensional vector space. Embedding is essential for a wide range of tasks such as content recommendation [27], traffic forecasting [28], image recognition [5] and protein function prediction [7]. Among the various embedding techniques, Graph Neural Networks (GNNs) (including Graph Convolutional Network (GCN) [12] and its variants [4,7,23,30]) have attained much attention. GNNs produce accurate and robust embedding without the need of manual feature selection.

On large graphs, GNN training proceeds in the unit of minibatches. Due to edge connections, the graph nodes are not I.I.D distributed, and thus cannot be sampled uniformly at random as minibatch data points. State-of-the-art methods construct minibatches by sampling on each GNN layer (i.e., *layer sampling*). The vanilla GCN [12] and its successor GraphSAGE [7] sample by tracking down the inter-layer connections. Their approaches preserve the training accuracy of the original model, but the parallel training is not work-efficient due to a phenomenon often referred to as "neighbor explosion" [4,6,7]. Namely, for every additional GNN layer traversed by their samplers, the number of sampled nodes (i.e., neighbors) grows by an order of magnitude. Consequently, the sampled nodes across different minibatches overlap significantly, especially at the first few GNN layers. The amount of redundant computation thus increases exponentially with the number of GNN layers. To alleviate such high redundancy, FastGCN [4] proposes to independently sample the nodes of each GNN layer, without explicitly considering the layer connection constraint. Although FastGCN is faster than [7,12], it incurs significant accuracy loss and requires preprocessing on the full graph which is expensive and not easily parallelizable.

Due to the layer sampling design philosophy, it is difficult for state-of-the-art methods [4,7,12] to simultaneously achieve accuracy, efficiency and scalability. In this work, we perform sampling

* Corresponding author.
E-mail addresses: zengh@usc.edu (H. Zeng), hongkuaz@usc.edu (H. Zhou), ajiteshs@usc.edu (A. Srivastava), rajgopal.kannan.civ@mail.mil (R. Kannan), prasanna@usc.edu (V. Prasanna).
[1] Equal contribution.

https://doi.org/10.1016/j.jpdc.2020.08.011
0743-7315/

on the graph rather than the GNN layers. Our novelty lies in proposing a *graph sampling*-based minibatch training algorithm via joint optimization on the learning quality and parallelization cost. We achieve scalability by (1) Developing a novel data structure that enables efficient subgraph sampling through supporting fast parallel updates on the sampling probability; (2) Optimizing parallel execution of intra-subgraph feature propagation and layer-wise weight updates — specifically a cache-efficient subgraph partitioning scheme that guarantees near-minimal DRAM traffic. Optimization in the above two steps can be generalized to support multiple kinds of GNN models and sampling algorithms. We achieve work-efficiency by avoiding "neighbor explosion", as each layer of our minibatched GNN contains the same number of neurons corresponding to the subgraph nodes. Finally, we achieve learning accuracy since our sampled subgraphs preserve connectivity characteristics of the original training graph. The main contributions of this paper are:

- We propose a parallel GNN training algorithm based on graph sampling:
  - *Accuracy* is achieved since the sampler returns small, representative subgraphs of the original graph.
  - *Efficiency* is optimized since we always build complete GNNs on the minibatch subgraphs to avoid "neighbor explosion" in deeper layers.
  - *Scalability* is achieved with respect to number of processing cores, graph size and GNN depth by parallelizing various key steps.
- We propose a novel data structure that supports fast, incremental and parallel updates to a probability distribution. Our parallel sampler based on this data structure theoretically and empirically achieves near-linear scalability with respect to number of processing units.
- We parallelize all the key operations to scale the overall minibatch training to a large number of processing cores. Specifically, for subgraph feature propagation, we perform intelligent partitioning along the feature dimension to achieve close-to-optimal DRAM and cache performance.
- We propose a runtime scheduling algorithm for training:
  - By rearranging the order of various operations, we significantly reduce the training time under a wide range of model configurations.
  - By partition scheduling and node clipping of subgraphs, we improve the feature propagation performance by better cacheline alignment.
- We show that our parallelization and scheduling techniques are applicable to a number of GNN architectures (including graph convolution and graph attention) and graph sampling algorithms (including random edge sampling and variants of random walk sampling).
- We perform thorough evaluation on a 40-core Xeon server. Compared with serial implementation, we achieve 15× overall training time speedup. Compared with state-of-the-art minibatch methods, our training achieves up to 7.8× speedup without accuracy loss.
- Our parallel training greatly facilitates development of deeper GNN models on larger graphs. We achieve two orders of magnitude speedup for 3-layer GNNs compared to state-of-the-art Tensorflow implementation.

2. Background and related work

Graph Neural Networks (GNNs), including Graph Convolutional Network (GCN) [12], GraphSAGE [7] and Graph Attention Network (GAT) [23], are the state-of-the-art deep learning models for graph embedding. They have been widely shown to learn highly accurate and robust representations of the graph nodes. Like CNNs, GNNs belong to a type of multi-layer neural network, which performs node embedding as follows. The input to a GNN is a graph whose each node is associated with a feature vector (i.e., node attribute). The GNN propagates the features of each node layer by layer, where each layer performs tensor operations based on the model weights and the input graph topology. The last GNN layer outputs embedding vectors for each node of the input graph. Essentially, both the input node attributes and the topological information of the graph are "embedded" into the output vectors.

2.1. Forward and backward propagation

In this paper, we mainly consider four types of widely used GNNs: Graph Convolutional Network (GCN) [12], GraphSAGE [7], MixHop [1] and Graph Attention Network (GAT) [23]. We first introduce in detail the GraphSAGE model architecture, and then summarize the layer operations of the other three.

Let the input graph be $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{X})$, where $\boldsymbol{X} \in \mathbb{R}^{|\mathcal{V}| \times f}$ stores the initial node attributes, and f is the initial feature length. A GraphSAGE layer aggregates signals of nodes $\mathcal{V}$ along the edges $\mathcal{E}$. A full GraphSAGE network is build by stacking multiple layers, where the inputs to the next layer are the outputs of the previous one. We use superscript "(ℓ)" to denote GNN layer-ℓ parameters. For a layer ℓ, it contains $|\mathcal{V}|$ nodes corresponding to the graph nodes. Each input and output node of the layer is associated with a feature vector of length $f^{(\ell-1)}$ and $f^{(\ell)}$, respectively. Denote $\boldsymbol{X}^{(\ell-1)} \in \mathbb{R}^{|\mathcal{V}| \times f^{(\ell-1)}}$ and $\boldsymbol{X}^{(\ell)} \in \mathbb{R}^{|\mathcal{V}| \times f^{(\ell)}}$ as the input and output feature matrices of the layer, where $\boldsymbol{X}^{(0)} = \boldsymbol{X}$ and $f^{(0)} = f$. A layer input node $v^{(\ell-1)}$ is connected to a layer output node $u^{(\ell)}$ if and only if $(v, u) \in \mathcal{E}$. If we view the input and output nodes as a bipartite graph, then the bi-adjacency matrix $\boldsymbol{A}^{(\ell)}$ equals the adjacency matrix $\boldsymbol{A}$ of $\mathcal{G}$.

Each GraphSAGE layer contains two learnable weight matrices: self-weight $\boldsymbol{W}_\circ$ the neighbor-weight $\boldsymbol{W}_\star$. The forward propagation of a layer is defined by:

$$\boldsymbol{X}^{(\ell)} = \texttt{ReLU}\left(\tilde{\boldsymbol{A}} \cdot \boldsymbol{X}^{(\ell-1)} \cdot \boldsymbol{W}_\star^{(\ell)} \,\middle\|\, \boldsymbol{X}^{(\ell-1)} \cdot \boldsymbol{W}_\circ^{(\ell)}\right) \tag{1}$$

where "$\|$" is the column-wise matrix concatenation operation, and $\tilde{\boldsymbol{A}}$ is the normalized adjacency matrix. The normalization can be calculated as $\tilde{\boldsymbol{A}} = \boldsymbol{D}^{-1} \cdot \boldsymbol{A}$, where $\boldsymbol{A}$ is the binary adjacency matrix of $\mathcal{G}$ and $\boldsymbol{D}$ is the diagonal degree matrix of $\boldsymbol{A}$ (i.e., $D_{ii} = \texttt{deg}(i)$).

From Eq. (1), each layer performs two key operations:

1. *Feature aggregation*: Each layer-ℓ node collects features of its layer-$(\ell-1)$ neighbors and then calculates the weighted sum, as shown by $\tilde{\boldsymbol{A}} \cdot \boldsymbol{X}^{(\ell-1)}$.
2. *Weight transformation*: The aggregated neighbor features are multiplied by $\boldsymbol{W}_\star^{(\ell)}$. The features of a layer-$(\ell-1)$ node itself are multiplied by $\boldsymbol{W}_\circ^{(\ell)}$.

After obtaining the node embedding from the outputs of the last GNN layer, we can further perform various downstream tasks by analyzing the embedding vectors. For example, we can use a simple Multi-Layer Perceptron (MLP) to classify the graph nodes into C classes. Let L be the total number of GNN layers. So $\boldsymbol{X}^{(L)}$ is the final node embedding. Following the design of [4,7,9], the classifier MLP generates the node prediction by:

$$\begin{aligned} \boldsymbol{X}_{\text{MLP}} &= \texttt{ReLU}\left(\boldsymbol{X}^{(L)} \cdot \boldsymbol{W}_{\text{MLP}}\right) \\ \boldsymbol{Y} &= \sigma\left(\boldsymbol{X}_{\text{MLP}}\right) \end{aligned} \tag{2}$$

where $\boldsymbol{W}_{\text{MLP}} \in \mathbb{R}^{f^{(L)} \times C}$. Function $\sigma(\cdot)$ is the element-wise sigmoid or row-wise softmax to generate the probability of a node belonging to a class.

Under the supervised learning setting, each node of $\mathcal{V}$ is also provided with the ground-truth class label(s). Let $\overline{\boldsymbol{Y}} \in \mathbb{R}^{|\mathcal{V}| \times C}$ be the binary matrix encoding the ground-truth labels. Comparing the prediction with the ground-truth, we can obtain a scalar loss value, $\mathcal{L}$, by cross-entropy (CE):

$$\mathcal{L} = \texttt{CE}\left(\boldsymbol{Y}, \overline{\boldsymbol{Y}}\right) \tag{3}$$

For the other three types of GNNs under consideration, we need to update Eq. (1) for different forward propagation rules. Specifically, for GCN [12], the main difference from GraphSAGE is that there is not an explicit term $\boldsymbol{X}^{(\ell-1)} \cdot \boldsymbol{W}_{\circ}^{(\ell)}$ to capture the influence of a node to itself. Instead, the self-influence is propagated by adding a self-connection in the graph. Therefore, the adjacency matrix becomes $\boldsymbol{I} + \boldsymbol{A}$ and the normalization is performed differently. The forward propagation of each layer is as follows:

$$\boldsymbol{X}^{(\ell)} = \texttt{ReLU}\left(\hat{\boldsymbol{A}} \cdot \boldsymbol{X}^{(\ell-1)} \cdot \boldsymbol{W}^{(\ell)}\right) \tag{4}$$

where $\hat{\boldsymbol{A}}$ is a symmetrically normalized adjacency matrix calculated by $\hat{\boldsymbol{A}} = (\boldsymbol{I} + \boldsymbol{D})^{-\frac{1}{2}} \cdot (\boldsymbol{I} + \boldsymbol{A}) \cdot (\boldsymbol{I} + \boldsymbol{D})^{-\frac{1}{2}}$, and $\boldsymbol{I}$ is the identity matrix.

For MixHop [1], each layer is able to propagate influence from nodes up to K-hops away (i.e., u is said to be K-hops away from v if the shortest path from u to v has length K). The forward propagation of each layer is defined as:

$$\boldsymbol{X}^{(\ell)} = \texttt{ReLU}\left(\Big\|_{k=0}^{K} \hat{\boldsymbol{A}}^{k} \cdot \boldsymbol{X}^{(\ell-1)} \cdot \boldsymbol{W}_{k}^{(\ell-1)}\right) \tag{5}$$

where "$\|$" is again the operation for matrix concatenation. $\hat{\boldsymbol{A}}^{k}$ means the symmetrically normalized adjacency matrix raised to the power of k. And "order" K is a hyperparameter of the model.

For GAT [23], instead of aggregating the features from the previous layer (i.e., $\boldsymbol{X}^{(\ell-1)}$) using a fixed adjacency matrix (i.e., $\hat{\boldsymbol{A}}$ in GCN or $\tilde{\boldsymbol{A}}$ in GraphSAGE), each GAT layer learns the weight of the adjacency matrix as the "attention". The forward propagation of a GAT layer is specified as:

$$\boldsymbol{X}^{(\ell)} = \texttt{ReLU}\left(\boldsymbol{A}_{\text{att}}^{(\ell-1)} \cdot \boldsymbol{X}^{(\ell-1)} \cdot \boldsymbol{W}^{(\ell)}\right) \tag{6}$$

where each element in the attention adjacency matrix $\boldsymbol{A}_{\text{att}}^{(\ell-1)}$ is calculated as:

$$\left[A_{\text{att}}^{(\ell-1)}\right]_{u,v} = \texttt{LeakyReLU}\left(\boldsymbol{a}^{\mathsf{T}} \cdot \left(\boldsymbol{W}^{(\ell)} \cdot \boldsymbol{x}_{u}^{(\ell-1)} \Big\| \boldsymbol{W}^{(\ell)} \cdot \boldsymbol{x}_{v}^{(\ell-1)}\right)\right) \tag{7}$$

where $\boldsymbol{a}$ is a learnable vector and $\boldsymbol{x}_u$ means the feature vector of node u (i.e., the u-th row of the feature matrix $\boldsymbol{X}^{(\ell-1)}$). As an extension, Eq. (6) can be modified to support "multi-head" attention. Note that the computation pattern of "multi-head" GAT is the same as that of "single-head" captured by Eq. (6) and our parallelization strategy can be easily extended to support the multi-head version. We therefore restrict to Eq. (6) for the discussion on GAT.

In summary, considering all the four models, the full forward propagation during training takes $\boldsymbol{X}$ as the input and generates $\mathcal{L}$ as the output by traversing the GNN layers, the classifier layers, and the loss layer. After obtaining $\mathcal{L}$, we perform backward propagation from the loss layer all the way to the first GNN layer and update the weights by gradients. The gradients are computed by chain-rule. In Section 5, we analyze the computation in backward propagation and propose parallelization techniques for each of the key operations.

