Accurate, Efficient and Scalable Graph Embedding

Hanqing Zeng[*1], Hongkuan Zhou[*1], Ajitesh Srivastava[1], Rajgopal Kannan[2], and Viktor Prasanna[1]

[1]Univerisity of Southern California, {zengh,hongkuaz,ajiteshs,prasanna}@usc.edu
[2]US Army Research Lab-West, rajgopal.kannan.civ@mail.mil

Abstract—The Graph Convolutional Network (GCN) model and its variants are powerful graph embedding tools for facilitating classification and clustering on graphs. However, a major challenge is to reduce the complexity of layered GCNs and make them parallelizable and scalable on very large graphs — state-of the art techniques are unable to achieve scalability without losing accuracy and efficiency. In this paper, we propose novel parallelization techniques for graph sampling-based GCNs that achieve superior scalable performance on very large graphs without compromising accuracy. Specifically, our GCN guarantees work-efficient training and produces order of magnitude savings in computation and communication. To scale GCN training on tightly-coupled shared memory systems, we develop parallelization strategies for the key steps in training: For the graph sampling step, we exploit parallelism within and across multiple sampling instances, and devise an efficient data structure for concurrent accesses that provides theoretical guarantee of near-linear speedup with number of processing units. For the feature propagation step within the sampled graph, we improve cache utilization and reduce DRAM communication by data partitioning. We prove that our partitioning strategy is a 2-approximation for minimizing the communication time compared to the optimal strategy. We demonstrate that our parallel graph embedding outperforms state-of-the-art methods in scalability (with respect to number of processors, graph size and GCN model size), efficiency and accuracy on several large datasets. On a 40-core Xeon platform, our parallel training achieves $64\times$ speedup (with AVX) in the sampling step and $25\times$ speedup in the feature propagation step, compared to the serial implementation, resulting in a net speedup of $21\times$. Our scalable algorithm enables deeper GCN, as demonstrated by $1306\times$ speedup on a 3-layer GCN compared to Tensorflow implementation of state-of-the-art.

Index Terms—Graph Embedding; Graph Convolutional Networks; Subgraph Sampling;

I. Introduction

Graph embedding is a powerful dimensionality reduction technique. Taking an unstructured, attributed graph as input, the embedding process outputs structured vectors which capture information of the original graph. Embedding facilitates data mining on graphs, and thus is essential in a wide range of tasks such as content recommendation and protein function prediction. Among the various embedding techniques, Graph Convolutional Network (GCN) [1] and its variants [2], [3] have attained much attention recently. GCN based approaches produce accurate and robust results without the need of manual feature selection.

In order to scale GCN to large graphs, the typical approach is to decompose training into "mini-batches" and attempt to parallelize mini-batch training by sampling on GCN layers (layer sampling). The batched GCN [1] and its successor GraphSAGE [2] sample the inter-layer edge connections. Their approaches preserve the training accuracy of the original GCN model, but their parallelization strategy is not work-efficient. The amount of redundant computation increases by a factor of node degree for every additional layer of GCN. To alleviate such high redundancy due to "neighbor explosion" in deeper layers, FastGCN [3] proposes to sample the nodes of GCN layers instead of the edge connections. Although this approach is empirically faster than [1], [2], it does not guarantee work-efficiency, incurs accuracy loss and requires expensive preprocessing which affects scaling.

Due to the layer sampling design philosophy, it is difficult for state-of-the-art methods [1]–[3] to simultaneously achieve accuracy, efficiency and scalability. In this work, we propose a new graph sampling-based GCN which achieves superior performance on a variety of large graphs. Our novelty lies in designing scalable GCN model based on *parallelized graph sampling* (rather than layer sampling), *without compromising* accuracy or efficiency. We achieve scalability and performance by 1) Developing a novel data structure that enables efficient parallel subgraph sampling through fast parallel updates to degree distributions; 2) Developing an optimized parallel implementation of intra-subgraph feature propagation and weight updates — specifically a cache-efficient partitioning scheme that reduces DRAM communication, resulting in optimized memory performance. We achieve work-efficiency by avoiding neighbor explosion, as each layer of our complete GCN consists of only small set of nodes. Finally, since our sampled subgraphs preserve connectivity characteristics of the original training graph, we demonstrate that accuracy also remains unaffected. The main contributions of this paper are:

- We propose parallel training algorithm for a novel graph sampling-based GCN model, where:
  - *Accuracy* is achieved due to connectivity-preserving graph sampling techniques.
  - *Efficiency* is optimal since "neighbor explosion" in layer sampling based GCNs is avoided.
  - *Scalability* is achieved with respect to number of processing cores, graph size and GCN depth by extracting parallelism at various key steps.
- We propose a novel data structure that accelerates degree distribution based sampling through fast incremental parallel updates. Our method has a near-linear scalability
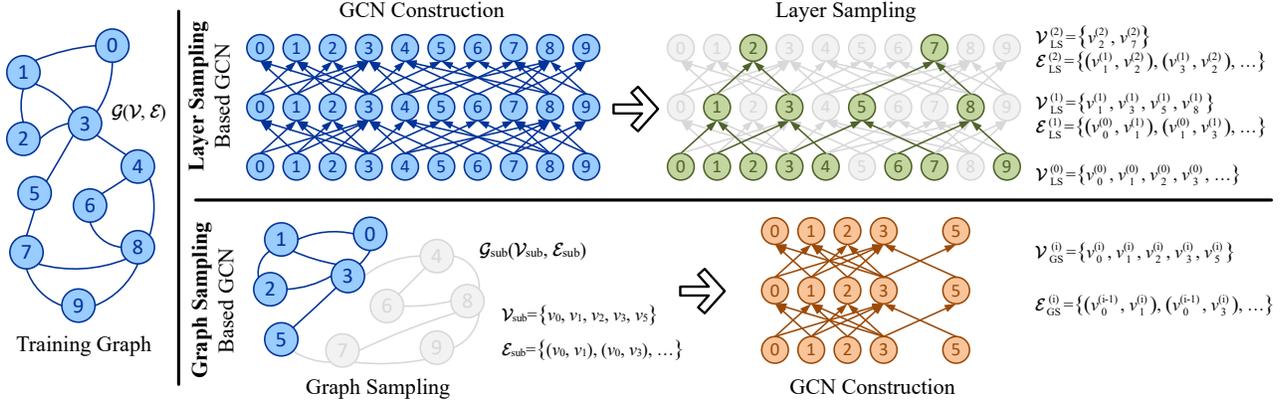
Fig. 1: Illustration on layer sampling and graph sampling based GCN design.

guarantee with respect to number of processing units.

- We parallelize the key training step of subgraph feature propagation. Using intelligent partitioning on the feature dimension, we optimize DRAM and cache performance and scale training to large number of cores.
- We perform thorough evaluation on a 40-core Xeon server. Compared with serial implementation, we achieve $21\times$ overall training time speedup. Compared with other GCN based methods, our parallel (serial) implementation achieves up to $37.4\times$ ($7.8\times$) training time reduction when reaching the same accuracy.
- We demonstrate that our implementation can enable deeper GCN. We obtain $1306\times$ speedup for 3-layer GCN compared to Tensorflow implementation of state-of-the-art on a 40-core machine.

II. BACKGROUND AND RELATED WORK

Graph Convolutional Network (GCN) along with its variations is the state-of-the-art embedding method, widely shown to yield highly accurate and robust results. A GCN is a kind of multi-layer neural network, and it performs vertex embedding as follows. An input graph with vertex attributes (or features) is fed into the GCN. The GCN propagates the features layer by layer, where each layer performs feature extraction based on the learned weight matrices and the input graph topology. The last GCN layer outputs embedding vectors for the vertices of the input graph. Essentially, both the input graph attributes and topology are "embedded" into the output vectors.