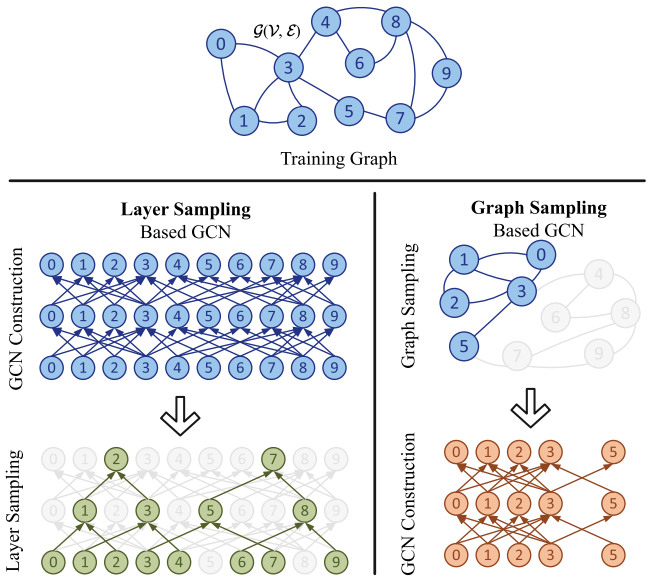

Fig. 1. Illustration on layer sampling and graph sampling based GCN design.

2.2. Minibatch training methods

For large scale graphs, training of the GNN has to proceed in minibatches, so that each iteration of weight update involves only a small number of graph nodes. GraphSAGE [7], FastGCN [4], AS-GCN [9] and VR-GCN [6] incorporate various *layer sampling* techniques to construct minibatches. Upper part of Fig. 1 abstracts the meta-steps of 1. Constructing a full GNN on the training graph $\mathcal{G}$, 2. Sampling nodes from the $|\mathcal{V}|$ nodes of each layer, and 3. Forward and backward propagation among the sampled nodes. For the sampling of step 2, various techniques have been proposed to improve learning quality or training speed. For [6,7,9], they first randomly select a small number of nodes from the outputs of the last GNN layer as the "minibatch" nodes. Then they treat such minibatch nodes as the roots and back-tracks the layer connections to sample connected nodes in the previous layers. When such back-tracking goes from layer L's outputs down to layer 1's inputs, the number of multi-hop neighbors of the roots can be orders of magnitude larger than the number of roots. This is referred to as "neighbor explosion" [4,6,7] (see also analysis in Section 3.2). Note that if u is a k-hop neighbor of v, then u is connected to v via a length-k path in $\mathcal{G}$. Equivalently, node u in layer ℓ of the GNN can influence v in layer $\ell+k$. While [6,9] have proposed techniques to alleviate such "neighbor explosion" of [7], none of them is scalability from the computation complexity perspective. Specifically, the variance reduction based sampler of [6] comes at the cost of much higher memory usage, and the sampler of [9] using an auxiliary neural network incurs significant computation overhead. On the other hand, for [4], the sampling is performed independently at each layer. [4] first computes the sampling probability for each node of $\mathcal{V}$, based on the sparse adjacency matrix $\boldsymbol{A}$. Then it selects a fixed number of nodes from each layer according to such probability. Finally, the sampled GNN to generate the embedding for the minibatch is built by connecting the sampled nodes in adjacent layers. Clearly, [4] avoids "neighbor explosion" since the number of samples in each layer is fixed. Unfortunately, such training can result in significant accuracy degradation. Since the sampling in each layer is independent, significant portion of the node samples in layer i may be disconnected to node samples in layer $i+1$ when $\mathcal{G}$ is large.

In our prior work [31], we proposed a minibatch training method for the GraphSAGE model based on graph sampling, and developed parallelization strategies targeting at shared-memory multi-core processors. We designed a table based data structure to support parallel graph sampling, and a data partitioning scheme supporting parallel feature propagation within subgraphs. In this work, we improve the parallel graph sampling algorithm by a more compact design of the data structure. Thus, we significantly reduce the computation cost and storage overhead of graph sampling. We also propose a scheduling algorithm for the overall training. The scheduler intelligently re-orders the operations in GNN layer propagation to reduce computation complexity, and updates the sampled subgraphs to improve the cache performance. Lastly, we show that our parallelization and scheduling strategies are general, and can be extended to various GNN models including but not limited to GraphSAGE.

Our other work, GraphSAINT [30], extends the idea of training GNNs with graph sampling. GraphSAINT focuses on further improving training accuracy by bias elimination and variance reduction techniques, while this work mostly focuses on the parallelization strategies to achieve superior scalability on multi-core platforms. Note that the training algorithm enhancements proposed by GraphSAINT can be easily incorporated into our parallel execution framework without losing any efficiency or scalability.

3. Graph sampling-based minibatch training

We present a novel graph sampling-based GNN training method. Our parallel minibatch training simultaneously outperforms the state-of-the-art in accuracy, efficiency and scalability. We present the design of the training (Section 3.1), and analyze the advantages in efficiency (Section 3.2) and accuracy (Section 3.3). We then present optimizations to scale training on parallel machines (Sections 4 and 5).

3.1. Design of the minibatch training algorithm

As shown in the lower part of Fig. 1, the graph sampling-based approach does not construct a GNN directly on the original input graph $\mathcal{G}$. Instead, for each iteration of weight update during training, we first sample a small induced subgraph $\mathcal{G}_s(\mathcal{V}_s, \mathcal{E}_s)$ from $\mathcal{G}(\mathcal{V}, \mathcal{E})$. We then construct a *complete*[2] GNN on $\mathcal{G}_s$. The forward and backward propagation are both on this small GNN. Algorithm 1 describes our approach. The key distinction from traditional training methods is that the computations (lines 5–13) are performed on nodes of the sampled graph instead of the sampled layer nodes, thus requiring much less computation in training due to reduced redundancy (Section 3.2). In addition, since the GNN on the subgraph $\mathcal{G}_s$ is complete, the forward propagation rule is almost the same as that of the GNN on the full graph. We can directly use Eqs. (1), (4), (5), (6), (2) and (3) by just replacing the full feature matrix $\boldsymbol{X}^{(\ell)}$ and the full adjacency matrix $\boldsymbol{A}$ with the ones for the subgraph, $\boldsymbol{X}_s^{(\ell)}$ and $\boldsymbol{A}_s$. In Section 3.3, we discuss the requirements for the SAMPLE function (line 3), and present three representative graph samplers that leads to high accuracy of training.

Note that for all the methods discussed in this paper (both the layer sampling based and our proposed graph sampling based), a "*minibatch*" is always defined as node samples in the output GNN layer. For example, consider a GNN with one hidden layer. If a particular method selects 1000, 100 and 10 nodes in the input, hidden and output layers respectively, then we say the *minibatch size* is 10, the *1-hop neighborhood size* is 100 and the *2-hop neighborhood size* is 1000. In this case, the GNN only generates label predictions for the 10 minibatch nodes. The number of hops is with respect to minibatch nodes.

Algorithm 1 Graph sampling based minibatch training algorithm

Input: Training graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{X})$; Ground-truth labels $\overline{\boldsymbol{Y}}$; L-layer GNN model
Output: GNN with trained weights

1: ▷ Iterate over minibatches
2: **while** not converged **do**
3: $\quad\mathcal{G}_s(\mathcal{V}_s, \mathcal{E}_s) \leftarrow \texttt{SAMPLE}(\mathcal{G}(\mathcal{V}, \mathcal{E}))$
4: $\quad\tilde{\boldsymbol{A}}_s \leftarrow$ adjacency matrix of $\mathcal{G}_s$
5: $\quad\boldsymbol{X}_s \leftarrow$ feature matrix by looking up $\boldsymbol{X}$ with $\mathcal{V}_s$
6: $\quad\overline{\boldsymbol{Y}}_s \leftarrow$ ground-truth labels by looking up $\overline{\boldsymbol{Y}}$ with $\mathcal{V}_s$
7: $\quad$Construct complete GNN on $\mathcal{G}_s$
8: $\quad$▷ Forward propagation (e.g. GraphSAGE model)
9: $\quad$**for** $\ell = 1$ to L **do**
10: $\quad\quad\boldsymbol{X}_s^{(\ell)} \leftarrow \texttt{ReLU}\left(\tilde{\boldsymbol{A}} \cdot \boldsymbol{X}_s^{(\ell-1)} \cdot \boldsymbol{W}_\star^{(\ell)} \,\middle\|\, \boldsymbol{X}_s^{(\ell-1)} \cdot \boldsymbol{W}_\circ^{(\ell)}\right)$
11: $\quad$**end for**
12: $\quad\boldsymbol{Y}_s \leftarrow \sigma\left(\texttt{ReLU}\left(\boldsymbol{X}_s^{(L)} \cdot \boldsymbol{W}_{\text{MLP}}\right)\right)$
13: $\quad\mathcal{L}_s \leftarrow \texttt{CE}\left(\boldsymbol{Y}_s, \overline{\boldsymbol{Y}}_s\right)$
14: $\quad$▷ Backward propagation
15: $\quad$Update $\boldsymbol{W}_{\text{MLP}}$, $\boldsymbol{W}_\circ^{(\ell)}$, $\boldsymbol{W}_\star^{(\ell)}$ by gradients w.r.t. $\mathcal{L}_s$
16: **end while**
17: **return** Trained GNN model

3.2. Complexity of graph sampling-based minibatch training

We analyze the computation complexity of our graph-sampling based training and show that it significantly reduces redundancy in computation. In the following analysis, we do not consider the sampling overhead, and we only focus on the forward propagation, since backward propagation has identical computation characteristics as forward propagation. Later, we also experimentally demonstrate that our technique is significantly faster even with the sampling step included (see Section 7).

Using the GraphSAGE design as a representative GNN model Eq. (1), the main operations to propagate forward by one GNN layer include:

- *Feature aggregation*: Each node feature vector from layer-ℓ propagates via layer connections. The aggregation requires $\mathcal{O}\left(|\mathcal{E}_s| \cdot f^{(\ell)}\right)$ operations.
- *Weight transformation*: Each node multiplies its feature with the weight, leading to the overall complexity of $\mathcal{O}\left(|\mathcal{V}_s| \cdot f^{(\ell-1)} \cdot f^{(\ell)}\right)$.

For simplicity, assume $f^{(\ell)} = f$. Further let d_s be the average degree of the subgraph $\mathcal{G}_s$. Complexity of L-layer forward propagation in one minibatch is:

$$\mathcal{O}\left(L \cdot |\mathcal{V}_s| \cdot f \cdot (f + d_s)\right) \tag{8}$$

By convention, one epoch of training is defined as one time traversal of all the training data points by predicting their labels. Thus, by the definition of "minibatch" in Section 3.1, we define an epoch in our training as $|\mathcal{V}| / |\mathcal{V}_s|$ number of minibatches (i.e., subgraphs). Clearly, the computation complexity of an epoch is $\mathcal{O}\left(L \cdot |\mathcal{V}| \cdot f \cdot (f + d_s)\right)$.

[2] Not to be confused with "complete graph". Here a GNN being complete means that the bi-adjacency matrix defining the GNN inter-layer connection has the same non-zeros as the adjacency matrix of the graph $\mathcal{G}_s$. i.e., we do not perform any sampling on the nodes in each GNN layer or the edges connecting consecutive layers.

Comparison against other GNN training methods. As discussed in Section 2.2, for [6,7], each sampled node in layer ℓ further selects d' number of neighbors in layer $\ell - 1$. For [7], d' ranges from 10 to 50, and for [6], $d' = 2$. So depending on the minibatch size (see Section 3.1), the complexity of one epoch falls between:

Case 1 [Small minibatch size]: $\mathcal{O}\left(\left(d'\right)^L \cdot |\mathcal{V}| \cdot f \cdot (f + d')\right)$.

Case 2 [Large minibatch size] $\mathcal{O}\left(L \cdot |\mathcal{V}| \cdot f \cdot (f + d')\right)$.

We observe that when the minibatch size is much smaller than the training graph size, the layer sampling techniques result in high training complexity (computation load grows exponentially with GNN depth). Essentially, due to "neighbor explosion", when the layer-L nodes are traversed only once, the nodes in the previous layer ℓ are sampled and evaluated $\left(d'\right)^{L-\ell}$ times on average. The repeated evaluation of the layer nodes across different minibatches makes training inefficient due to computation redundancy. On the other hand, when the minibatch size of [6,7] becomes comparable to the training graph size, the training complexity grows linearly with the GNN depth and training graph size. However, the resolution of "neighbor explosion" comes at the cost of slow convergence and low accuracy [11], since overly large minibatch size hurts generalization of neural networks. So such training configuration of Case 2 does not scale to large graphs.

If we ignore the convergence rate dependent on the input graph, our graph-sampling based training leads to a parallel algorithm whose complexity is linear in GNN depth and training graph size. The work-efficiency of our training is guaranteed by design: throughout the entire training, for each node v, the number of times its label is predicted in the output layer is equal to the number of times its feature is computed in any hidden layer. In this sense, there is no redundant computation arising from repeated evaluation of hidden layer nodes as discussed above. In addition, by choosing proper graph sampling algorithms, we can construct small representative subgraphs whose sizes do not grow proportionally with the training graph size (as shown in Section 7).

3.3. Accuracy of graph sampling-based training

Layer-based sampling methods assume that a subset of neighbors of a given node is sufficient to learn its representation. We achieve the same goal by sampling the graph itself. If the sampling algorithm constructs enough number of representative subgraphs $\mathcal{G}_s$, our training process should absorb all the information in $\mathcal{G}$, and generate accurate embeddings. More specifically, as discussed in Section 2, the output vectors "embed" the input graph topology as well as the initial node attributes. A good graph sampler, thus, should guarantee:

1. Sampled subgraphs preserve the connectivity characteristics of the training graph.
2. Each training graph node has non-negligible probability to be sampled.

It has been widely studied [8] that various random walk based graph sampling algorithms (including unbiased random walk [30], forest fire [15,16], multiple random walk and frontier sampling [19]) can preserve the various input graph characteristics well. In addition, all these sampling algorithms are able to explore the full set of nodes and edges in the original graph due to the stochasticity in sampling. Thus, such algorithms are all valid candidates for our subgraph sampling based training. From the perspective of computation, unbiased random walk, forest fire and multiple random walk algorithms fall within the "*static*" category of the random walk family according to [25]. In other words, throughout the sampling process, these three sampling algorithms follow a fixed probability distribution on node or edges, regardless of the historically traversed subgraph structure. However, the frontier sampling algorithm maintains a *dynamic* probability distribution updated by the "frontier nodes" at the current timestamp. Therefore, for frontier sampling, computation complexity as well as difficulty in parallelization are both higher compared with the other three static algorithms. In the following, we use frontier sampling as a representative and analyze in detail its performance in terms of accuracy and parallel execution. We then discuss how the proposed techniques can be extended to the other three samplers in Section 4.4.

Before going into the specific steps in sampling, we first give some intuition on why training with frontier sampling may lead to high accuracy. Recall the two requirements above characterizing a good sampler. For requirement 1, while "connectivity" may have several definitions, subgraphs output by [19] approximate the original graph with respect to multiple connectivity measures, including degree distribution, assortative mixing coefficient and clustering coefficients. These graph measures critically define how signals on the graph nodes would propagate and mix via GNN layers, and thus should be carefully maintained by the subgraph samples. For requirement 2, during initialization, the frontier sampler picks some root nodes uniformly at random from the original graph (see Section 4.1). These roots constitute a significant portion of the subgraph nodes. Thus, over large enough number of sampling iterations, all input attributes of the training graph will be covered by the frontier sampler. For readers interested in theoretical justification on the choice of those sampling algorithms, please check the analysis in [30].