Figure 1 shows the network structure of GCNs. We use superscript "(ℓ)" to denote GCN layer-ℓ parameters. An L-layer GCN upon the input graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is constructed as follows. Each layer consists of $|\mathcal{V}|$ nodes — layer node $v_i^{(\ell)} \in \mathcal{V}^{(\ell)}$ corresponds to the graph vertex $v_i \in \mathcal{V}$. Edges in GCN connect nodes in adjacent layers based on the input graph edges. Specifically, the collection of edges between layers $\ell-1$ and ℓ are defined by $\mathcal{E}^{(\ell)} \in \left\{ \left(v_i^{(\ell-1)}, v_j^{(\ell)} \right) \middle| (v_i, v_j) \in \mathcal{E} \right\}$. Note that $\mathcal{V}^{(\ell-1)}$, $\mathcal{V}^{(\ell)}$ and $\mathcal{E}^{(\ell)}$ form a bipartite graph represented by bi-adjacency matrix $\boldsymbol{A}^{(\ell)}$. Each GCN node $v_i^{(\ell)}$ is associated with a feature vector $\boldsymbol{h}_i^{(\ell)} \in \mathbb{R}^{f^{(\ell)}}$ of length $f^{(\ell)}$, and let $\boldsymbol{H}^{(\ell)} \in \mathbb{R}^{|\mathcal{V}| \times f^{(\ell)}}$ be the feature matrix. Note that $\boldsymbol{H}^{(0)}$ consists of vertex attributes of input graph $\mathcal{G}$.

The propagation of $\boldsymbol{h}_i^{(\ell)}$ from layer $\ell-1$ to ℓ is processed by two weight matrices: $\boldsymbol{W}_{\text{self}}^{(\ell)}$ and $\boldsymbol{W}_{\text{neigh}}^{(\ell)}$. Both weight matrices are learned during training. The forward propagation of GCN from layer $\ell-1$ to ℓ involves two major steps:

1) *Feature aggregation*: Each $v_i^{(\ell)}$ collects features of its layer $\ell-1$ neighbors, $\left\{ \boldsymbol{h}_j^{(\ell-1)} \middle| \left(v_j^{(\ell-1)}, v_i^{(\ell)} \right) \in \mathcal{E}^{(\ell)} \right\}$, and calculates the mean of neighbor features.
2) *Weight application*: The mean-aggregated neighbor features are multiplied by $\boldsymbol{W}_{\text{neigh}}^{(\ell)}$. The self-features $\boldsymbol{h}_i^{(\ell-1)}$ from layer $\ell-1$ are also multiplied by the $\boldsymbol{W}_{\text{self}}^{(\ell)}$.

The features after applying weights are sent to layer ℓ. With an optional neighbor-self feature concatenation and non-linear activation, we obtain the next layer features $\boldsymbol{H}^{(\ell)}$.

To train GCNs on large scale graphs, it is essential to do mini-batch training, where number of vertices involved in each weight update is much smaller than the training graph size. Batched GCN [1], GraphSAGE [2] and FastGCN [3] incorporate various GCN *layer sampling* techniques to construct mini-batches. Upper part of Figure 1 abstracts the edge based layer sampling technique [2] and node based layer sampling technique [3], where subscript LS dentes **L**ayer **S**ampling. Since layer sampling requires each node in layer ℓ to select multiple neighbor nodes in layer $\ell-1$, the deeper the layer sampler goes, the more nodes will be sampled (i.e., $\left| \mathcal{V}_{\text{LS}}^{(\ell)} \right|$ becomes larger when ℓ gets smaller). We refer to this phenomenon as "neighbor explosion". On the other hand, [3] proposes a node based layer sampler. It samples in a two phased fashion. The first phase samples nodes of all the L layers based on a pre-calculated probability distribution, where the samplers of each layer are independent. The second phase re-constructs the inter-layer edges to connect the sampled nodes. Empirically, this method mitigates the "neighbor explosion" problem at the cost of accuracy loss and potentially expensive pre-processing (calculating probability distribution). Deeper layers may still require larger sampling population to avoid overly sparse inter-layer connection.

III. Graph Sampling-Based GCN Model

We present a novel graph sampling-based GCN model. Our parallel mini-batch training outperforms state-of-the-art GCN in accuracy, efficiency and scalability, simultaneously. We present the design of the graph sampling-based GCN (Section III-A), and analyze the advantages in efficiency (Section III-B) and accuracy (Section III-C). We then present optimizations to scale training on parallel machines (Sections IV, V).

A. Design of the Model

As illustrated by the lower part of Figure 1, the graph sampling-based approach does not construct a GCN directly on the original input graph $\mathcal{G}$. Instead, for each iteration of weight update during training, we first sample a small induced subgraph $\mathcal{G}_{\text{sub}}(\mathcal{V}_{\text{sub}}, \mathcal{E}_{\text{sub}})$ from $\mathcal{G}(\mathcal{V}, \mathcal{E})$. We then construct a *complete* GCN on $\mathcal{G}_{\text{sub}}$. The forward and backward propagation are both on this small yet complete GCN. Algorithm 1 presents the details of our approach, where subscript GS denotes **G**raph **S**ampling. The key distinction from traditional GCNs 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 (Section III-B). We also discuss the requirement for the SAMPLE$_G$ function (line 3) in Section III-B, and present a representative sampling algorithm that leads to high accuracy.

Algorithm 1 Training algorithm for mini-batched GCN using graph sampling techniques

---

Input: Training graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{H}^{(0)})$; Labels $\boldsymbol{L}$
Output: Trained weights $\left\{ \boldsymbol{W}_{\text{self}}^{(\ell)}, \boldsymbol{W}_{\text{neigh}}^{(\ell)} \mid 1 \leq \ell \leq L \right\}$

1: **while** not terminate **do**   ▷ Iterate over mini-batches
2:      $\mathcal{G}_{\text{sub}}(\mathcal{V}_{\text{sub}}, \mathcal{E}_{\text{sub}}) \leftarrow \text{SAMPLE}_G\big(\mathcal{G}(\mathcal{V}, \mathcal{E})\big)$
3:      $\left\{ \mathcal{V}_{\text{GS}}^{(\ell)} \right\}, \left\{ \mathcal{E}_{\text{GS}}^{(\ell)} \right\} \leftarrow$ GCN construction on $\mathcal{G}_{\text{sub}}$
4:      $\left\{ \boldsymbol{A}_{\text{GS}}^{(\ell)} \right\} \leftarrow$ bi-adj matrix of $\left\{ \left(\mathcal{V}_{\text{GS}}^{(\ell-1)}, \mathcal{V}_{\text{GS}}^{(\ell)}, \mathcal{E}_{\text{GS}}^{(\ell)} \right) \right\}$
5:      $\boldsymbol{H}_{\text{GS}}^{(0)} \leftarrow \boldsymbol{H}^{(0)} \left[\mathcal{V}_{\text{GS}}^{(0)} \right]$
6:      **for** $\ell = 1$ to L **do**   ▷ Forward propagation
7:          $\boldsymbol{H}_{\text{neigh}} \leftarrow \left(\boldsymbol{A}_{\text{GS}}^{(\ell)} \right)^T \cdot \boldsymbol{H}_{\text{GS}}^{(\ell-1)} \cdot \boldsymbol{W}_{\text{neigh}}^{(\ell)}$
8:          $\boldsymbol{H}_{\text{self}} \leftarrow \boldsymbol{H}_{\text{GS}}^{(\ell-1)} \cdot \boldsymbol{W}_{\text{self}}^{(\ell)}$
9:          $\boldsymbol{H}_{\text{GS}}^{(\ell)} \leftarrow \sigma \left(\boldsymbol{H}_{\text{neigh}} | \boldsymbol{H}_{\text{self}} \right)$ ▷ Concat & activation
10:      ▷ Label prediction by embedding; SGD weight update
11:      $\boldsymbol{X} \leftarrow \text{PREDICT}\big(\boldsymbol{H}_{\text{GS}}^{(L)}\big)$
12:      $C \leftarrow \text{LOSS}\left(\boldsymbol{X}, \boldsymbol{L}\left[\mathcal{V}_{\text{GS}}^{(L)} \right] \right)$
13:      $\text{ADAM}\left(C, \boldsymbol{X}, \left\{ \boldsymbol{H}_{\text{GS}}^{(\ell)} \right\}, \left\{ \boldsymbol{W}_{\text{neigh}}^{(\ell)} \right\}, \left\{ \boldsymbol{W}_{\text{self}}^{(\ell)} \right\} \right)$
14: **return** $\left\{ \boldsymbol{W}_{\text{self}}^{(\ell)} \right\}, \left\{ \boldsymbol{W}_{\text{neigh}}^{(\ell)} \right\}$