4. Parallel graph sampling algorithm

In this section , we first describe our parallelization strategies for the frontier sampling algorithm [19]. Then in Section 4.4, we show how to extend our strategies to other graph samplers.

4.1. Graph sampling algorithm

The frontier sampling algorithm proceeds as follows. Throughout the sampling process, the sampler maintains a constant-size frontier set FS consisting of m vertices in $\mathcal{G}$. In each iteration, the sampler randomly pops out a node v in FS according to a degree based probability distribution, and replaces v in FS with a randomly selected neighbor of v. The popped out v is added to the node set $\mathcal{V}_s$ of $\mathcal{G}_s$. The sampler repeats the above update process on the frontier set FS, until the size of $\mathcal{V}_s$ reaches the desired budget n. Algorithm 2 shows the details. According to [19], a good empirical value of m is around 1000.

Algorithm 2 Frontier sampling algorithm

Input: Training graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; Frontier size m; Node budget n
Output: Induced subgraph $\mathcal{G}_s(\mathcal{V}_s, \mathcal{E}_s)$

```
FS ← Set of m nodes selected uniformly at random from V
V_s ← FS
for i = 0 to n − m − 1 do
    Select u ∈ FS with probability deg(u) / Σ_{v∈FS} deg(v)
    Select u' from { w | (u, w) ∈ E } uniformly at random
    FS ← (FS \ { u }) ∪ { u' }
    V_s ← V_s ∪ { u }
end for
G_s ← Subgraph of G induced by V_s
return G_s (V_s, E_s)
```

In our sequential implementation of training, we notice that about half of the time is spent in the sampling phase. This motivates us to parallelize the graph sampler. The challenges

are: 1. While sampling from a discrete distribution is a well-researched problem, we focus on fast parallel sampling from a *dynamic* probability distribution. Such dynamism is due to the addition/deletion of new nodes in the frontier. Existing methods for fast sampling such as aliasing [24] (which can output a sample in $\mathcal{O}(1)$ time with linear processing) cannot be modified easily for our problem. It is non-trivial to select a node from the evolving FS with low complexity. A straightforward implementation by partitioning the total probability of 1 into m intervals would require $\mathcal{O}(m)$ work to update the intervals for each replacement in FS. Given $m = 1000$ as recommended by the authors in the original paper [19], the $\mathcal{O}(m \cdot n)$ complexity to sample a single $\mathcal{G}_s$ is too expensive. 2. The sampling is inherently sequential as the nodes in the frontier set should be popped out one at a time. Otherwise, $\mathcal{G}_s$ may not preserve the characteristics of the original graph well enough.

To address the above challenges, we first propose a novel data structure that lowers the complexity of frontier sampler and allows thread-safe parallelization (Section 4.2). We then propose a training scheduler that exploits parallelization within and across sampler instances (Sections 4.3 and 6).

4.2. Dashboard based implementation

Since nodes in the frontier set are replaced only one at a time, an efficient implementation should allow incremental update of the probability distribution over the m nodes. To achieve such goal, we propose a "Dashboard" table to store the status of current and historical frontier nodes (a node becomes historical after it gets popped out of the frontier set). The next node to pop out is selected by probing the Dashboard using randomly generated indices. In the following, we formally describe the data structure and operations in the Dashboard-based sampler. The implementation involves two arrays:

- **Dashboard** DB $\in \mathbb{R}^{\eta \cdot m \cdot d}$: A vector maintaining the status and sampling probabilities of the current and historical frontier nodes. If a node v is in the frontier, we "pin" a "tile" of v to the "dashboard". Here a tile is a small data structure storing the meta-data of v, and a pin is an address pointer to the tile. One entry of DB corresponds to one pin. A node v will have $\texttt{deg}(v)$ pins allocated continuously in DB, each pointing to the same tile belonging to v. If v is popped out of the frontier, we invalidate all its pins to `NULL`. The optimal value of the parameter η is explained later.
- **Index array** IA $\in \mathbb{R}^{2\times(\eta \cdot m \cdot d+1)}$: An auxiliary array to help cleanup DB upon table overflow. The j-th column in IA has 2 slots, the first slot records the starting index of the DB pins corresponding to v, where v is the jth node added into DB. The second slot is a flag, which is `True` when v is a current frontier node, and `False` when v is a historical one.

The symbols related to the design and analysis of the Dashboard data structure are summarized in Table 1.

Since the probability of popping out a node in frontier is proportional to its degree, we allocate $\texttt{deg}(v_i)$ continuous entries in DB, for each v_i currently in the frontier set. This way, the sampler only needs to probe DB uniformly at random to achieve line 4 of Algorithm 2. Clearly, DB should contain at least $m{\cdot}d$ entries, where d is the average degree of the frontier nodes. For the sake of incremental updates, we append the entries for the new node and invalidate the entries of the popped out node, instead of changing the values in-place and shifting the tailing entries. The invalidated entries become historical. To accommodate the append operation, we introduce an enlargement factor η (where $\eta > 1$), and set the length of DB to be $\eta \cdot m \cdot d$. As an approximation, we set d as the average degree of the training graph $\mathcal{G}$. As the sampling proceeds, eventually, all of the $\eta \cdot m \cdot d$ entries in DB may be filled up by the information of current and historical frontier nodes. In this case, we free up the space occupied by historical nodes before resuming the sampler. Although cleanup of the Dashboard is expensive, due to the factor η, such scenario does not happen frequently (see complexity analysis in Section 4.3). Using the information in IA, the cleanup phase does not need to traverse all of the $\eta \cdot m \cdot d$ entries in DB to locate the space to be freed. When DB is full, the entries in DB can correspond to at most $\eta \cdot m \cdot d$ vertices. Thus, we safely set the capacity of IA to be $\eta \cdot m \cdot d + 1$. Slot 1 of the last entry of IA contains the current number of used DB entries.

Table 1
Summary of symbols related to the Dashboard based frontier sampling.

Name	Meaning
Dashboard (DB)	Data structure consisting of "pins" and "tiles" to support fast dynamic update of probability distribution
tile	Data structure storing meta-information of frontier nodes
pin	Pointer pointing to the tiles. All pins belonging to the same node will point to a shared tile
Index array (IA)	Data structure helping the cleanup of DB when it is full
m	Number of nodes in the frontier set
n	Total number of nodes to be sampled in the subgraph
d	Average degree of frontier nodes
η	Enlargement factor controlling the computation-storage tradeoff. Larger η: larger DB and less frequent cleanup

4.3. Intra- and inter-subgraph parallelization

Since our subgraph-based GNN training requires independently sampling multiple subgraphs, we can sample different subgraphs on different processors in parallel. Also, we can further parallelize within each sampling instance by exploiting the parallelism in probing, book-keeping and cleanup of DB.

Algorithm 3 shows the details of Dashboard-based parallel frontier sampling, where all arrays are zero-based. Considering the main loop (lines 20 to 30), we analyze the complexity of the three functions in Algorithm 4. Denote $\text{COST}_{\text{rand}}$ and COST_{mem} as the cost to generate one random number and to perform one memory access, respectively.

pardo_POP_FRONTIER. Anytime during sampling, on average, the ratio of valid DB entries (those occupied by current frontier vertices) over total number of DB entries is $1/\eta$. Probability of one probing falling on a valid entry equals $1/\eta$. Expected number of rounds for p processors to generate at least 1 valid probing can be shown to be $1/\left(1-\left(1-\frac{1}{\eta}\right)^p\right)$, where one round refers to one repetition of lines 5 to 7 of Algorithm 4. After selection of v_{pop}, $\texttt{deg}\left(v_{\text{pop}}\right)$ number of slots needs to be updated to invalid values `INV`. Since this operation occurs $(n-m)$ times, the `para_POP_FRONTIER` function incurs $(n-m)\left(\frac{1}{1-(1-1/\eta)^p} \cdot \text{COST}_{\text{rand}} + \frac{d}{p} \cdot \text{COST}_{\text{mem}}\right)$ cost.

pardo_CLEANUP. Each time cleanup of DB happens, we need one traversal of IA to calculate the cumulative sum of indices (slot 1) masked by the status (slot 2), to obtain the new location for each valid entries in DB. On expectation, only $\eta{\cdot}m$ entries of IA is filled, so this step costs $\eta{\cdot}m$. Afterwards, only the valid entries in DB will be moved to the new, empty DB based on the accumulated shift amount. This translates to $m \cdot d$ number of memory operations. The `para_CLEANUP` function is fully parallelized. The cleanup happens only when DB is full, i.e., $\frac{n-m}{(\eta-1)m}$ times throughout sampling. Thus, the cost is $\frac{n-m}{(\eta-1)\cdot m} \cdot \frac{m\cdot d}{p} \cdot \text{COST}_{\text{mem}}$. We ignore the cost of computing the cumulative sum as $\eta m \ll md$.

Algorithm 3 Parallel Dashboard based frontier sampling

Input: Original graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; Frontier size m; Budget n; Enlargement factor η; Number of processors p
Output: Induced subgraph $\mathcal{G}_s(\mathcal{V}_s, \mathcal{E}_s)$

$d \leftarrow |\mathcal{E}|/|\mathcal{V}|$
DB $\leftarrow$ Array of $\mathbb{R}^{1\times(\eta\cdot m\cdot d)}$ with value `NULL`
IA $\leftarrow$ Array of $\mathbb{R}^{2\times(\eta\cdot m\cdot d+1)}$ with value `INV` ▷ INValid
FS $\leftarrow$ Set of m nodes selected uniformly at random from $\mathcal{V}$
$\mathcal{V}_s \leftarrow$ FS
Convert the set FS to an indexable list of nodes
IA$[0, 0] \leftarrow 0$; IA$[1, 0] \leftarrow$`True`;
for $i = 1$ to m **do** ▷ Initialize IA from FS
  IA$[0, i] \leftarrow$ IA$[0, i-1] +$ `deg` (FS$[i-1]$)
  IA$[1, i] \leftarrow$ `True`
end for
IA$[1, m] \leftarrow$ `False`
for $i = 0$ to $m-1$ **pardo** ▷ Initialize DB from FS
  pin $\leftarrow$ Address of 4-tuple tile (FS[i], IA[0, i], IA[0, $i+1$], i)
  for $k =$ IA$[0, i]$ to IA$[0, i+1]-1$ **do**
    DB[k] $\leftarrow$ pin
  end for
end for
cnt $\leftarrow m$; $\mathcal{V}_s \leftarrow \emptyset$;
for $i = m$ to $n-1$ **do** ▷ Main loop of sampling
  v_{pop}, pin $\leftarrow$ `pardo_POP_FRONTIER` (DB, p)
  $v_{\text{new}} \leftarrow$ Node randomly sampled from v_{pop}'s neighbors
  if `deg` $(v_{\text{new}}) > \eta \cdot m \cdot d -$ IA$[0, s] + 1$ **then**
    DB $\leftarrow$ `pardo_CLEANUP` (DB, IA, p)
    cnt $\leftarrow m-1$
  end if
  `pardo_ADD_TO_FRONTIER` (v_{new}, pin, cnt, DB, IA, p)
  $\mathcal{V}_s \leftarrow \mathcal{V}_s \cup \{ v_{\text{new}} \}$
  cnt $\leftarrow$ cnt $+ 1$
end for
$\mathcal{G}_s \leftarrow$ Subgraph of $\mathcal{G}$ induced by $\mathcal{V}_s$
return $\mathcal{G}_s(\mathcal{V}_s, \mathcal{E}_s)$

Algorithm 4 Functions in Dashboard Based Sampler

function PARDO_POP_FRONTIER(DB, p)
  $\text{idx}_{\text{pop}} \leftarrow$`INV` ▷ Shared variable
  for $j = 0$ to $p-1$ **pardo**
    while $\text{idx}_{\text{pop}} ==$ `INV` **do** ▷ Probing DB
      $\text{idx}_p \leftarrow$ Index generated uniformly at random
      if DB$\left[\text{idx}_p\right] \neq$ `NULL` **then**
        $\text{idx}_{\text{pop}} \leftarrow \text{idx}_p$
      end if
    end while
  end for
  $\text{pin}_{\text{pop}} \leftarrow$ DB$\left[\text{idx}_{\text{pop}}\right]$
  $v_{\text{pop}}, i_{\text{pinStart}}, i_{\text{pinEnd}}, i_{\text{IA}} \leftarrow$ tile data pointed to by pin_{pop}
  for $j = 0$ to $p-1$ **pardo**
    Set DB entries to `NULL` from index i_{pinStart} to i_{pinEnd}
  end for
  IA$[1, i_{\text{IA}}] \leftarrow$ `False` ▷ Update IA
  return v_{pop}, pin_{pop}
end function
function PARDO_CLEANUP(DB, IA,p)
  $\text{DB}_{\text{new}} \leftarrow$ New, empty dashboard
  $k \leftarrow$ Cumulative sum of IA$[0, :]$ masked by IA$[1, :]$
  for $i = 0$ to $p-1$ **pardo**
    Move entries from DB to DB_{new} by offsets in k
  end for
  for $i = 0$ to $p-1$ **pardo**
    Re-index IA based on DB_{new}
  end for
  return DB_{new}
end function
function PARDO_ADD_TO_FRONTIER(v_{new}, pin, i, DB, IA, p)
  IA$[0, i+1] \leftarrow$ IA$[0, i] +$ `deg` (v_{new}); IA$[1, i] \leftarrow$ `True`;
  Assign values $(v_{\text{new}}, \text{IA}[0, i], \text{IA}[0, i+1], i)$ to the tuple pointed to by pin
  for $j = 0$ to $p-1$ **pardo**
    Set DB entries to pin from index IA$[0, i]$ to IA$[0, i+1]$
  end for
end function

pardo_ADD_TO_FRONTIER. Adding a new frontier v_{new} to DB requires appending `deg` (v_{new}) new entries to DB. This costs $(n-m)\cdot \frac{d}{p} \cdot \text{COST}_{\text{mem}}$.