B. Complexity of Graph Sampling-Based GCN

We analyze the computation complexity of our graph-sampling based GCN and show that it achieves work-optimality. 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 VI).

The main operations to propagate a batch (consisting of vertices of the sampled graph) forward by one layer are:

- Feature aggregation: Each feature vector from layer-ℓ propagates via edges in $\mathcal{E}_{\text{GS}}^{(\ell)}$ to be aggregated in the next layer, leading to $\mathcal{O}\left(\left| \mathcal{E}_{\text{GS}}^{(\ell)} \right| \cdot f^{(\ell)} \right)$ operations.
- Weight application: Each vertex multiplies its feature with the weight, leading to $\mathcal{O}\left(\left| \mathcal{V}_{\text{GS}}^{(\ell)} \right| \cdot f^{(\ell-1)} f^{(\ell)} \right)$ operations.

Complexity of L-layer forward propagation in one batch is:

$$\mathcal{O}\left(\sum_{\ell=0}^{L-1} \left(\left| \mathcal{E}_{\text{GS}}^{(\ell)} \right| \cdot f^{(\ell)} + \left| \mathcal{V}_{\text{GS}}^{(\ell+1)} \right| \cdot f^{(\ell)} \cdot f^{(\ell+1)} \right) \right) \tag{1}$$

The GCN constructed on our sampled subgraph has the same set of nodes in all layers $\left| \mathcal{V}_{\text{GS}}^{(\ell)} \right| = |\mathcal{V}_{\text{sub}}|$. If the average degree of the subgraph is d_{GS}, then number of edges between each layer of GCN $\mathcal{E}_{\text{GS}}^{(\ell)} = d_{\text{GS}}|\mathcal{V}_{\text{sub}}|$. This results in the complexity of a batch (subgraph) of our algorithm to $\mathcal{O}\left(L \cdot |\mathcal{V}_{\text{sub}}| \cdot f \cdot (f + d_{\text{GS}}) \right)$. If we define an epoch as one full traversal of all training vertices (i.e., processing of $|\mathcal{V}|/|\mathcal{V}_{\text{sub}}|$ batches), complexity of an epoch is $\mathcal{O}\left(L \cdot |\mathcal{V}| \cdot f \cdot (f + d_{\text{GS}}) \right)$

Comparison Against Other GCN Models: In [2], a certain number d_{LS} of neighbors are selected from the next layer for each vertex in the current layer. It can be shown that depending on the batch size the complexity falls between:

Case 1 [Small batch size]: $\mathcal{O}\left(d_{\text{LS}}^L \cdot |\mathcal{V}| \cdot f \cdot (f + d_{\text{LS}}) \right)$.

Case 2 [Large batch size] $\mathcal{O}\left(L \cdot |\mathcal{V}| \cdot f \cdot (f + d_{\text{LS}}) \right)$.

We observe that when the batch size is much smaller than the training graph size, layer sampling technique by [2] results in high training complexity (it grows exponentially with GCN depth, although, often small values of L are used). This is the "neighbor explosion" phenomenon, which is effectively due to redundant computations for small batches, making the mini-batch training of [2] very inefficient. On the other hand, when the batch size of [2] becomes comparable to the training graph size, the training complexity grows linearly with GCN depth and training graph size. However, this resolution of the "neighbor explosion" problem comes at the cost of slow convergence and low accuracy [4]. Thus, such training configuration is not applicable to large scale graphs.

Our graph-sampling based GCN technique leads to a parallel training algorithm whose complexity is linear in GCN depth and training graph size. The work-efficiency of our GCN model is guaranteed by design, since the GCN in this case is always complete. In addition, by choosing proper graph sampling algorithms, we can construct subgraphs whose sizes are small, and do not need to be grown with the training graph (as shown in Section VI). Thus, the graph sampling-based

GCN model achieves work-optimality without any sacrifice in accuracy due to large batch sizes.

C. Accuracy of Graph Sampling-Based GCN

By design, 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 $\mathcal{G}_{\text{sub}}$, our graph sampling-based GCN model should absorb all the information in $\mathcal{G}$, and generate accurate embeddings. More specifically, as discussed in Section II, the output vectors "embed" the input graph topology as well as the vertex attributes. A good graph sampling algorithm, thus, should guarantee:

1) Sampled subgraphs preserve the connectivity characteristics in the training graph.
2) Each vertex in the training graph has some non-negligible probability to be sampled;

For the first requirement, while "connectivity" may have several definitions, subgraphs output by the well-known frontier sampling algorithm [5] approximate the original graph with respect to multiple connectivity measures. In addition, the frontier sampling algorithm also satisfies the second requirement. At the beginning of the sampling phase, the sampler picks some initial root vertices uniformly at random from the original graph. These root vertices constitute a significant portion of the sampled graph vertices. Thus, over large enough number of sampling iterations, all input vertex attributes of the training vertices will be covered by the sampler.

For the above reasons, we use frontier sampling algorithm as the sampler for our GCN model. We also perform detailed accuracy evaluation on our GCN model using the frontier sampling algorithm. The results in Section VI prove empirically that our design does not incur any accuracy loss.

IV. Parallel Graph Sampling Algorithm

A. 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 vertex v in FS using 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 vertices $\mathcal{V}_{\text{sub}}$ of $\mathcal{G}_{\text{sub}}$. The sampler repeats the above update process on the frontier set FS, until the size of $\mathcal{V}_{\text{sub}}$ reaches the desired budget n. Algorithm 2 shows the details. According to [5], a good empirical value of m is 1000.

In our experiments sequential implementation, we notice that about half of the time is spent in sampling phase. Therefore, there is a need to parallelize the sampling step. The challenges to parallelization are: 1) While sampling from a discrete distribution is a well-researched problem, we are focused on fast parallel sampling of a *dynamic* degree distribution (changes with addition/deletion of new vertex to the frontier). Existing well-known methods for fast sampling such as aliasing (which can output a sample in $\mathcal{O}(1)$ time with

---

Algorithm 2 Frontier sampling algorithm

Input: Original graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; Frontier size m; Budget for number of sampled vertices n
Output: Induced subgraph $\mathcal{G}_{\text{sub}}(\mathcal{V}_{\text{sub}}, \mathcal{E}_{\text{sub}})$
1: FS $\leftarrow$ m vertices selected uniformly at random from $\mathcal{V}$
2: $\mathcal{V}_{\text{sub}} \leftarrow \{\, v_i \mid v_i \in \text{FS} \,\}$
3: **for** $i = 0$ to $n - m - 1$ **do**
4: $\quad$ Select $u \in \text{FS}$ with probability $\deg(u)/\sum_{v \in \text{FS}} \deg(v)$
5: $\quad$ Select u' from $\{\, w \mid (u, w) \in \mathcal{E} \,\}$ uniformly at random
6: $\quad$ FS $\leftarrow$ $\left(\text{FS} \setminus \{\, u \,\}\right) \cup \{\, u' \,\}$
7: $\quad$ $\mathcal{V}_{\text{sub}} \leftarrow \mathcal{V}_{\text{sub}} \cup \{\, u \,\}$
8: $\mathcal{G}_{\text{sub}} \leftarrow$ Subgraph of $\mathcal{G}$ induced by $\mathcal{V}_{\text{sub}}$
9: **return** $\mathcal{G}_{\text{sub}}(\mathcal{V}_{\text{sub}}, \mathcal{E}_{\text{sub}})$

---

linear processing) cannot be modified easily for this problem. It is non-trivial to select a vertex from evolving FS with low complexity. A straightforward implementation requires $\mathcal{O}(m \cdot n)$ work to sample a single $\mathcal{G}_{\text{sub}}$, which is expensive given $m = 1000$. 2) The sampler is inherently sequential. The vertices in the frontier set should be popped out one at a time to guarantee the quality of the sampled graph $\mathcal{G}_{\text{sub}}$.

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 IV-B). We then propose a training scheduler that exploits parallelization within and across sampler instances (Section IV-C).

B. Dashboard Based Implementation

Since vertices in the frontier set is replaced only one at a time, an efficient implementation should allow incremental update of the probability distribution. We propose a "Dashboard" table to store the status of current and historical frontier vertices (a vertex becomes a historical frontier vertex after it gets popped out of the frontier set). The vertex to be popped out next is selected by probing the Dashboard using randomly generated indices. The data structure ensures that the probability distribution for picking vertices is updated quickly and incrementally. Below we formally describe the data structure and operations in the Dashboard-based sampler. The implementation involves two arrays:

- Dashboard DB $\in \mathbb{R}^{3 \times (\eta \cdot m \cdot d)}$: A table maintaining the status and probabilities of current and historical frontier vertices. An entry corresponding to v_i has 3 slots. The first slot is i; the second slot is an offset value, helping with the invalidation of all of the v_i related entries when v_i is popped out; the third slot is a value k, if v_i is the kth vertex added into DB. 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 when the Dashboard overflows. The jth entry in IA has 2 slots, the first slot records the starting index of the DB entries corresponding to v, where v is the jth vertex added into DB. The second slot is a Boolean flag, which is `True` when v is a current frontier vertex, and `False` when v is a historical frontier vertex.

Since the probability of popping out a vertex in frontier is proportional to its degree, we allocate $\deg(v_i)$ 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. Thus, DB should contain at least $m \cdot d$ entries, where d is the average degree of vertices in frontier. For ease of incremental update of DB we append the entries for the new vertex and invalidate the entry of popped out vertex, instead of replacing one with the other. To accommodate this we introduce an enlargement factor η ($\eta > 1$), 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 current and historical frontier vertices. When this happens, we free up the space occupied by historical frontier vertices, and resume 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 IV-C). 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.

C. Intra- and Inter-Subgraph Parallelization

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

Algorithm 3 shows the details of Dashboard-based parallel frontier sampling, where all arrays are zero-based. Considering the main loop (lines 16 to 25), 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 $1/\left(1 - \left(1 - \frac{1}{\alpha}\right)^p\right)$, where one round refers to one repetition of lines 5 to 7 of Algorithm 4. After selection of v_{pop}, $\deg(v_{\text{pop}})$ number of slots needs to be updated to invalid values INV. Since this operation occurs $(n - m)$ times, the para_POP_FRONTIER function incurs $(n-m)\left(\frac{1}{1-(1-1/\alpha)^p} \cdot \text{COST}_{\text{rand}} + \frac{d}{p} \cdot \text{COST}_{\text{mem}}\right)$ 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), so as 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

---

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 $\mathcal{G}_{\text{sub}}(\mathcal{V}_{\text{sub}}, \mathcal{E}_{\text{sub}})$
1: $d \leftarrow |\mathcal{E}|/|\mathcal{V}|$
2: DB $\leftarrow$ Array of $\mathbb{R}^{3\times(\eta \cdot m \cdot d)}$ with value INV $\triangleright$ INValid
3: IA $\leftarrow$ Array of $\mathbb{R}^{2\times(\eta \cdot m \cdot d+1)}$ with value INV
4: FS $\leftarrow m$ vertices selected uniformly at random from $\mathcal{V}$
5: $\mathcal{V}_{\text{sub}} \leftarrow \{\, v \mid v \in \text{FS} \,\}$
6: FS $\leftarrow$ Indexable list of vertices converted from set FS
7: IA$[0,0] \leftarrow 0$;   IA$[1,0] \leftarrow$ True;
8: **for** $i = 1$ to m **do** $\triangleright$ Initialize IA from FS
9:     IA$[0,i] \leftarrow$ IA$[0, i-1] + \deg(\text{FS}[i-1])$
10:     IA$[1,i] \leftarrow (i \neq m)?$True$:$False
11: **for** $i = 0$ to $m - 1$ **pardo** $\triangleright$ Initialize DB from FS
12:     **for** $k =$ IA$[i]$ to IA$[i+1] - 1$ **do**
13:        DB$[0,k] \leftarrow$ FS$[i]$
14:        DB$[1,k] \leftarrow (k \neq$ IA$[i])?(k -$ IA$[i]) : -\deg(\text{FS}[i])$
15:        DB$[2,k] \leftarrow i$
16: $s \leftarrow m$
17: **for** $i = m$ to $n - 1$ **do** $\triangleright$ Sampling main loop
18:     $v_{\text{pop}} \leftarrow$ pardo_POP_FRONTIER(DB,p)
19:     $v_{\text{new}} \leftarrow$ Vertex sampled from v_{pop}'s neighbors
20:     **if** $\deg(v_{\text{new}}) > \eta \cdot m \cdot d -$ IA$[0,s] + 1$ **then**
21:        DB $\leftarrow$ pardo_CLEANUP(DB, IA, p)
22:        $s \leftarrow m - 1$
23:     pardo_ADD_TO_FRONTIER $(v_{\text{new}}, s, \text{DB}, \text{IA}, p)$
24:     $\mathcal{V}_{\text{sub}} \leftarrow \mathcal{V}_{\text{sub}} \cup \{\, v_{\text{new}} \,\}$
25:     $s \leftarrow s + 1$
26: $\mathcal{G}_{\text{sub}} \leftarrow$ Subgraph of $\mathcal{G}$ induced by $\mathcal{V}_{\text{sub}}$
27: **return** $\mathcal{G}_{\text{sub}}(\mathcal{V}_{\text{sub}}, \mathcal{E}_{\text{sub}})$

DB based on the accumulated shift amount. This translates to $3 \cdot m \cdot d$ 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 of cleanup is $\frac{n-m}{(\eta-1)\cdot m} \cdot \frac{3 \cdot m \cdot d}{p} \cdot \text{COST}_{\text{mem}}$. We ignore the cost of computing the cumulative sum as $\eta m \ll 3md$.

pardo_ADD_TO_FRONTIER: Adding a new frontier v_n to DB requires appending $\deg(v_n)$ new entries to DB. This corresponds to cost of $(n - m) \cdot \frac{3 \cdot d}{p} \cdot \text{COST}_{\text{mem}}$.