Considering all operations in pardo_POP_FRONTIER, pardo_CLEANUP and pardo_ADD_TO_FRONTIER, the overall cost to sample one subgraph on p processors equals:

$$\left(\frac{1}{1-(1-1/\eta)^p} \cdot \text{COST}_{\text{rand}} + \left(2 + \frac{1}{\eta-1}\right)\frac{d}{p} \cdot \text{COST}_{\text{mem}}\right) \cdot (n-m) \tag{9}$$

Assuming $\text{COST}_{\text{mem}} = \text{COST}_{\text{rand}}$, we have the following scalability bound:

Theorem 1. *For any given $\epsilon > 0$, Algorithm 2 guarantees a speedup of at least $\frac{p}{1+\epsilon}$, $\forall p \leq \epsilon d\left(2 + \frac{1}{\eta-1}\right) - \eta$.*

Proof. Note that $\frac{1}{1-(1-1/\eta)^p} \leq \frac{1}{1-\exp(-p/\eta)} \leq \frac{\eta+p}{p}$. This follows from $\frac{1}{1-e^{-x}} = \frac{1}{1-\frac{1}{e^x}} \leq \frac{1}{1-\frac{1}{1+x}} \leq \frac{x+1}{x}$. Further, since $p \leq \epsilon d \cdot (2 + 1/(\eta-1)) - \eta$, we have $\frac{\eta+p}{p} \leq \frac{\epsilon d\cdot(2+1/(\eta-1))}{p}$. Now, speedup obtained by Algorithm 2 compared to a serial implementation ($p = 1$) is

$$\frac{(\eta + d(1/(\eta-1)+2))(n-m)}{\left(\frac{1}{1-(1-1/\eta)^p} + \frac{d}{p}(1/(\eta-1)+2)\right)(n-m)}$$

$$\geq \frac{d(1/(\eta-1)+2)}{\frac{\epsilon d}{p}(1/(\eta-1)+2) + \frac{d}{p}(1/(\eta-1)+2)} \geq \frac{p}{1+\epsilon}. \quad \square$$

Setting $\epsilon = 0.5$, then for any value of η, Theorem 1 guarantees good scalability ($p/1.5$) for at least $p = d - \eta$ processors. As we will see later in this section, we perform the intra-sampler parallelism via AVX instructions. So we do not require p to scale to a large number in practice. Note that the above performance analysis always holds as long as we know the expected node degree in the subgraphs. During the sampling process, when the sampler enters a well connected local region of the original graph, cleanup may happen more frequently since the frontier contains more high degree nodes. However, the sampler would eventually replace those high degree frontier nodes with low degree ones, so that the overall subgraph degree is similar to that of the original graph. Also, note that for graphs with skewed degree distribution, it is possible that the next node to be added into the frontier set has very high degree. Such a node may even require more slots than that is totally available in DB. In this case, we would cleanup DB and allocate all the remaining slots to that node, without further expanding the size of DB. This only slightly alters the sampling distribution since the higher the node degree is, the sooner it would be popped out of the frontier. In the experiments, we also observe that such a corner case does not affect the training accuracy (see Section 7.2).

While the scalability can be high for dense graphs, it is challenging to scale the sampler to massive number of processors on sparse graphs. Feasible parallelism is bound by the graph degree. In summary, the parallel Dashboard based frontier sampling algorithm 1. enables lower serial complexity by incremental update on probability distribution, and 2. scales well up to $p = \mathcal{O}(d)$ number of processors. Compared with our original Dashboard based sampling in [31], the data structure presented in this section is more compact. In the original design, the meta-data of a frontier node v (i.e., the 4-tuple in line 14 of Algorithm 3) is repeatedly stored $\texttt{deg}(v)$ times in DB. In the current design, the meta data is only stored once by introducing the "pin-tile" mechanism. Thus, the DB size is reduced from $4 \cdot \eta \cdot m \cdot d$ to $\eta \cdot m \cdot d$. Such "pin-tile" design significantly reduces both the memory storage and the memory movement cost simultaneously.

To further scale the graph sampling step, we exploit task parallelism across multiple sampler instances. Since the topology of the training graph $\mathcal{G}$ is fixed over the training iterations, sampling and GNN computation can proceed in an interleaved fashion, without any dependency constraints. Detailed scheduling algorithm of the sampling phase and the GNN computation phase is described in Section 6. The general idea is that, during training, we maintain a pool of sampled subgraphs $\{\mathcal{G}_i\}$. When $\{\mathcal{G}_i\}$ is empty, the scheduler launches p_{inter} frontier samplers in parallel, and fill the pool with subgraphs independently sampled from the full graph $\mathcal{G}$. Each of the p_{inter} sampler instances runs on p_{intra} number of processing units. Thus, the scheduler exploits both intra- and inter-subgraph parallelism. In each training iteration, we remove a subgraph $\mathcal{G}_s$ from $\{\mathcal{G}_i\}$, and build a complete GNN upon $\mathcal{G}_s$. Forward and backward propagation stay the same as lines 9 to 15 in Algorithm 1.

When filling the pool of subgraphs, total amount of parallelism $p_{\text{intra}} \cdot p_{\text{inter}}$ is fixed on the target platform. We should choose the value of p_{intra} and p_{inter} carefully chosen based on the trade-off between the two levels of parallelism. Note that the operations on DB mostly involve a chunk of memory with continuous addresses. This indicates that intra-subgraph parallelism can be well exploited at the instruction level using vector instructions (e.g., AVX). In addition, since most of the memory traffic going into DB is in a random manner, it is desirable to have DB stored in cache. As coarse estimation, with $m = 1000$, $\eta = 2$, $d = 25$, the memory consumption by one DB is 400KB.[3] This indicates that DB mostly fits into the private L2 cache (size 256KB) in modern shared memory parallel machines. Therefore, during sampling, we bind one sampler to one processor core, and use AVX instructions to parallelize within a single sampler. For example, on a 40-core machine with AVX2, $p_{\text{intra}} = 8$ and $p_{\text{inter}} = 40$.

Finally, note that the size of DB is determined by the number of frontier nodes, m, rather than the number of subgraph nodes n. While it is true that we may need to increase n when the original training graph $\mathcal{G}$ grows, the size of m would not need to change. The authors of [19] interpret m as the dimensionality of the random walk — frontier sampling on $\mathcal{G}$ is equivalent to a single random walk on $\mathcal{G}$ raised to the mth Cartesian power. With such understanding, the authors of [19] use a fixed number of $m = 1000$ on all experiments in ranging from small graphs to large ones.

4.4. Extension to other graph sampling algorithms

By Section 3.3, it is reasonable to use other graph sampling algorithms to perform minibatch GNN training. Here we evaluate two sampling algorithms: random edge sampling ("Edge") and unbiased random walk sampling ("RW"). The two algorithms are recommended in [30]. The "Edge" sampler assigns the probability of picking an edge (u, v) as $p_{u,v} \propto \frac{1}{\deg(u)} + \frac{1}{\deg(v)}$, and can be understood as a special case of the "RW" algorithm by setting the walk length to be 1. Algorithm 5 specifies the steps of the two algorithms. Under the categorization in Section 3.3, "Edge" and "RW" samplers are static since the sampling probability does not change during the sampling process. Therefore, their computation complexity is much lower than that of frontier sampling. It is easy to show that both have computation complexity of $\mathcal{O}(|\mathcal{V}_s| + |\mathcal{E}_s|)$ (we can use alias method [24] for "Edge" sampling to achieve such complexity).

Algorithm 5 Other graph sampling algorithms ("Edge" and "RW")

Input: Training graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; Sampling parameters: edge budget b; number of roots r; random walk length h
Output: Induced subgraph $\mathcal{G}_s(\mathcal{V}_s, \mathcal{E}_s)$

function EDGE($\mathcal{G}$, m) ▷ Random edge sampler
  $P((u,v)) := \left(\frac{1}{\deg(u)} + \frac{1}{\deg(v)}\right) / \sum_{(u',v')\in\mathcal{E}} \left(\frac{1}{\deg(u')} + \frac{1}{\deg(v')}\right)$
  $\mathcal{E}_s \leftarrow m$ edges randomly sampled from $\mathcal{E}$ according to distribution P
  $\mathcal{V}_s \leftarrow$ Set of nodes that are end-points of edges in $\mathcal{E}_s$
  $\mathcal{G}_s \leftarrow$ Node induced subgraph of $\mathcal{G}$ from $\mathcal{V}_s$
end function
function RW($\mathcal{G}$, r, h) ▷ Unbiased random walk sampler
  $\mathcal{V}_{\text{root}} \leftarrow r$ root nodes sampled uniformly at random from $\mathcal{V}$
  $\mathcal{V}_s \leftarrow \mathcal{V}_{\text{root}}$
  for $v \in \mathcal{V}_{\text{root}}$ **do**
    $u \leftarrow v$
    for $d = 1$ to h **do**
      $u \leftarrow$ Node sampled uniformly at random from u's neighbor
      $\mathcal{V}_s \leftarrow \mathcal{V}_s \cup \{u\}$
    end for
  end for
  $\mathcal{G}_s \leftarrow$ Node induced subgraph of $\mathcal{G}$ from $\mathcal{V}_s$
end function

For the "Edge" and "RW" samplers, we thus only apply inter-sampler parallelism to achieve scalability. We can use exactly the same inter-sampler parallelization strategy discussed above. The only difference is that each subgraph in the pool $\{\mathcal{G}_i\}$ is now obtained by a serial "Edge" or "RW" sampler.

To further improve the training accuracy with "Edge" and "RW" samplers, we further integrate the *aggregator normalization* and *loss normalization* techniques [30] into our implementation. Such normalization requires two minor modifications to our training algorithm:

- Pre-processing: Before training, we would need to independently sample a given number of subgraphs to estimate the probability of each $v \in \mathcal{V}$ and $e \in \mathcal{E}$ being picked by the sampling algorithm. The pre-processing can be parallelized by the strategies discussed above.
- Applying the normalization coefficients: with aggregator normalization, the feature aggregation (i.e., $\tilde{\boldsymbol{A}}_s \cdot \boldsymbol{X}_s$) would be based on a re-normalized adjacency matrix. With loss normalization, the loss $\mathcal{L}_s$ would be computed with *weighted* sum for the minibatch nodes. Therefore, the two normalization steps do not make any change on the computation pattern.

[3] Assume 8-byte address pointing to the tuple of pins. So the size of DB is $2 \cdot 1000 \cdot 25 \cdot 8$ Bytes and the size for the pins is $1000 \cdot 4 \cdot 4$ Bytes.

5. Parallel training algorithm

We next present parallelization techniques for the forward and backward propagation. Specifically, the subgraph based training enables a simple partitioning scheme that ensures near-optimal feature propagation performance.

5.1. Computation kernels in training

After obtaining the subgraphs as minibatches, the GNN computation mainly involves forward and backward propagation along the layers. We first analyze in detail the backward propagation computation for the GraphSAGE model [7]. Then we show that all the four GNN variants presented in Section 2 share the same set of key computation operations. And thus the parallelization strategy can be generally applied to all the models. As for the forward propagation, Eqs. (1)–(3) have already defined all the operations required for the various layers. Next, we derive the equations for calculating gradients.

Starting from the minibatch loss $\mathcal{L}_s$, we first compute the gradient with respect to the classifier output on the subgraph nodes $(\boldsymbol{X}_{\text{MLP}})_s$. Then, using chain-rule, we compute the gradients with respect to the variables of the MLP layer and the graph convolution layers (from layer L back to layer 1).

For the layer with cross-entropy loss, the gradients are computed by:

$$\nabla_{(\boldsymbol{X}_{\text{MLP}})_s}\mathcal{L}_s = \frac{1}{|\mathcal{V}_s|}\cdot\left(\boldsymbol{Y}_s - \overline{\boldsymbol{Y}}_s\right) \tag{10}$$

For the MLP layer, the gradients are computed by:

$$\begin{aligned}\nabla_{\boldsymbol{W}_{\text{MLP}}}\mathcal{L}_s &= \left(\boldsymbol{X}_s^{(L)}\right)^{\mathsf{T}}\cdot \texttt{mask}\left(\nabla_{(\boldsymbol{X}_{\text{MLP}})_s}\mathcal{L}_s\right)\\ \nabla_{\boldsymbol{X}_s^{(L)}}\mathcal{L}_s &= \texttt{mask}\left(\nabla_{(\boldsymbol{X}_{\text{MLP}})_s}\mathcal{L}_s\cdot\left(\boldsymbol{W}_{\text{MLP}}\right)^{\mathsf{T}}\right)\end{aligned} \tag{11}$$

For each graph convolution layer ℓ, the gradients are computed by:

$$\begin{aligned}\nabla_{\boldsymbol{W}_\circ^{(\ell)}}\mathcal{L}_s &= \left(\boldsymbol{X}_s^{(\ell-1)}\right)^{\mathsf{T}}\cdot\texttt{mask}\left(\left[\nabla_{\boldsymbol{X}_s^{(\ell)}}\mathcal{L}_s\right]_{:,0:\frac{1}{2}f^{(\ell)}}\right)\\ \nabla_{\boldsymbol{W}_\star^{(\ell)}}\mathcal{L}_s &= \left(\tilde{\boldsymbol{A}}_s\boldsymbol{X}_s^{(\ell-1)}\right)^{\mathsf{T}}\cdot\texttt{mask}\left(\left[\nabla_{\boldsymbol{X}_s^{(\ell)}}\mathcal{L}_s\right]_{:,\frac{1}{2}f^{(\ell)}:f^{(\ell)}}\right)\\ \nabla_{\boldsymbol{X}_s^{(\ell-1)}}\mathcal{L}_s &= \texttt{mask}\left(\left[\nabla_{\boldsymbol{X}_s^{(\ell)}}\mathcal{L}_s\right]_{:,0:\frac{1}{2}f^{(\ell)}}\right)\cdot\left(\boldsymbol{W}_\circ^{(\ell)}\right)^{\mathsf{T}}\\ &\quad+\left(\tilde{\boldsymbol{A}}_s\right)^{\mathsf{T}}\cdot\texttt{mask}\left(\left[\nabla_{\boldsymbol{X}_s^{(\ell)}}\mathcal{L}_s\right]_{:,\frac{1}{2}f^{(\ell)}:f^{(\ell)}}\right)\cdot\left(\boldsymbol{W}_\star^{(\ell)}\right)^{\mathsf{T}}\end{aligned} \tag{12}$$

From the equations of forward and backward propagation, we observe that the GraphSAGE computation consists of three kernels:

- Feature/gradient propagation in the sparse subgraph – e.g., $\tilde{\boldsymbol{A}}_s\boldsymbol{X}_s^{(\ell)}$;
- Dense weight transformation on the feature/gradient – e.g., $\boldsymbol{X}_s^{(\ell-1)}\boldsymbol{W}_\circ^{(\ell)}$;
- Sparse adjacency matrix transpose – i.e., $\left(\tilde{\boldsymbol{A}}_s\right)^{\mathsf{T}}$.

In fact, the above three are also the key operations for GCN [12], MixHop [1] and GAT [23]. For GCN [12], the forward propagation only contains one pass as compared to the two paths in GraphSAGE (i.e., the two paths being concatenated by the "$\|$" operation). Therefore, in the backward propagation, we replace $\tilde{\boldsymbol{A}}_s$ with $\hat{\boldsymbol{A}}_s$ and only keep the terms containing $\hat{\boldsymbol{A}}_s$ in Eq. (12). For example, we have $\nabla_{\boldsymbol{X}_s^{(\ell-1)}}\mathcal{L}_s = \left(\hat{\boldsymbol{A}}\right)^{\mathsf{T}}\cdot\texttt{mask}\left(\nabla_{\boldsymbol{X}_s^{(\ell)}}\mathcal{L}_s\right)\cdot\left(\boldsymbol{W}^{(\ell)}\right)^{\mathsf{T}}$.