Overall cost to sample one subgraph with p processors is:

$$\left(\frac{\text{COST}_{\text{rand}}}{1 - (1 - 1/\eta)^p} + \left(4 + \frac{3}{\eta - 1}\right)\frac{d \cdot \text{COST}_{\text{mem}}}{p}\right) \cdot (n - m) \tag{2}$$

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 $\frac{p}{1+\epsilon}, \forall p \leq \epsilon d \left(4 + \frac{3}{\eta-1}\right) - \eta$.*

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

Algorithm 4 Functions in Dashboard Based Sampler

1: **function** PARDO_POP_FRONTIER(DB, p)
2:     $\text{idx}_{\text{pop}} \leftarrow$ INV   ▷ Shared variable
3:     **for** $j = 0$ to $p - 1$ **pardo**
4:         **while** $\text{idx}_{\text{pop}} ==$ INV **do**   ▷ Probing DB
5:             $\text{idx}_p \leftarrow$ Index generated uniformly at random
6:             **if** DB$[0, \text{idx}_p] \neq$ INV **then**
7:                 $\text{idx}_{\text{pop}} \leftarrow \text{idx}_p$
8:         BARRIER
9:         $v_{\text{pop}} \leftarrow$ DB$[0, \text{idx}_{\text{pop}}]$
10:        $\text{offset} \leftarrow$ DB$[1, \text{idx}_{\text{pop}}]$
11:        $\text{idx}_{\text{start}} \leftarrow (\text{offset} > 0)?(\text{idx}_{\text{start}} - \text{offset}) : \text{idx}_{\text{start}}$
12:        $\deg \leftarrow$ DB$[1, \text{idx}_{\text{start}}]$
13:        **for** $k = j \cdot \frac{\deg}{p}$ to $(j+1) \cdot \frac{\deg}{p} - 1$ **do** ▷ Update DB
14:            DB$[0, k + \text{idx}_{\text{start}}] \leftarrow$ INV
15:        BARRIER
16:     IA$[1,$ DB$[2, \text{idx}_{\text{pop}}]] \leftarrow$ False   ▷ Update IA
17:     **return** v_{pop}
18: **function** PARDO_CLEANUP(DB, IA, p)
19:     DB$_{\text{new}} \leftarrow$ New, empty dashboard
20:     $k \leftarrow$ Cumulative sum of IA$[0, :]$ masked by IA$[1, :]$
21:     **for** $i = 0$ to $p - 1$ **pardo**
22:         Move entries from DB to DB$_{\text{new}}$ by offsets in k
23:     **for** $i = 0$ to $p - 1$ **pardo**
24:         Re-index IA based on DB$_{\text{new}}$
       return DB$_{\text{new}}$
25: **function** PARDO_ADD_TO_FRONTIER($v_{\text{new}}, i,$ DB, IA, p)
26:     IA$[0, i+1] \leftarrow$ IA$[0, i] + d$;   IA$[1, i] \leftarrow$ True;
27:     $\text{idx} \leftarrow$ IA$[0, i]$
28:     $d \leftarrow \deg(v_{\text{new}})$
29:     **for** $j = 0$ to $p - 1$ **pardo**
30:         **for** $k = \text{idx} + j \cdot \frac{d}{p}$ to $\text{idx} + (j+1) \cdot \frac{d}{p} - 1$ **do**
31:             DB$[0, k] \leftarrow n$
32:             DB$[1, k] \leftarrow (k \neq \text{idx})?(k - \text{idx}) : -d$
33:             DB$[2, k] \leftarrow i$

$p \leq \epsilon d(4 + 3/(\eta - 1)) - \eta$, we have $\frac{\eta+p}{p} \leq \frac{\epsilon d(4+3/(\eta-1))}{p}$. Now, speedup obtained by Algorithm 2 compared to a serial implementation ($p = 1$) is

$$\frac{(\eta + d(3/(\eta - 1) + 4))(n - m)}{\left(\frac{1}{1-(1-1/\eta)^p} + \frac{d}{p}(3/(\eta - 1) + 4)\right)(n - m)}$$
$$\geq \frac{d(3/(\eta - 1) + 4)}{\frac{\epsilon d}{p}(3/(\eta - 1) + 4) + \frac{d}{p}(3/(\eta - 1) + 4)} \geq \frac{p}{1 + \epsilon}.$$

$\square$

Setting $\epsilon = 0.5$ and $\eta = 3$, Theorem 1 guarantees good scalability ($p/1.5$) up to $p = 2.25 \cdot d - 3$ processors. While the scalability can be high for dense graphs, it is difficult to scale the sampler to massive number of processors on sparse graphs. The total parallelism available is bound by the graph degree. In summary, the parallel Dashboard based frontier sampler (1) leads to lower serial complexity by incremental update on probability distribution, and (2) scales well up to $p = \mathcal{O}(d)$.

To further scale the sampler, we exploit task parallelism by a GCN training scheduler. Since the topology of the training graph $\mathcal{G}$ is fixed over the training iterations, sampling and GCN computation can proceed in an interleaved fashion, without any dependency constraints. By Algorithm 5, the scheduler maintains 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 $\{ \mathcal{G}_i \}$ with subgraphs independently sampled from the training graph $\mathcal{G}$. Each of the p_{inter} sampler instances runs on p_{intra} processing units. Thus, the scheduler exploits both intra- and inter-subgraph parallelism. In each training iteration, we remove a subgraph $\mathcal{G}_{\text{sub}}$ from $\{ \mathcal{G}_i \}$, and build a complete GCN upon $\mathcal{G}_{\text{sub}}$. Forward and backward propagation stay unchanged as lines 4 to 13 in Algorithm 1.

During construction of $\{ \mathcal{G}_i \}$, total amount of parallelism $p_{\text{intra}} \cdot p_{\text{inter}}$ is fixed on the target platform. The value of p_{intra} and p_{inter} should be 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, note that most of the memory traffic going into DB is in a random manner during sampling. This indicates that ideally, DB should be stored in cache. As a coarse estimate, with $m = 1000$, $\eta = 2$, $d = 30$, the memory consumption by one DB is 480KB [1]. 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 $p_{\text{inter}} = 40$.