For MixHop [1], each layer in the forward propagation consists of K paths as compared to the two paths in GraphSAGE. Therefore, we need to introduce the $\left(\hat{\boldsymbol{A}}\right)^k$ terms (where $1 \le k \le K$) to Eq. (12) in the backward pass. For example, we need $\left(\hat{\boldsymbol{A}}_s\right)^2\boldsymbol{X}_s^{(\ell-1)}$ to compute $\nabla_{\boldsymbol{W}_2^{(\ell)}}\mathcal{L}_s$. Further note that $\left(\hat{\boldsymbol{A}}_s\right)^2\boldsymbol{X}_s^{(\ell-1)} = \hat{\boldsymbol{A}}_s\cdot\left(\hat{\boldsymbol{A}}_s\boldsymbol{X}_s^{(\ell-1)}\right)$. And even though $\boldsymbol{A}_s$ is sparse, the product $\hat{\boldsymbol{A}}_s\boldsymbol{X}_s^{(\ell-1)}$ is again a dense matrix. So the forward and backward propagation for MixHop does not involve sparse–sparse matrix multiplication and the MixHop computation can still be covered by the three key operations listed above.

For GAT [23], in the forward pass, we need to compute the attention values for each element in the subgraph adjacency matrix. Such computation according to Eq. (7) only involves dense algebra. After obtaining the attention adjacency matrix, the rest of the propagation by Eq. (6) is the same as that of GCN. In the backward pass, according to chain rule, we can still break down the computation steps following the logic in the forward pass. For example, to obtain the gradient with respect to attention parameters $\boldsymbol{a}$, we first obtain the gradients with respect to the attention matrix $\boldsymbol{A}_{\text{att}}^{(\ell-1)}$ by a series of dense matrix operations on $\boldsymbol{X}^{(\ell-1)}$, $\nabla_{\boldsymbol{X}^{(\ell)}}\mathcal{L}_s$ and $\boldsymbol{W}$. Then we obtain the gradient with respect to $\boldsymbol{a}$ based on the gradient with respect to $\boldsymbol{A}_{\text{att}}^{(\ell-1)}$. Even though the mathematical expression for the GAT gradient computation is more complicated, it is easy to see that all the operations involved are again covered by the three key operations listed above.

In summary, if we can efficiently parallelize the three operations listed above, we are automatically able to execute the full forward and backward propagation for the four GNNs. We present our method for transposing the sparse adjacency matrix in Section 5.2 and the techniques for parallel feature propagation in Section 5.3. Now consider the dense matrix multiplication involved in the weight transformation step. Since this operation is a standard BLAS level 2 routine, it can be efficiently parallelized using standard libraries such as Intel® MKL [10].

In the following, we use $\tilde{\boldsymbol{A}}_s$ to represent the subgraph adjacency matrix used in each GNN layer. For different models, the $\tilde{\boldsymbol{A}}_s$ may be replaced by $\hat{\boldsymbol{A}}_s$ or $\boldsymbol{A}_{\text{att}}$.

5.2. Transpose of the sparse adjacency matrix

Since we assume the training graph and the sampled subgraphs are undirected, the transpose of the subgraph adjacency matrix $\left(\tilde{\boldsymbol{A}}_s\right)^{\mathsf{T}}$ can be performed efficiently with low computation and space complexity. We first discuss the serial implementation before moving forward to the parallel version.

Suppose the original adjacency matrix $\tilde{\boldsymbol{A}}$ is represented in the CSR format, consisting of a size-$|\mathcal{V}_s+1|$ index pointer array (INDPTR), a size-$|\mathcal{E}_s|$ indices array (INDICES) and a size-$|\mathcal{E}_s|$ data array (DATA). For an undirected graph, if edge $(u,v)\in\mathcal{E}_s$, then $(v,u)\in\mathcal{E}_s$. Therefore, the index pointer and the indices arrays of $\tilde{\boldsymbol{A}}_s$ are identical as the ones of $\left(\tilde{\boldsymbol{A}}_s\right)^{\mathsf{T}}$. To transpose $\tilde{\boldsymbol{A}}_s$ thus means to generate a new data array by permuting the original DATA of the CSR of $\tilde{\boldsymbol{A}}_s$.

We propose to generate the permuted data array for $\left(\tilde{\boldsymbol{A}}_s\right)^{\mathsf{T}}$ by a single pass of INDPTR and INDICES of $\tilde{\boldsymbol{A}}_s$. Our algorithm relies on a weak assumption on INDICES of $\tilde{\boldsymbol{A}}_s$: for any node v, we assume its neighbor IDs in the indices array, INDICES[INDPTR[v] : INDPTR[$v+1$]], is sorted in ascending order. The transpose operation is shown in Algorithm 6. The correctness of the algorithm can be reasoned as follows. Suppose a column v of the original adjacency matrix has n non-zeros denoted as $\left[\tilde{\boldsymbol{A}}_s\right]_{u_i,v} = a_i$, where $1 \le i \le n$ and the node IDs satisfy $u_i < u_j$ for $i < j$. When we traverse the CSR of $\tilde{\boldsymbol{A}}_s$ (lines 4 to 5), we will read a_i before

Algorithm 6 Transpose of the subgraph adjacency matrix

Input: Original adjacency matrix $\tilde{\boldsymbol{A}}_s$ in the CSR format

Output: Transposed adjacency matrix $\left(\tilde{\boldsymbol{A}}_s\right)^\top$ in the CSR format

```
1: INDPTR, INDICES, DATA ← CSR arrays of Ã_s
2: DATATRANS ← array of size |E_s| initialized to INV
3: PTRDATA ← array of size |V_s| initialized to INDPTR[: |V_s|]
4: for v from 0 to |V_s| − 1 do
5:     for j from INDPTR[v] to INDPTR[v + 1] do
6:         u ← INDICES[j];      a ← DATA[j];
7:         DATATRANS[PTRDATA[u]] ← a
8:         Increment PTRDATA[u] by 1  ▷ Next position to append
9:     end for
10: end for
11: return (Ã_s)^⊤ from INDPTR, INDICES, DATATRANS
```

a_j if the node IDs have $u_i < u_j$. After transpose, the neighbor data – $a_1, \ldots, a_n$ – should be placed in a continuous subarray DATA [INDPTR[v] : INDPTR[$v+1$]] of $\left(\tilde{\boldsymbol{A}}_s\right)^\top$. In addition, a_i should locate to the left of a_j if $u_i < u_j$. Therefore, once reading a_i of the edge (u_i, v) from $\tilde{\boldsymbol{A}}_s$, we can simply append a_i to v's data subarray of the transposed CSR.

The computation and space complexity of Algorithm 6 are $\mathcal{O}(|\mathcal{V}_s| + |\mathcal{E}_s|)$ and $\mathcal{O}(|\mathcal{E}_s|)$ respectively, which are low compared with other operations in training. We parallelize the adjacency matrix transpose at the subgraph level. During sampling, each of the p_{inter} processors sample one subgraph and permute the corresponding DATA array by Algorithm 6. The information of the original and transposed subgraphs are all stored in the pool of $\{\mathcal{G}_i\}$ (Section 4.3), to be later consumed by the GNN layer propagation.

5.3. Parallel feature propagation within subgraph

During training, each node in the graph convolution layer ℓ pulls features from its neighbors, along the layer edges. Essentially, the operation of $\tilde{\boldsymbol{A}}\boldsymbol{X}_s^{(\ell-1)}$ can be viewed as feature propagation within the subgraph $\mathcal{G}_s$.

A similar problem, label propagation within graphs, has been extensively studied in the literature. State-of-the-art methods based on vertex-centric [2], edge-centric [20] and partition-centric [13] paradigms perform node partitioning on graphs so that processors can work independently in parallel. The work in [32] also performs label partitioning along with graph partitioning when the label size is large. In our case, we borrow the above ideas to allow two dimensional partitioning along the graph as well as the feature dimensions. However, we also realize that the above techniques may lead to sub-optimal performance in our GNN based feature propagation, due to two reasons:

- The propagated data from each node is a long feature vector (consisting of hundreds of elements) rather than a small scalar label.
- Our graph sizes are small after graph sampling, so partitioning of the graph may not lead to significant advantage.

In the following, we analyze the computation and communication costs of feature propagation after graph and feature partitioning. We temporarily ignore load-imbalance and partitioning overhead, and address them later on.

Suppose we partition the subgraph into Q_v number of disjoint node partitions $\left\{\mathcal{V}_s^i | 0 \le i \le Q_v - 1\right\}$. Let the set of nodes that send features to $\mathcal{V}_s^i$ be $\mathcal{V}_{\text{src}}^i = \left\{u | (u, v) \in \mathcal{E}_s \wedge v \in \mathcal{V}_s^i\right\}$. Note that $\mathcal{V}_s^i \subseteq \mathcal{V}_{\text{src}}^i$, since we follow the design in [7] to add a self-connection to each node. We further partition the feature vector $\boldsymbol{x}_v \in \mathbb{R}^f$ of each node v into Q_f equal parts $\left\{\boldsymbol{x}_v^i | 0 \le i \le Q_f - 1\right\}$. Each of the processors is responsible for propagation of $\boldsymbol{X}_s^{i,j} = \left\{\boldsymbol{x}_v^j | v \in \mathcal{V}_{\text{src}}^i\right\}$, flowing from $\mathcal{V}_{\text{src}}^i$ into $\mathcal{V}^i$ (where $0 \le i \le Q_v - 1$ and $0 \le j \le Q_f - 1$).

Define $\gamma_v = \frac{\left|\mathcal{V}_{\text{src}}^i\right|}{|\mathcal{V}|}$ as a metric reflecting the graph partitioning quality. While γ_v depends on the partitioning algorithm, it is always bound by $\frac{1}{Q_v} \le \gamma_v \le 1$.

Let $n = |\mathcal{V}_s|$ and $f = |\boldsymbol{x}_v|$. So $\left|\mathcal{V}^i\right| = \frac{n}{Q_v}$ and $\left|\boldsymbol{x}_v^i\right| = \frac{f}{Q_f}$.

In our performance model, we assume p processors operating in parallel. Each processor is associated with a private fast memory (i.e., cache). The p processors share a slow memory (i.e., DRAM). Our objective in partitioning is to minimize the overall processing time in the parallel system. After partitioning, each processor owns $\frac{Q_v \cdot Q_f}{p}$ number of $\boldsymbol{X}_s^{i,j}$, and propagates its $\boldsymbol{X}_s^{i,j}$ into $\mathcal{V}^i$. Due to the irregularity of graph edge connections, accesses into $\boldsymbol{X}_s^{i,j}$ are random. On the other hand, using the CSR format, the neighbor lists of nodes in $\mathcal{V}^i$ can be streamed into the processor, without the need to stay in cache. In summary, an optimal partitioning scheme should:

- Let each $\boldsymbol{X}_s^{i,j}$ fit into the fast memory;
- Utilize all of the available parallelism in the system;
- Minimize the total computation workload;
- Minimize the total slow-to-fast memory traffic;
- Balance the computation and communication load among the processors.

Each round of feature propagation has $\frac{n}{Q_v} \cdot d \cdot \frac{f}{Q_f}$ computation, and $2 \cdot \frac{n}{Q_v} \cdot d + 8 \cdot n \cdot \gamma_v \cdot \frac{f}{Q_f}$ communication (in bytes).[4] Computation and computation over $Q_v \cdot Q_f$ rounds are:

$$
\begin{aligned}
g_{\text{comp}}(Q_v, Q_f) &= n \cdot d \cdot f \\
g_{\text{comm}}(Q_v, Q_f) &= 2 \cdot Q_f \cdot n \cdot d + 8 \cdot Q_v \cdot n \cdot f \cdot \gamma_v
\end{aligned} \tag{13}
$$

Note that $g_{\text{comp}}(Q_v, Q_f)$ is not affected by the partitioning scheme. We thus formulate the following *communication minimization problem*:

$$
\begin{aligned}
\underset{Q_v, Q_f}{\text{minimize}} \quad & g_{\text{comm}}(Q_v, Q_f) = 2Q_f \cdot nd + 8Q_v \cdot nf\gamma_v \\
\text{subject to} \quad & Q_v Q_f \ge p; \quad \frac{8nf\gamma_v}{Q_f} \le S_{\text{cache}}; \quad Q_v, Q_f \in \mathbb{Z}^+;
\end{aligned} \tag{14}
$$

Next, we prove that *without any graph partitioning* we can obtain a 2-approximation for this optimization problem for small subgraphs.

Theorem 2. *$Q_v = 1, Q_f = \max\left\{p, \frac{8nf}{S_{cache}}\right\}$ results in a 2-approximation of the communication minimization problem Eq. (14), for $p \le \frac{4f}{d}$ and $2nd \le S_{cache}$, irrespective of the partitioning algorithm.*

Proof. Note that since $Q_v, Q_f \ge 1$ and $\gamma_v \ge 1/Q_v$, $\forall Q_v, Q_f$:

$$
g_{\text{comm}}(Q_v, Q_f) \ge 2Q_f nd + 8Q_v nf \frac{1}{Q_v} \ge 8nf.
$$

[4] Given that sampled graphs are small, we use `INT16` to represent the node indices. We use `DOUBLE` to represent each feature value.

Set $Q_v = 1$ and $Q_f = \max\left\{p, \frac{8nf}{S_{\text{cache}}}\right\}$. Clearly, $\gamma_v = 1$.

Case 1, $p \geq \frac{8nf}{S_{cache}}$ In this case, $Q_f = p \geq 8nf/S_{\text{cache}}$. Thus both constraints are satisfied. And,

$$g_{\text{comm}}(1, p) = 2ndp + 8nf$$
$$= 8nf\left(\frac{pd}{4f} + 1\right) \leq 8nf \cdot (1 + 1) = 16nf$$

due to $p \leq 4f/d$.

Case 2, $p \leq \frac{8nf}{S_{cache}}$ In this case, $Q_f = 8nf/S_{\text{cache}}$ is a feasible solution. And,

$$g_{\text{comm}}\left(1, \frac{8nf}{S_{\text{cache}}}\right) = 2nd\frac{8nf}{S_{\text{cache}}} + 8nf$$
$$= 8nf\left(\frac{2nd}{S_{\text{cache}}} + 1\right) \leq 8nf \cdot (1 + 1) = 16nf$$

due to $2nd \leq S_{\text{cache}}$.

In both cases, the approximation ratio of our solution is ensured to be:

$$\frac{g_{\text{comm}}\left(1, \max\left\{p, \frac{8nf}{S_{\text{cache}}}\right\}\right)}{\min g_{\text{comm}}(Q_v, Q_f)} \leq \frac{16nf}{8nf} = 2$$

Note that this holds for $S_{\text{cache}} \geq 2nd$. So for a cache size of 256 KB, number of edges in the subgraph (i.e., nd) can be up to 128 K. Such upper bound on $|\mathcal{E}_s|$ can be met by the subgraphs in consideration. Also, since $f \gg d$, the condition $p \leq 4f/d$ holds for most of the shared memory platforms in the market. Note that the above theorem is derived by a simple lower bounding on the ratio γ_v for any (including the optimal) partitioning scheme. However, finding such optimal partitioning is computationally infeasible even on small subgraphs, since there are exponential number of possible partitioning. We thus do not provide experimental evaluation on this theorem. □

Using typical values $n \leq 8000$, $f = 512$, and $d = 15$, then for up to $p \leq \frac{4f}{d} = 136$ cores,[5] the total slow-to-fast memory traffic under feature only partitioning is less than 2 times the optimal. Recall the two properties (see the beginning of this section) that differentiate our case with the traditional label propagation. Because the graph size n is small enough, we can find a feasible $Q_f \in \mathbb{Z}^+$ solution to satisfy the cache constraint $\frac{8nf}{Q_f} \leq S_{\text{cache}}$. Because the value f is large enough, we can find enough number of feature partitions such that $Q_f \geq p$. Algorithm 7 specifies our feature propagation.

Lastly, the feature only partitioning leads to two more important benefits. Since the graph is not partitioned, load-balancing (with respect to both computation and communication) is optimal across processors. Also, our partitioning incurs almost zero pre-processing overhead since we only need to extract continuous columns to form sub-matrices. In summary, the feature propagation in our graph sampling-based training achieves 1. Minimal computation; 2. Optimal load-balancing; 3. Zero pre-processing cost; 4. Low communication volume.

Algorithm 7 Feature propagation within sampled graph

Input: $\mathcal{G}_s(\mathcal{V}_s, \mathcal{E}_s)$ with adjacency matrix $\tilde{\boldsymbol{A}}_s$; Node feature matrix $\boldsymbol{X}_s^{(\ell-1)}$; Cache size S_{cache}; Number of processors p
Output: Feature matrix $\boldsymbol{X}_s^{(\ell)}$

```
n ← |V_s| ;   f ← length of the feature vector of a node;
Q_f ← max{p, 8nf/S_cache} ;   f' ← f/Q_f ;
Column-partition X_s^(ℓ-1) into Q_f equal-size parts [X_s^(ℓ-1)]_{:, i·f':(i+1)·f'}
for r = 0 to Q_f/p − 1 do
    for j = 0 to p − 1 pardo
        i ← r + j · Q_f/p
        [X_s^(ℓ)]_{:, i·f':(i+1)·f'} ← Ã_s · [X_s^(ℓ-1)]_{:, i·f':(i+1)·f'}
    end for
end for
return X_s^(ℓ)
```

6. Runtime scheduling

6.1. Computation order of layer operations

Both the forward and backward propagation of GNN layers (Eq. (4), (1), (5), (6) and (12)) involve multiplying a chain of three matrices. Given a chain of matrix multiplication, it is known that different orders of computing the chain leads to different computation complexity. In general, we can use dynamic programming techniques to obtain the optimal order corresponding to the lowest computation complexity [17]. Specifically, for our training problem, we have a chain of three matrices whose sizes and densities are known once the subgraphs are sampled. Consider a sparse matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ (with density δ), and two dense matrices $\boldsymbol{W}_1 \in \mathbb{R}^{n \times f_1}$ and $\boldsymbol{W}_2 \in \mathbb{R}^{f_1 \times f_2}$. To calculate $\boldsymbol{AW}_1\boldsymbol{W}_2$, there are two possible computation orders. Order 1 of $(\boldsymbol{AW}_1)\boldsymbol{W}_2$ computes the partial result $\boldsymbol{P} = \boldsymbol{AW}_1$ first and then computes $\boldsymbol{PW}_2$. This order of computation requires $\delta \cdot n^2 \cdot f_1 + n \cdot f_1 \cdot f_2$ Multiply-ACcumulate (MAC) operations. Order 2 of $\boldsymbol{A}(\boldsymbol{W}_1\boldsymbol{W}_2)$ computes the partial result $\boldsymbol{P} = \boldsymbol{W}_1\boldsymbol{W}_2$ first and then computes $\boldsymbol{AP}$. This order requires $\delta \cdot n^2 \cdot f_2 + n \cdot f_1 \cdot f_2$ MAC operations. Therefore, if $f_1 < f_2$, order 1 is better. Otherwise, we should use order 2. Similarly, suppose $\boldsymbol{W}_3 \in \mathbb{R}^{n \times f_3}$ and our target is $(\boldsymbol{W}_1)^{\mathsf{T}}\boldsymbol{AW}_3$. Then order 1 of $(\boldsymbol{AW}_1)^{\mathsf{T}}\boldsymbol{W}_3$ is better than order 2 of $(\boldsymbol{W}_1)^{\mathsf{T}}(\boldsymbol{AW}_3)$ if and only if $f_1 < f_3$.

Consider a GraphSAGE layer ℓ. If $f^{(\ell-1)} < f^{(\ell)}$, we should use order 1 to calculate the forward propagation of Eq. (4), order 1 to calculate $\nabla_{\boldsymbol{W}_\circ^{(\ell)}}\mathcal{L}_s$ of Eq. (12) and order 2 to calculate $\nabla_{\boldsymbol{X}_s^{(\ell-1)}}\mathcal{L}_s$ of Eq. (12).

Note that the decision of the scheduler only relies on the dimension of the matrices, and thus can be made during runtime at almost no cost. In addition, the partitioning strategy presented in Section 5.3 does not rely on any specific computation order. In summary, the light-weight scheduling algorithm reduces computation complexity without affecting scalability.

6.2. Scheduling the feature partitions

After partitioning the feature matrix (Section 5.3), the question still remains how to schedule these partitions for further performance optimization. Ideally, since the operations on the partitions are completely independent, any scheduling would lead to identical performance. However, in reality, the partitions may undesirably interact with each other due to "false sharing" of data in private caches. If the size of each feature partition is not divisible by the cacheline size, then in the private cache of the processor owning partition i, there may be one cacheline containing data of both partitions i and $i + 1$, and another cacheline

[5] d here refers to the average degree of the sampled graph rather than the training graph. Thus, d value here is set to be lower than that in Section 4.

containing data of both partitions $i-1$ and i. Therefore, if the partitions $i-1$, i and $i+1$ are computed concurrently, there may be undesirable data eviction to keep the three caches clean. So the scheduler should try not to dispatch adjacent partitions at the same time, and we follow the processing order as specified by lines 5 and 6 of Algorithm 7 to achieve this goal.

When the number of processors is large or the number of feature partitions is small (i.e., line 4 of Algorithm 7 finishes in one iteration), it is inevitable to process adjacent partitions in parallel. On the other hand, note that if the partition size is divisible by the cacheline size, we can avoid "false sharing" regardless of the scheduling. The partition size equals $w\cdot|\mathcal{V}_s|\cdot f/Q_f$, where w specifies the word-length. Suppose the cacheline size is S_{cline}. Then our goal is to make $|\mathcal{V}_s|$ divisible by S_{cline}/w. For example, if we use double-precision floating point numbers in training and the cacheline size is 128 bytes, then we can clip the number of subgraph nodes to be divisible by 16. Considering that $|\mathcal{V}_s|$ is in the order of 10^3, such clipping has negligible effect on the subgraph connectivity and the training accuracy. The node clipping can be performed before the induction step (line 9 of Algorithm 2) by randomly dropping nodes in $\mathcal{V}_s$. Therefore, the clipping step incurs almost zero cost.

6.3. Overall scheduler

Algorithm 8 Runtime scheduling (e.g., Frontier sampling)

Input: Training graph $\mathcal{G}(\mathcal{V},\mathcal{E},\boldsymbol{X})$; Ground truth labels $\overline{\boldsymbol{Y}}$; L-layer GNN model; Sampler parameters m, n, η; Parallelization parameters $p_{\text{inter}}, p_{\text{intra}}$
Output: Trained GNN weights

1: $\{\mathcal{G}_i\} \leftarrow \emptyset$ ▷ Set of unused subgraphs
2: **while** not terminate **do** ▷ Iterate over minibatches
3: **if** $\{\mathcal{G}_i\}$ is empty **then**
4: **for** $p = 0$ to $p_{\text{inter}} - 1$ **pardo**
5: $\mathcal{G}_s \leftarrow$ SAMPLE$(\mathcal{G}(\mathcal{V},\mathcal{E}))$ with p_{intra}; Clip nodes by cacheline size
6: Transpose $\mathcal{G}_s$ by Algorithm 6
7: Add $\mathcal{G}_s$ and its transposed array DATA to $\{\mathcal{G}_i\}$
8: **end for**
9: **end if**
10: $\mathcal{G}_s \leftarrow$ Subgraph popped out from $\{\mathcal{G}_i\}$
11: Construct GNN on $\mathcal{G}_s$
12: Determine the matrix multiplication order by Section 6.1
13: Parallel forward and backward propagation on GNN
14: **end while**
15: **return** Trained GNN weights

Algorithm 8 presents the overall training scheduler. By Section 4.3, multiple samplers can be launched in parallel without any data dependency. This is shown by lines 4 to 8. Note that the clipping follows the objective specified in Section 6.2 and the transpose of $\mathcal{G}_s$ follows Algorithm 6. After the GNN is constructed, the forward and backward propagation operations are parallelized by the techniques presented in Section 5. The scheduler performs two decisions based on the sampled subgraphs. The first decision (during runtime) is to perform node clipping to improve cache performance (Section 6.2). The second decision (statically performed before the actual training) is to determine the order of matrix chain multiplication in both forward and backward propagation to reduce computation complexity (Section 6.1).

Note that our scheduler is a general one, in the sense that the training can replace the frontier sampler with any other graph sampling algorithm in a plug-and-play fashion. The processing by the scheduler has negligible overhead.

Table 2
Dataset statistics.

Dataset	Nodes	Edges	Attributes	Classes	Train/Val/Test
PPI	14,755	225,270	50	121 (M)	0.66/0.12/0.22
Reddit	232,965	11,606,919	602	41 (S)	0.66/0.10/0.24
Yelp	716,847	6,977,410	300	100 (M)	0.75/0.15/0.10
Amazon	1,598,960	132,169,734	200	107 (M)	0.80/0.05/0.15
Synthetic	2^{20}–2^{25}	2^{23}–2^{30}	50	2 (S)	0.50/0.25/0.25

(M): **M**ulti-class classification; (S): **S**ingle-class.

7. Experiments

7.1. Experimental setup

We conduct experiments on 4 large scale real-world graphs as well as on synthetic graphs. Details of the datasets are described as follows:

- PPI [21]: A protein–protein interaction graph. A node represents a protein and edges represent protein interactions.
- Reddit [21]: A post–post graph. A node represents a post. An edge exists between two posts if the same user has commented on both posts.
- Yelp [26,30]: A social network graph. A node is a user. An edge represents friendship. Node attributes are user comments converted from text using Word2Vec [18].
- Amazon [30]: An item–item graph. A node is a product sold by Amazon. An edge is present if two items are bought by the same customer. Node attributes are converted from bag-of-words of text item descriptions using singular value decomposition (SVD).
- Synthetic graphs: Graphs generated by Kronecker generator [14]. We follow [14] and set the initiator matrices to be proportional to the 2 by 2 matrix [[0.9, 0.5], [0.5, 0.1]]. We generate two sets of Kronecker graphs. The first set consists of graphs with fixed average degree of 16 and number of nodes equals 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24} and 2^{25}. The second set consists of graphs with 2^{20} nodes and the average degree equals 8, 16, 32, 64, 128, 256 and 512.

The PPI and Reddit datasets are standard benchmarks used in [4,6,7,9,12]. The larger scale graphs, Yelp and Amazon, are processed and evaluated in [30,31]. We use the set of four real-world graphs for a thorough evaluation on accuracy, efficiency and scalability. Table 2 shows the specification of the graphs. We use "fixed-partition" split, and the "Train/Val/Test" column shows the percentage of nodes in the training, validation and test sets. "Classes" shows the total number of node classes (i.e., number of columns of $\boldsymbol{Y}$ and $\overline{\boldsymbol{Y}}$ in Eq. (3)). For synthetic graphs, we can only generate the graph topology. The node attributes and the class memberships are filled by random numbers.

For our graph sampling based GNN training, we open-source two implementations in `Python` (with Tensorflow) and `C++` (with OpenMP), respectively.[6] We use the `Python` (Tensorflow) version for single threaded accuracy evaluation in Section 7.2, since the baseline implementations are provided in `Python` with Tensorflow. We use the `C++` version to measure scalability of our parallel training in Sections 7.3, 7.4 and 7.7. The `C++` implementation is necessary, since `Python` and Tensorflow are not flexible enough for parallel computing experiments (e.g., AVX and thread binding are not explicit in `Python`). Our `C++` implementation achieves comparable accuracy as the Tensorflow one.

We run experiments on a dual 20-Core Intel® Xeon E5-2698 v4 @2.2 GHz machine with 512GB of DDR4 RAM. For the `Python`

[6] Code available at: https://github.com/GraphSAINT/GraphSAINT.

implementation, we use `Python` 3.6.5 with Tensorflow 1.10.0. For the `C++` implementation, the compilation is via Intel® ICC (`-O3` flag). ICC (version 19.0.5.281), MKL (version 2019 Update 5) and OMP are included in Intel Parallel Studio Xe 2018 update 3.

7.2. Evaluation on accuracy and efficiency

Our graph sampling-based training significantly reduces computation complexity without accuracy loss. To eliminate the impact of different parallelization strategies on training time, here we run our implementation as well as all the baselines using single thread. Fig. 2 plots the relation between accuracy (F1 micro score) and sequential training time. To be consistent with the settings in the original papers of the baselines, all measurements here are based on the GNN models of two GCN/GraphSAGE layers. Accuracy is measured on the validation set at the end of each epoch. Between the two baselines [7,12], GraphSAGE [7] achieves higher accuracy and faster convergence. Compared with [7], our minibatch training achieves higher accuracy on all graphs, showing that our graph sampler can preserve important characteristics from the original training graph. Frontier, random walk and edge sampling algorithms perform similarly on Reddit, Yelp and Amazon. On PPI, random walk and edge sampling algorithms result in lower accuracy than the frontier sampler. This is potentially due to the fact that frontier sampler preserves some graph measures better than simpler samplers such as Edge and RW [19]. Due to the stochasticity in training, we define an accuracy threshold to measure training time speedup. Let a_0 be the highest accuracy achieved by the baselines on a given dataset. We define the accuracy threshold as $a_0 - 0.0025$. Serial training time speedup is calculated as: the time for the best performing baseline to reach the threshold divided by the time for our model to reach the threshold. We achieve serial training time speedup of 1.9×, 7.8×, 4.7× and 2.1× for PPI, Reddit, Yelp and Amazon, respectively. As stated in Section 7.1, in this set of experiments, all the runs are executed under the same Tensorflow framework using single thread. Therefore, the speedup achieved by us is not related to our parallelization strategies and is purely due to our graph sampling based training algorithm. Such significant speedup verifies that our minibatch training improves the computation efficiency by avoiding "neighbor explosion" (see Section 3.2).

7.3. Evaluation on scalability

In the following, we evaluate scalability of the various operations (graph sampling, feature propagation and weight transformation) in training.