Algorithm 5 GCN training with parallel frontier sampler

Input: Training graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{H}^{(0)})$; Labels $\boldsymbol{L}$; Sampler parameters m, n, η; Parallelization parameters $p_{\text{inter}}, p_{\text{intra}}$
Output: Trained weights $\left\{ \boldsymbol{W}_{\text{self}}^{(\ell)}, \boldsymbol{W}_{\text{neigh}}^{(\ell)} \mid 1 \leq \ell \leq L \right\}$
1: $\{ \mathcal{G}_i \} \leftarrow \emptyset$ ▷ Set of unused subgraphs
2: **while** not terminate **do**   ▷ Iterate over mini-batches
3:     **if** $\{ \mathcal{G}_i \}$ is empty **then**
4:         **for** $p = 0$ to $p_{\text{inter}} - 1$ **pardo**
5:             $\{ \mathcal{G}_i \} \leftarrow \{ \mathcal{G}_i \} \cup$ SAMPLE$_{\text{G}}(\mathcal{G}, m, n, \eta, p_{\text{intra}})$
6:     $\mathcal{G}_{\text{sub}} \leftarrow$ Subgraph popped out from $\{ \mathcal{G}_i \}$
7:     $\left\{ \mathcal{V}_{\text{GS}}^{(\ell)} \right\}, \left\{ \mathcal{E}_{\text{GS}}^{(\ell)} \right\} \leftarrow$ GCN construction on $\mathcal{G}_{\text{sub}}$
8:     Forward and backward propagation of GCN
9: **return** $\left\{ \boldsymbol{W}_{\text{self}}^{(\ell)} \right\}, \left\{ \boldsymbol{W}_{\text{neigh}}^{(\ell)} \right\}$

V. PARALLEL TRAINING ALGORITHM

In this section, we present parallelization techniques for the forward and backward propagation. Specifically, the graph sampling based GCN enables a simple partitioning scheme that guarantees close-to-optimal feature propagation performance.

---

[1]We use INT32 for slot 1 of DB, and INT16 for slots 2 and 3.

A. Computation Kernels in Training

Forward and backward propagation involves two kernels:

- Feature propagation in subgraph (step $\left(\boldsymbol{A}_{\mathrm{GS}}^{(\ell)}\right)^{T} \cdot \boldsymbol{H}_{\mathrm{GS}}^{(\ell-1)}$);
- Dense matrix multiplication (steps involving the weight matrices $\boldsymbol{W}_{\mathrm{neigh}}^{(\ell)}$ and $\boldsymbol{W}_{\mathrm{self}}^{(\ell)}$).

We study the parallel feature propagation in Section V-B. As for the computation on dense matrix multiplication, this is a standard BLAS level 2 routine, which can be efficiently parallelized using libraries such as Intel $^{®}$ MKL [6].

B. Parallel Feature Propagation within Subgraph

During training, each of the $\mathcal{V}_{\mathrm{sub}}^{(\ell)}$ nodes pulls features from its neighbors, along the inter-layer edges. Essentially, this can be viewed as feature propagation within $\mathcal{G}_{\mathrm{sub}}$.

A similar problem, label propagation within graphs, has been extensively studied in the literature. State-of-the-art methods based on vertex-centric [7], edge-centric [8] and partition-centric [9] paradigms perform vertex partitioning on graphs so that processors can work independently in parallel. The work in [10] 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 aforementioned techniques may lead to sub-optimal performance in our GCN, due to two reasons:

- The propagated data from each vertex is a long vector rather than a single value label.
- Our graph sizes are small after graph sampling, so partitioning of the graph may not lead to significant advantage.

Below, we analyze the computation and communication costs of feature propagation after graph and feature partitioning. We temporarily ignore the issues of load-balancing and pre-processing overhead, and address these issues later on. For conciseness, we use $\mathcal{G}(\mathcal{V}, \mathcal{E})$ to refer to $\mathcal{G}_{\mathrm{sub}}(\mathcal{V}_{\mathrm{sub}}, \mathcal{E}_{\mathrm{sub}})$.

Suppose we partition $\mathcal{G}$ into Q_v disjoint vertex partitions $\left\{ \mathcal{V}^{(i)} \mid 0 \leq i \leq Q_v - 1 \right\}$. Let the set of vertices that send features to vertices in $\mathcal{V}^{(i)}$ be $\mathcal{V}_{\mathrm{src}}^{(i)} = \left\{ u \mid (u, v) \in \mathcal{E} \wedge v \in \mathcal{V}^{(i)} \right\}$. Note that $\mathcal{V}^{(i)} \subseteq \mathcal{V}_{\mathrm{src}}^{(i)}$, since we follow the design in [2] to add a self-connection to each vertex. We further partition the feature vector $\boldsymbol{h}_v \in \mathbb{R}^f$ of each vertex v into Q_f equal parts $\left\{ \boldsymbol{h}_v^{(i)} \mid 0 \leq i \leq Q_f - 1 \right\}$. Each of the processors is responsible for propagation of $\boldsymbol{H}^{(i,j)} = \left\{ \boldsymbol{h}_v^{(j)} \mid v \in \mathcal{V}_{\mathrm{src}}^{(i)} \right\}$, flowing from $\mathcal{V}_{\mathrm{src}}^{(i)}$ into $\mathcal{V}^{(i)}$ (where $0 \leq i \leq Q_v - 1$ and $0 \leq j \leq Q_f - 1$).

Define the ratio $\gamma_v = \frac{|\mathcal{V}_{\mathrm{src}}^{(i)}|}{|\mathcal{V}|}$. While γ_v depends on the partitioning algorithm, it is always bound by $\frac{1}{Q_v} \leq \gamma_v \leq 1$.

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

In our performance model, we assume p processors operating in parallel. Each processor is associated with a private fast memory (cache). The p processors share a slow memory (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 $\boldsymbol{H}^{(i,j)}$, and propagates

its $\boldsymbol{H}^{(i,j)}$ into $\mathcal{V}^{(i)}$. Due to the irregularity of graph edge connections, access into $\boldsymbol{H}^{(i,j)}$ is random. On the other hand, using CSR format, the neighbor lists of vertices 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 $\boldsymbol{H}^{(i,j)}$ fit into the fast memory;
- Utilize all of the available parallelism;
- Minimize the total computation workload;
- Minimize the total slow-to-fast memory traffic.

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)[2]. Computation and computation over $Q_v \cdot Q_f$ rounds are:

$$
\begin{aligned}
g_{\mathrm{comp}}(Q_v, Q_f) &= n \cdot d \cdot f \\
g_{\mathrm{comm}}(Q_v, Q_f) &= 2 \cdot Q_f \cdot n \cdot d + 8 \cdot Q_v \cdot n \cdot f \cdot \gamma_v \quad (3)
\end{aligned}
$$

Note that $g_{\mathrm{comp}}(Q_v, Q_f)$ is independent of the partitioning scheme. We formulate *communication minimization problem*:

$$
\begin{aligned}
& \underset{Q_v, Q_f}{\text{minimize}} \quad g_{\mathrm{comm}}(Q_v, Q_f) = 2Q_f \cdot nd + 8Q_v \cdot nf\gamma_v \\
& \text{subject to} \quad Q_v Q_f \geq p; \quad \frac{8nf\gamma_v}{Q_f} \leq S_{\mathrm{cache}}; \quad Q_v, Q_f \in \mathbb{Z}^+
\end{aligned}
$$
$$(4)$$

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

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

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

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

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

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

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

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

b) Case 2, $p \leq \frac{8nf}{S_{\mathrm{cache}}}$: In this case, $Q_f = 8nf/S_{\mathrm{cache}}$ forms a feasible solution. And,

$$
\begin{aligned}
g_{\mathrm{comm}}\left(1, \frac{8nf}{S_{\mathrm{cache}}}\right) &= 2nd\frac{8nf}{S_{\mathrm{cache}}} + 8nf \\
&= 8nf\left(\frac{2nd}{S_{\mathrm{cache}}} + 1\right) \leq 8nf \cdot (1 + 1) = 16nf.
\end{aligned}
$$

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

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 $S_{\text{cache}} \geq 2nd$, which means for a cache size of 256KB, number of (directed) edges in the subgraph (nd) can be up to 128K, which is higher than that of the subgraphs in consideration. Also, since $f \gg d$, $p \leq 4f/d$ holds up to large values of p.