7.3.1. Scalability of overall training

For the proposed GNN training, Fig. 3 shows the parallel training speedup relative to sequential execution. The execution time includes every training steps specified by lines 2 to 13 of Algorithm 8 — 1. frontier graph sampling (with AVX enabled) and subgraph transpose, 2. feature aggregation in the forward propagation and its corresponding operation in the backward propagation, 3. weight transformation in the forward propagation and its corresponding operation in the backward propagation, and 4. all the other operations (e.g., `ReLU` activation, sigmoid function, etc.) in the forward and backward propagation. As before, we evaluate scaling on a 2-layer GraphSAGE model, with small and large hidden dimensions, $f^{(1)} = f^{(2)} = 512$ and 1024, respectively. As shown by the plots A and D of Fig. 3, the overall training is highly scalable, consistently achieving around 15× speedup on 40-cores for all datasets. The performance breakdown in plots G and H of Fig. 3 suggests that sampling time corresponds to only a small portion of the total training time. This is due to 1. low serial complexity of our Dashboard based implementation, and 2. highly scalable implementation using intra- and inter-sampler parallelism. In addition, feature aggregation for Amazon corresponds to a significantly higher portion of the total time compared with other datasets. This is due to the higher degree of the subgraphs sampled from Amazon. The main bottleneck in scaling is the weight transformation step performing dense matrix multiplication (see analysis in Section 7.3.4). The overall performance scaling is also data dependent. For denser graphs such as Amazon, the scaling of the feature aggregation step dominates the overall scalability. For the other sparser graphs, the weight transformation step has a higher impact on the training. Lastly, our parallel algorithm can scale well under a wide range of configurations — whether the hidden dimension is small or large; whether the training graph is small or large, sparse or dense.

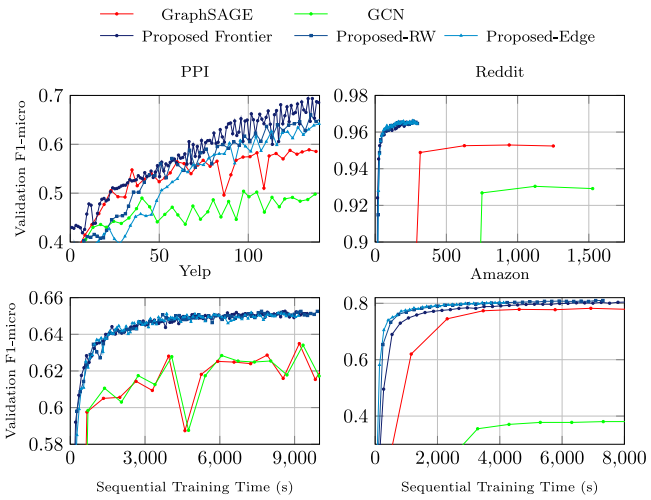

Fig. 2. Time–Accuracy plot for sequential execution..

7.3.2. Scalability of parallel graph sampling

We evaluate the effect of inter-sampler parallelism for the frontier, random walk and edge sampling algorithms, and intra-sampler parallelism for the frontier sampling algorithm.

For the frontier sampling algorithm, the AVX2 instructions supported by our target platform translate to maximum of 8 intra-subgraph parallelism ($p_{\text{intra}} = 8$). The total of 40 Xeon cores makes $1 \leq p_{\text{inter}} \leq 40$. Fig. 4. A shows the effect of p_{inter}, when $p_{\text{intra}} = 8$ (i.e., we launch $1 \leq p_{\text{inter}} \leq 40$ independent samplers, where AVX is enabled within each sampler). Sampling is highly scalable with inter-subgraph parallelism. We observe that scaling performance degrades when going from 20 to 40 cores, due to mixed effect of lower boost frequency and limited memory bandwidth. With all the 20 cores in one chip executing AVX2 instructions, the Xeon CPU can only boost to 2.2 GHz, in contrast with 3.4 GHz for executing AVX instructions only on one core. Fig. 4.B shows the effect of p_{intra} under various p_{inter}. The bars show the speedup of using AVX instructions comparing with otherwise. We achieve around 4× speedup on average. The scaling on p_{intra} is data dependent. Depending on the training graph degree distribution, there may be significant portion of nodes with less than 8 neighbors, resulting in under-utilization of the AVX2 instruction. We can understand such under-utilization of instruction-level parallelism as a result of load-imbalance due to node degree variation. Such load-imbalance explains the discrepancy from the theoretical modeling on the sampling scalability (Theorem 1).

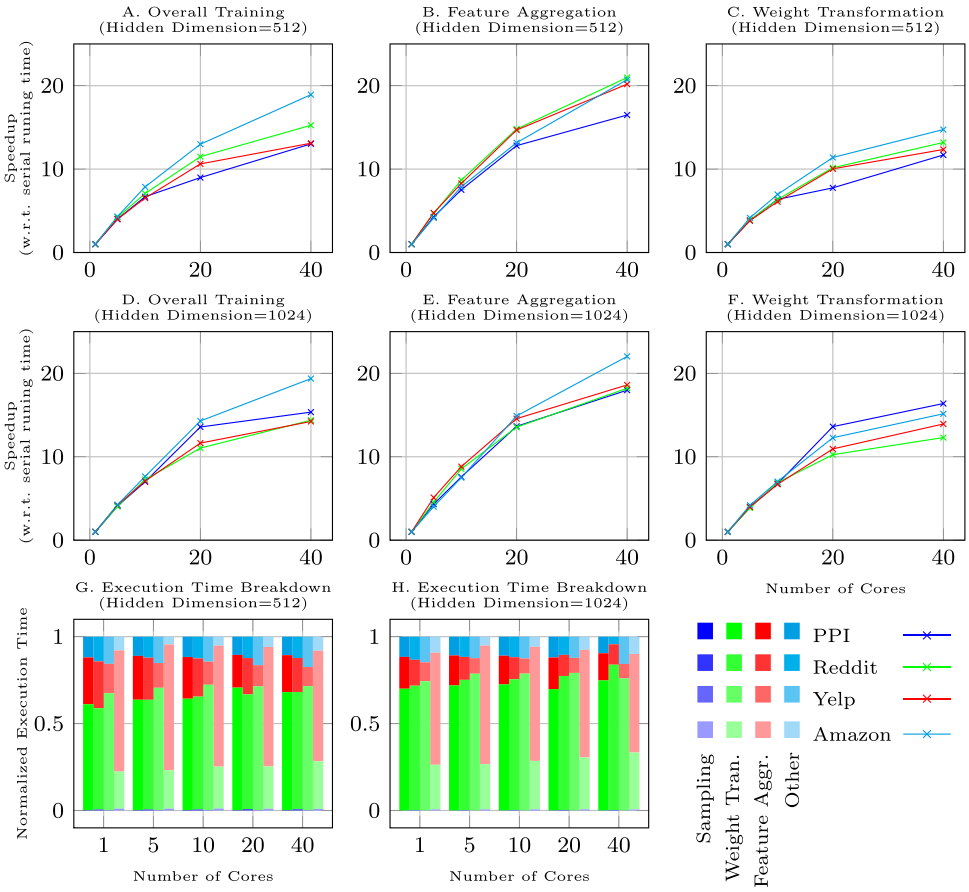

Fig. 3. Scaling evaluation with hidden feature dimensions: 512 and 1024..

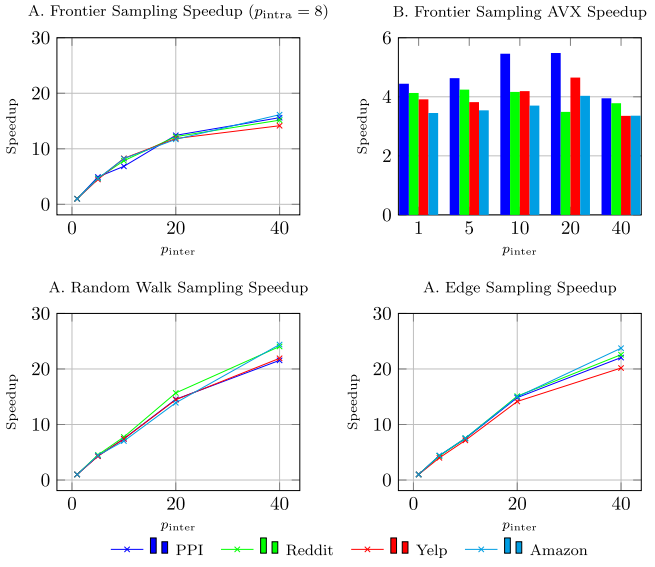

Fig. 4. Sampling speedup (inter- & intra-subgraph parallelism).

Fig. 4.C and 4.D show the effect of p_{inter} for random walk and edge sampling algorithms. Both sampling algorithms scale more than 20× when $p_{inter} = 40$. As we do not use AVX instructions for these two samplers (i.e., $p_{intra} = 1$, and the CPU frequency is unaffected), the scalability from 20 cores to 40 cores is better than that of the frontier sampler.

7.3.3. Scalability of feature aggregation

Fig. 3 shows the scalability of the feature aggregation step using our partitioning strategy. We achieve good scalability (around 20× speedup on 40 cores) for all datasets under various feature sizes, thanks to our caching strategy and the optimal load-balance discussed in Section 5.3. According to the analysis, the scalability of feature aggregation should not be significantly affected by the subgraph topological characteristics. Therefore, we observe from plots B and E of Fig. 3 that, the curves for the four datasets look similar to each other.

7.3.4. Scaling of weight transformation

As discussed in Section 5.1, the weight transformation operation is implemented by `cblas_dgemm` routine of the Intel® MKL [10] library. All optimizations on the dense matrix multiplication are internally implemented in the library. Plots C and F of Fig. 3 show the scalability result. On 40 cores, average of 13× speedup is achieved. We speculate that the overhead of MKL's internal thread and buffer management is the bottleneck on further scaling.

7.4. Effect of cache size

Since our partitioning strategy for feature aggregation (Section 5.3) is based on the L2-cache size of the system, we evaluate the cache miss rate under various cache sizes by simulation. We use CSR format to represent the sparse adjacency matrix of the subgraph and column major layout to represent the dense feature matrix $\boldsymbol{X}_s$. We use the open-source simulator DynamoRIO [3] to simulate our C++ implementation. We configure the system to be 40 cores with two levels of cache, where the first level of

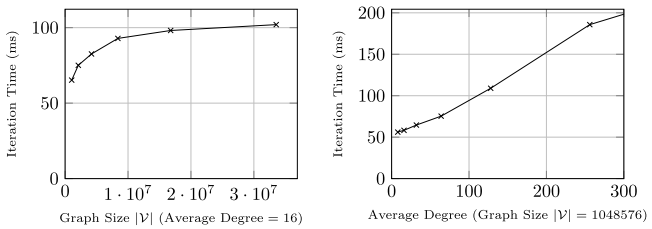

Fig. 5. Training Time in Synthetic Graph..

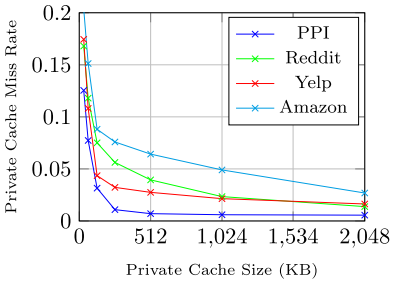

Fig. 6. L2 Cache Miss Rate..

cache corresponds to the L2-cache of the real system. We vary the size of the first level of private cache from 32 KB to 2048 KB. We fix the size of the second level of shared cache to be 50MB. We let the simulator to run one full training iteration and record the cache miss rate for the first level of private cache. Fig. 6 shows the effect of cache size on cache miss rate. When the cache size increases from 32 KB to 512 KB, the cache miss rate quickly drops to below 5%. The parallel execution using our partitioning strategy indeed leads to low cache miss rate. This indicates small amount of slow-to-fast memory data traffic as a benefit of our partitioning strategy.

7.5. Comparison with GPU

We compare the proposed training algorithm with GPU implementation from Tensorflow. We run the GPU program on an Nvidia Tesla P100 GPU with 16 GB of GDDR5 memory, with the same Xeon CPU server as described in Section 7.1. Table 3 shows the performance of the proposed training algorithm on CPU and the Tensorflow implementation on GPU. Both use the same parallel graph sampling algorithm as described in Section 4. For the CPU execution, we use all the available 40 cores. For GPU program, the sampling is done on CPU with 40 cores, while the rest parts are done on GPU. We use the frontier sampling algorithm with node budget $n = 8000$ and $p_{\text{intra}} = 8$. We choose hidden dimension $f = 512$ and record the average execution time per iteration for 100 iterations. The GPU program runs faster than the CPU program by 1.93×, 2.71×, 2.05× and 2.20× on PPI, Reddit, Yelp and Amazon dataset. Note that the peak performance of the CPUs is only 3.5 TFLOPS while the peak performance of the GPU is 10.3 TFLOPS. As stated in Section 5, the proposed parallel training algorithm scales up to 136 cores on CPU. On a 64- or 128-core machine, the proposed algorithm would out-perform GPU based on our modeling (Section 5.3). Importantly, the fast training on GPU also indicates the effectiveness of our graph sampling based minibatch algorithm as well as our parallelization strategy on the frontier sampler.

Table 3
Execution time (s) per iteration (hidden dimension = 512).

Dataset	CPU	GPU
PPI	0.1974	0.1021
Reddit	0.3676	0.1357
Yelp	0.2917	0.1420
Amazon	0.4416	0.2004

7.6. Evaluation on synthetic graphs

Since the largest available real-world dataset for GNN training (i.e., Amazon) contains only about 1.5 million nodes, we generate synthetic graphs of much larger sizes to perform more thorough scalability evaluation. In the left plot of Fig. 5, the sizes of the synthetic graphs grow from 1 million nodes to around 33 million nodes. All synthetic graphs have average degree of 16. We run a 2-layer GNN with hidden dimension of 512 on the subgraphs of the synthetic graphs. The vertical axis denotes the time to compute one iteration (i.e., the time to perform forward and backward propagation on one minibatch subgraph). The subgraphs are all sampled by the frontier sampling algorithm with the same sampling parameters of $n = 8000$ and $m = 1000$. With the increase of the training graph size, the iteration time converges to a constant value of around 100 ms. This indicates that our parallel training is highly scalable with respect to the graph size. When increasing the number of graph nodes, we keep the average degree unchanged. Therefore, the degree of the sampled subgraphs also keeps unchanged (due to the property of frontier sampling). Since we set the node budget n to be fixed, the subgraph size (in terms of number of nodes and edges) in each iteration is approximately independent of the total number of nodes in the training graph. So the cost to perform one step of gradient update does not depend on the training graph size (for a given training graph degree).

In the right plot of Fig. 5, we fix the graph size as $|\mathcal{V}| = 2^{20}$ and increase the average degree. Under the same sampling algorithm, if the original graph becomes denser, the sampled subgraphs are more likely to be denser as well. The computation complexity of feature aggregation is proportional to the subgraph degree. We observe that the iteration time approximately grows linearly with the average degree of the original training graph. This indicates that our parallel training algorithm can handle both sparse and dense graphs very well.

7.7. Deeper learning

Although state-of-the-art training methods [4,6,7,9] are not evaluated on GNN models deeper than 3 layers, adding more layers in a neural network is proven to be very effective in increasing the expressive power (and thus accuracy) of the network [22]. Here we evaluate the efficiency and overall training speedup of our GNN implementation compared with [7], under various number of layers using 40 processors. The evaluation is based on our C++ implementation.

We first evaluate the computation efficiency. As discussed in Section 3.2, layer sampling based training methods such as [7] suffer from "neighbor explosion". Therefore, on deep models, there may be significant amount of redundant computation across training iterations. Recall that we analyze the per epoch computation complexity in Section 3.2, under the two cases of large and small batch sizes respectively. Fig. 7 shows the severity of "neighbor explosion" by visualizing the number of sampled nodes per GNN layer for the two training methods. Denote L as number of graph convolution layers. The minibatch sampling of [7] proceeds as follows. [7] first randomly pick the r number of root nodes

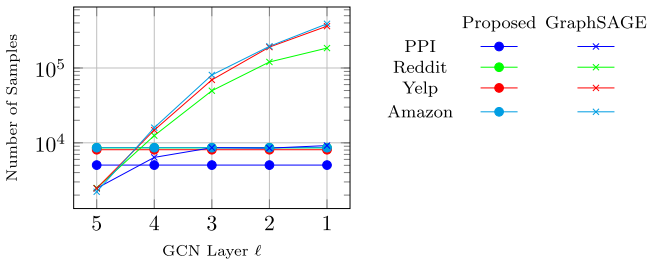

Fig. 7. Comparison on the number of sampled nodes per GNN layer..

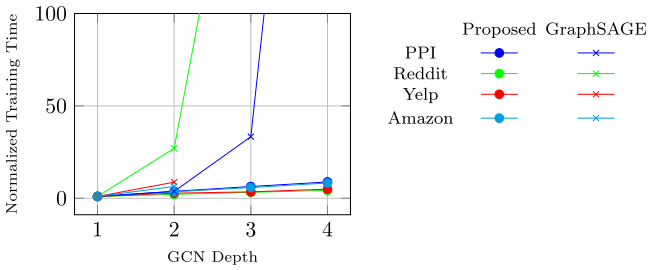

Fig. 8. Comparison of training time on deep GNN models..

from the output of the last graph convolution layer (i.e., layer-L). Then, to generate the layer $\ell-1$ samples, it randomly pick $s^{(\ell)}$ neighbors of each layer ℓ sampled nodes. [7] completes the minibatch construction when it has finished picking the input nodes of layer 1. Following the recommended setting of [7], we set $r = 512$, $s^{(L)} = 25$ and $s^{(\ell)} = 10$ for $1 \leq \ell \leq L-1$. Regarding our proposed training algorithm, since the sampling is performed on the training graph rather than the GNN, all layers have the same $|\mathcal{V}_s|$ number of nodes. Fig. 7 shows the number of unique sampled nodes per layer for the two training methods. When the GNN model is deep, [7] requires orders of magnitude more samples than our training method. In addition, the number of sampled nodes of [7] eventually converges to the full graph size $|\mathcal{V}|$ when the GNN depth is high. In summary, Fig. 7 empirically verifies the complexity analysis in Section 3.2 and shows the advantage in high training efficiency of our method.

We further compare the overall training time for deep GNN models. As shown in Fig. 8, we increase the GNN depth from $L = 1$ to $L = 4$, and set the sampling parameters as described in the above paragraph. Execution of both training methods uses all the 40 processing cores. We do not consider the difference in convergence rate and thus only measures the per-iteration execution time. We normalize the training time by setting the 1-layer GNN execution time as 1. When $L \geq 3$, the implementation of [7] results in prohibitively high training cost on PPI and Reddit, and throws runtime error on Yelp and Amazon. On the other hand, the training time of our method scales almost linearly with respect to the model depth. We conclude that our minibatch training algorithm, together with the parallelization and scheduling techniques, significantly facilitate the development and deployment of deeper GNN models.

8. Discussion

This work proposed co-design of the GNN minibatch training algorithm and the corresponding parallelization strategy. We next discuss potential extensions to our parallel training algorithm.

Hardware acceleration. Our minibatch training algorithm can be used to facilitate hardware accelerator design as well. Apart from higher computation efficiency, another benefit of constructing minibatches by subgraphs is the reduction in communication cost. Suppose we use a resource-constrained hardware accelerator (e.g., FPGA) to speedup GNN training. We can sample small subgraphs so that the features of the subgraph nodes fit in the on-chip memory (whose typical size is tens of mega bits). Each iteration, once the input node features of the subgraph is transferred on-chip, the FPGA can perform the full forward and backward propagation without any communication to the external DDR memory. Therefore, we potentially achieve close-to-peak computation performance on the FPGA. The work in [29] has developed a high-performance accelerator on the CPU-FPGA heterogeneous platform using our graph sampling based training algorithm. They quantify the feasibility of implementing the various training algorithms [4,6,7,9] on hardware by a metric called computation–communication ratio γ, where higher value of γ indicates lower overhead in external memory communication. They further show that our algorithm achieves significantly higher γ than the other methods [4,6,7,9].

Distributed processing. The graph sampling based minibatch training is suitable to be executed in the distributed environment. After partitioning the training graph in distributed memory, each processing node can perform graph sampling independently on the local partition. Afterwards, forward and backward propagation can be executed without data access to the remote memory. In order to ensure convergence quality, shuffling of the node and edge data is required during the training. The optimal shuffling probability may then be derived given the graph sampling algorithm and the connectivity among the processing nodes. It is worth noticing that on each processing node, we can still locally speedup the forward and backward layer computation by designing hardware accelerators or using the parallelization strategy shown in this paper.

9. Conclusion and future work

We presented an accurate, efficient and scalable GNN training method. Considering the redundant computation incurred in state-of-the-art GNN training, we proposed a graph sampling-based minibatch algorithm which ensures accuracy and efficiency by resolving the "neighbor explosion" challenge. We further proposed parallelization techniques and a runtime scheduler to scale the graph sampling and overall training to large number of processors.

We will extend our graph sampling based training by integrating other graph sampling algorithms and evaluating their impact on learning accuracy. We will also work on the theoretical foundation to understand the convergence property of the graph sampling based minibatch training.

CRediT authorship contribution statement

Hanqing Zeng: Conceptualization, Methodology, Software, Writing - original draft, Writing - review & editing, Formal analysis, Visualization, Investigation. **Hongkuan Zhou:** Conceptualization, Methodology, Software, Writing - original draft, Writing - review & editing, Visualization, Data curation, Investigation. **Ajitesh Srivastava:** Conceptualization, Methodology, Formal analysis, Writing - original draft, Writing - review & editing. **Rajgopal Kannan:** Methodology, Writing - original draft, Supervision. **Viktor Prasanna:** Methodology, Writing - original draft, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work is supported by the U.S. National Science Foundation (NSF) under grants OAC-1911229 and CCF-1919289.

References

[1] S. Abu-El-Haija, B. Perozzi, A. Kapoor, H. Harutyunyan, N. Alipourfard, K. Lerman, G.V. Steeg, A. Galstyan, Mixhop: Higher-order graph convolutional architectures via sparsified neighborhood mixing, CoRR abs/1905.00067 (2019) arXiv:1905.00067.

[2] S. Beamer, K. Asanovic, D. Patterson, Reducing pagerank communication via propagation blocking, in: 2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2017, http://dx.doi.org/10.1109/IPDPS.2017.112.

[3] D. Bruening, DynamoRIO: Dynamic instrumentation tool platform.

[4] J. Chen, T. Ma, C. Xiao, FastGCN: Fast learning with graph convolutional networks via importance sampling, in: International Conference on Learning Representations, ICLR, 2018.

[5] Z.-M. Chen, X.-S. Wei, P. Wang, Y. Guo, Multi-label image recognition with graph convolutional networks, 2019, arXiv:1904.03582.

[6] J. Chen, J. Zhu, L. Song, Stochastic training of graph convolutional networks with variance reduction, 2017, ArXiv preprint arXiv:1710.10568.

[7] W. Hamilton, Z. Ying, J. Leskovec, Inductive representation learning on large graphs, in: Advances in Neural Information Processing Systems (NIPS), 2017, pp. 1024–1034.

[8] P. Hu, W.C. Lau, A survey and taxonomy of graph sampling, 2013, ArXiv preprint arXiv:1308.5865.

[9] W. Huang, T. Zhang, Y. Rong, J. Huang, Adaptive sampling towards fast graph representation learning, in: Advances in Neural Information Processing Systems, 2018, pp. 4558–4567.

[10] Intel MKL, https://software.intel.com/en-us/mkl, (Accessed 12 October 2018).

[11] N.S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, P.T.P. Tang, On large-batch training for deep learning: Generalization gap and sharp minima, CoRR abs/1609.04836 (2016) arXiv:1609.04836.

[12] T. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, in: International Conference on Learning Representations, ICLR, 2016.

[13] K. Lakhotia, R. Kannan, V. Prasanna, Accelerating pagerank using partition-centric processing, in: 2018 USENIX Annual Technical Conference (USENIX ATC 18), USENIX Association, Boston, MA, 2018.

[14] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, Z. Ghahramani, Kronecker graphs: An approach to modeling networks, J. Mach. Learn. Res. 11 (Feb) (2010) 985–1042.

[15] J. Leskovec, C. Faloutsos, Sampling from large graphs, in: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2006, pp. 631–636.

[16] J. Leskovec, J. Kleinberg, C. Faloutsos, Graphs over time: densification laws, shrinking diameters and possible explanations, in: Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, 2005, pp. 177–187.

[17] Matrix chain multiplication, http://faculty.cs.tamu.edu/klappi/csce629-f17/csce411-set6c.pdf, (Accessed 10 July 2020).

[18] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, in: Advances in Neural Information Processing Systems, 2013.

[19] B. Ribeiro, D. Towsley, Estimating and sampling graphs with multidimensional random walks, in: Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, IMC '10, http://dx.doi.org/10.1145/1879141.1879192.

[20] A. Roy, I. Mihailovic, W. Zwaenepoel, X-stream: Edge-centric graph processing using streaming partitions, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.

[21] SNAP datasets, http://snap.stanford.edu/graphsage/#datasets.

[22] M. Telgarsky, Benefits of depth in neural networks, in: V. Feldman, A. Rakhlin, O. Shamir (Eds.), 29th Annual Conference on Learning Theory, in: Proceedings of Machine Learning Research, PMLR.

[23] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, Y. Bengio, Graph attention networks, 2017, arXiv:1710.10903.

[24] A.J. Walker, An efficient method for generating discrete random variables with general distributions, ACM Trans. Math. Software 3 (3) (1977) 253–256.

[25] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, Y. Jiang, Knightking: a fast distributed graph random walk engine, in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019, pp. 524–537.

[26] Yelp 2018 challenge, https://www.yelp.com/dataset.

[27] R. Ying, R. He, K. Chen, P. Eksombatchai, W.L. Hamilton, J. Leskovec, Graph convolutional neural networks for web-scale recommender systems, in: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2018, pp. 974–983.

[28] B. Yu, H. Yin, Z. Zhu, Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting, in: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, International Joint Conferences on Artificial Intelligence Organization, 2018, http://dx.doi.org/10.24963/ijcai.2018/505.

[29] H. Zeng, V. Prasanna, Graphact: Accelerating GCN training on CPU-FPGA heterogeneous platforms, in: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, in: FPGA '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 255–265, http://dx.doi.org/10.1145/3373087.3375312.

[30] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, V. Prasanna, GraphSAINT: Graph sampling based inductive learning method, in: International Conference on Learning Representations, 2020, https://openreview.net/forum?id=BJe8pkHFwS.

[31] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, V. Prasanna, Accurate, efficient and scalable graph embedding, in: 2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2019, pp. 462–471, http://dx.doi.org/10.1109/IPDPS.2019.00056.

[32] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, W. Zheng, Exploring the hidden dimension in graph processing, in: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 16.

Hanqing Zeng received the B.Eng degree in electronic engineering from the University of Hong Kong in 2016. He is currently a Ph.D. candidate in Computer Engineering at University of Southern California. His research interests include large scale graph representation learning, parallel and distributed computing, and algorithm-architecture co-optimization for deep learning.

Hongkuan Zhou is currently a Ph.D. student in Computer Engineering at University of Southern California. His research interests include data analytics, parallel and distributed systems, and acceleration of deep neural network. He received his BS degree in Electrical and Computer Engineering from University of Michigan-Shanghai Jiaotong University Joint Institute and his MS degree in Computer Engineering from University of Southern California. He is a student member of the IEEE and ACM.

Ajitesh Srivastava is a Senior Research Associate at University of Southern California. His research interests include Social Networks, Algorithms, Parallel Computing, and Machine Learning applied to social good, crime, smart grids, and computer architecture

Rajgopal Kannan received the B.Tech degree in computer science and engineering from IIT Bombay, in 1991 and the Ph.D. degree in computer science from the University of Denver, in 1996. He is currently a computer scientist at the Army Research Lab in the Computing Architectures Branch and a research adjunct professor in electrical engineering at the University of Southern California. He was formerly a professor with the Department of Computer Science, Louisiana State University (2000–2015). His academic research was funded by DARPA, NSF and DOE and he has published more than 150 research papers in international journals and conferences with two patents awarded in the area of network optimization. His research interests are at the intersection of graph analytics, machine learning and edge computing — enabling application acceleration at the edge on low power devices, for example using Software-Defined Memory for memory bound applications. He is also interested in cyber–physical systems, especially data-driven models and analytics driving Smartgrid optimization and control.

Viktor K. Prasanna received the BS degree in electronics engineering from Bangalore University, the MS degree from the School of Automation, Indian Institute of Science, and the Ph.D. degree in computer science from Pennsylvania State University. He is Charles Lee Powell chair in engineering in the Ming Hsieh Department of Electrical Engineering and professor of computer science with the University of Southern California (USC). His research interests include high performance computing, parallel and distributed systems, reconfigurable computing, and embedded systems. He is the executive director of the USC-Infosys Center for Advanced Software Technologies (CAST) and was an associate director of the USC Chevron Center of Excellence for Research and Academic Training on Interactive Smart Oilfield Technologies (Cisoft). He also serves as the director of the Center for Energy Informatics, USC. He served as the editor-in-chief of the IEEE Transactions on Computers during 2003–06. Currently, he is the editor-in-chief of the Journal of Parallel and Distributed Computing. He was the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is the steering cochair of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) and is the steering chair of the IEEE International Conference on High Performance Computing (HiPC). He received the 2009 Outstanding Engineering Alumnus Award from the Pennsylvania State University. He received the W. Wallace McDowell Award from the IEEE Computer Society, in 2015 for his contributions to reconfigurable computing. His work on regular expression matching received one of the most significant papers in FCCM during its first 20 years award in 2013. He is a fellow of the IEEE, the ACM, and the American Association for Advancement of Science (AAAS).