□

Using typical values $n \leq 8000$, $f = 512$, and $d = 15$, then for up to 136 cores[3], the total slow-to-fast memory traffic under feature only partitioning is less than 2 times the optimal. Recall the two properties (listed at the beginning of this section) that differentiate our case with 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 6 shows details of our feature propagation.

Algorithm 6 Feature propagation within sampled graph

Input: Sampled graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; Vertex features $\{\, \boldsymbol{h}_v \,\}$; Cache size S_{cache}; Number of processors p
Output: Updated features $\{\, \boldsymbol{h}_v \,\}$
1: $n \leftarrow |\mathcal{V}|$; $\quad f \leftarrow |\boldsymbol{h}_v|$;
2: $Q_f \leftarrow \max\left\{p, \frac{8nf}{S_{\text{cache}}}\right\}$
3: Equal partition each $\boldsymbol{h}_v$ into $\left\{\, \boldsymbol{h}_v^{(i)} \,\middle|\, 0 \leq i \leq Q_f - 1 \,\right\}$
4: **for** $r = 0$ to $Q_f/p - 1$ **do**
5:     **for** $i = 0$ to $p - 1$ **pardo**
6:         Propagation of $\left\{\, \boldsymbol{h}_v^{(i+r\cdot p)} \,\middle|\, v \in \mathcal{V} \,\right\}$ into $\mathcal{V}$
7: **return** $\{\, \boldsymbol{h}_v \,\}$

Lastly, the feature only partitioning leads to two more important benefits. Since graph is not partitioned, load-balancing is optimal across processors. Also, the partitioning along features incurs almost zero pre-processing overhead. In summary, the feature propagation in our graph sampling-based GCN achieves 1) Minimal computation; 2) Optimal load-balancing; 3) Zero pre-processing cost; 4) Low communication volume.

VI. EXPERIMENTS

A. Experimental Setup

We conduct our experiments on 4 large scale graph datasets:

- PPI: A protein-protein interaction graph [11]. A vertex represents a protein and edges represent interactions.
- Reddit: A post-post graph [11]. A vertex represents a post. An edge exists between two posts if the same user has commented on both posts.
- Yelp: A social network graph [12]. A vertex is a user. An edge represents friendship. Vertex attributes are user comments converted from text using Word2Vec [13].

- Amazon: An item-item graph. A vertex is a product sold by Amazon. An edge is present if two items are bought by the same customer. Vertex attributes are converted from bag-of-words of text item descriptions using singular value decomposition (SVD).

The PPI and Reddit datasets are used in [1], [2]. Note that the Reddit graph is currently the largest one evaluated by state-of-the-art embedding methods, such as [2], [3], [14]. We further prepare two graphs of much larger sizes (Yelp, Amazon), to evaluate scaling more thoroughly. Table I shows the details.

TABLE I: Dataset Statistics

Dataset	Vertices	Edges	Attribute	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

* The (M) mark stands for **M**ulti-class classification, while (S) stands for **S**ingle-class.

For our graph sampling-based GCN, we provide two implementations, in `Python` (with Tensorflow) and `C++`, respectively [4]. The two implementations follow an identical training algorithm (Algorithm 5). We use the `Python` (Tensorflow) version for single threaded accuracy evaluation in Section VI-B, since all baseline implementations are provided in `Python` with Tensorflow. We use the `C++` version to measure scalability of our parallel training (Section VI-C). 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`).

We run experiments on a dual 20-Core Intel® Xeon E5-2698 v4 @2.2GHz machine with 512GB of DDR4 RAM. For the `Python` 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, MKL and OMP are included in Intel® Parallel Studio Xe 2018 update 3.

B. Evaluation on Accuracy and Efficiency

Our graph sampling-based GCN model significantly reduces training 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. Figure 2 shows the accuracy (F1 micro score) vs sequential training time plots. Since all baselines report accuracy using 2 layers in their original papers, all measurements here are based on a 2-layer GCN model. Accuracy is tested on the validation set at the end of each epoch. Between the baselines, [2] achieves the highest accuracy and fastest convergence. Compared with [2], our GCN model achieves higher accuracy on PPI and Reddit, and the same accuracy on Yelp and Amazon. To measure training time speedup, we define an accuracy threshold for each of the datasets. Let a_0 be the highest accuracy achieved by the baselines. We define the threshold[5] as $a_0 - 0.0025$. Serial training time speedup is

---

[3]Note that d here refers to the average degree of the sampled graph rather than the training graph. Thus, d value here is lower than that in Section IV.

[4]Code available at `https://github.com/ZimpleX/gcn-ipdps19`
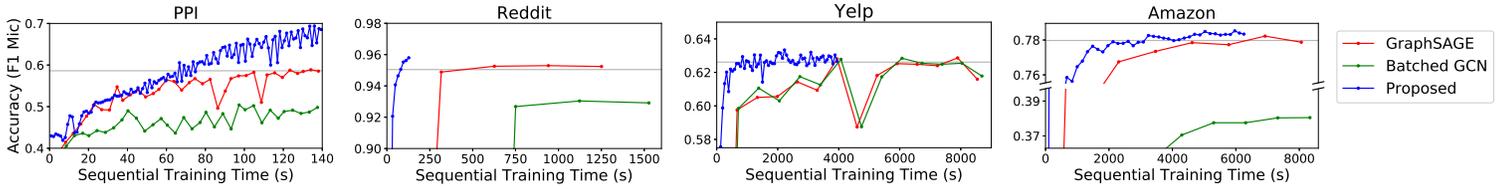[5]Due to stochasticity in training, we allow 0.25% variance in accuracy.

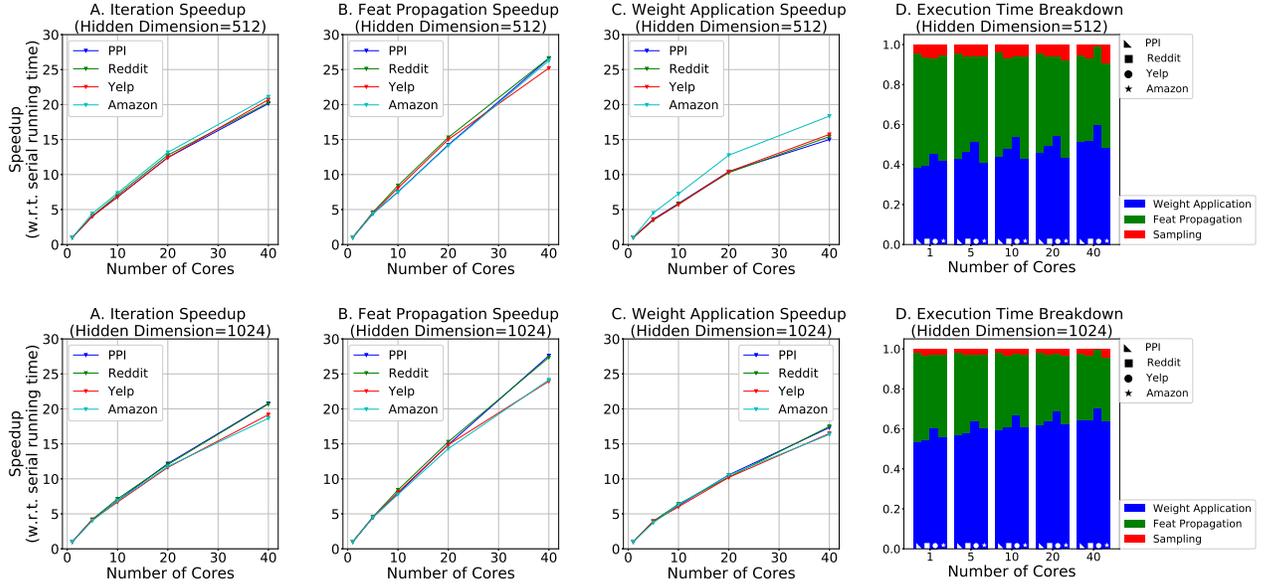Fig. 2: Time-Accuracy plot for sequential execution

Fig. 3: Scaling evaluation with hidden feature dimensions: 512 (Upper), 1024 (Lower)

calculated with respect to training time for the best performing baseline to reach the threshold divided by training time for our model to reach the threshold. We achieve a serial training time speedup of $1.9\times$, $7.8\times$, $4.7\times$ and $2.1\times$ for PPI, Reddit, Yelp and Amazon, respectively.

C. Evaluation on Scalability

1) Scaling of overall training: For our GCN model, Figure 3 shows the parallel training speedup relative to sequential execution. The execution time includes every training steps specified by lines 2 to 8, Algorithm 5 — sampling (with AVX enabled), feature propagation (forward and backward) and weight application (forward and backward). As before, we evaluate scaling on a 2-layer GCN, with small and large hidden dimensions ($f^{(0)} = f^{(1)} = 512$ and 1024). The training is highly scalable, consistently achieving around $20\times$ speedup on 40-cores for all datasets. The performance breakdown in Figure 3 suggests that sampling time corresponds to only a small portion of the total time. This is due to 1) low serial complexity of our Dashboard based implementation; 2) highly scalable design using intra- and inter-subgraph parallelism. The main scaling bottleneck is the weight application step using dense matrix multiplication. To some extent, the scaling is data dependent. High-centrality vertices in the training graph are more likely to be sampled, making caching across training iterations a non-negligible factor. The time on dense matrix multiplication becomes more significant with more processors.

2) Scaling of parallel frontier sampling: We evaluate the effect of both intra- and inter-subgraph parallelism. 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 $1 \leq p_{\text{inter}} \leq 40$. Figure 4A 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 degrades from 20 to 40 cores, due to NUMA effects on cross socket communication — all the p_{inter} samplers on the two sockets keep reading a single copy of the training graph adjacency list, when selecting the next frontier vertices. Figure 4B 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 $4\times$ speedup on average. The scaling on p_{intra} is data dependent. Depending on the degree distribution of the training graph, there may be significant portion of vertices with less than 8 neighbors. Such load-imbalance explains the discrepancy from the theoretical modeling. As a side-note, for highly skewed graphs such as Amazon, we allocate no more than 30 DB entries to a vertex (regardless of its actual degree), in order to upper bound the probability of popping it from the frontier. This helps accuracy improvement on graphs with skewed degree distributions, since it prevents the situation where all subgraphs contain mostly the same set of vertices.
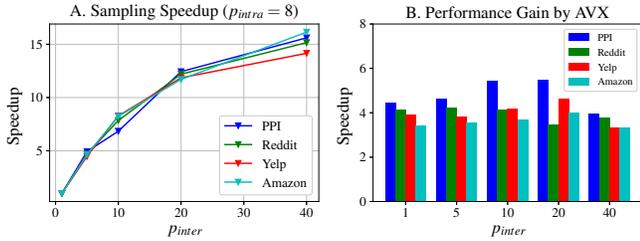
Fig. 4: Sampling speedup (inter- & intra-subgraph parallelism)

3) Scaling of feature propagation: Figure 3 shows the scaling of feature propagation using the partitioning scheme presented in Section V-B. We achieve high scaling (around $25\times$ speedup on 40 cores) for all datasets for various feature sizes, due to our caching strategy and optimal load-balancing.

4) Scaling of weight application: As discussed in Section V-A, the weight application is implemented by `cblas_dgemm` routine of Intel® MKL [6]. All optimizations are internal to the library. Figure 3 shows the scaling result. On 40 cores, average of $16\times$ speedup is achieved. We speculate that the overhead of MKL's internal thread and buffer management is the bottleneck on further scaling.

D. Deeper Learning

Though state-of-the-art methods do not provide accuracy results on deeper GCN models, adding more layers in a neural network is proven to be very effective in increasing the expressive power (and thus accuracy) of the network [15]. Here we evaluate the speedup of our GCN implementation compared with [2], under various number of layers and processors. The speedup is calculated by comparing our `C++` implementation with a `Python` one, whose parallelism is internally highly optimized via Tensorflow. Use Reddit dataset as an example, Table II shows the speedup. Our implementation achieves significantly higher speedup in deeper GCNs ($335\times$ for a 3-layer model, under single thread comparison) as well as with more processors ($37.4\times$ for a standard 2-layer model, using 40 cores). This demonstrates the better scalability of our implementation, and verifies the conclusion in Section III-B. Note that due to "neighbor explosion", for every unit of computation, [2] requires approximately d_{LS} times more communication compared with our model. This explains the scalability results with respect to number of processors. In summary, despite the programming language overheads, the better performance of our design is mainly due to the efficient GCN algorithm and parallelization strategies. Accuracy evaluation for deeper GCN models is out of scope of this paper.

TABLE II: Speedup Comparison with Parallelized [2] (Reddit)

	1-core	5-core	10-core	20-core	40-core
1-layer	$2.03\times$	$4.77\times$	$9.34\times$	$17.25\times$	$23.93\times$
2-layer	$7.74\times$	$12.95\times$	$18.50\times$	$28.43\times$	$37.44\times$
3-layer	$335.36\times$	$568.93\times$	$828.25\times$	$1164.45\times$	$1306.21\times$

VII. CONCLUSION AND FUTURE WORK

We presented an accurate, efficient and scalable graph embedding method. Considering the redundant computation in current GCN training, we proposed a graph sampling-based method which ensures accuracy and efficiency by constructing a new GCN on a small sampled subgraph in every iteration. We further proposed parallelization techniques to scale the graph sampling and overall training to large number of processors.

We will extend our graph sampling-based GCN by evaluating impact on accuracy using various sampling algorithms. We will extend the parallel sampler implementation to support a wider class of sampling algorithms. We will also work on the theoretical foundation of the graph sampling-based GCN.

REFERENCES

[1] T. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations (ICLR)*, 2016.

[2] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[3] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling," in *International Conference on Learning Representations (ICLR)*, 2018.

[4] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima," *CoRR*, vol. abs/1609.04836, 2016.

[5] B. Ribeiro and D. Towsley, "Estimating and Sampling Graphs with Multidimensional Random Walks," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC'10.

[6] "Intel MKL," https://software.intel.com/en-us/mkl, accessed: 2018-10-12.

[7] S. Beamer and K. Asanovic and D. Patterson, "Reducing Pagerank Communication via Propagation Blocking," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017.

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

[9] K. Lakhotia, R. Kannan, and V. Prasanna, "Accelerating PageRank using Partition-Centric Processing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*.

[10] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng, "Exploring the Hidden Dimension in Graph Processing," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[11] "Snap datasets," http://snap.stanford.edu/graphsage/#datasets.

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

[13] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013.

[14] H. Gao, Z. Wang, and S. Ji, "Large-Scale Learnable Graph Convolutional Networks," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD'18.

[15] M. Telgarsky, "Benefits of depth in neural networks," in *29th Annual Conference on Learning Theory*, ser. Proceedings of Machine Learning Research, V. Feldman, A. Rakhlin, and O. Shamir, Eds. PMLR.