CM-GCN: A Distributed Framework for Graph Convolutional Networks using Cohesive Mini-batches

Guoyi Zhao, Tian Zhou and Lixin Gao
Dept. of Electrical and Computer Engineering
University of Massachusetts Amherst
guoyi@umass.edu, tzhou@umass.edu, lgao@engin.umass.edu

2021 IEEE International Conference on Big Data (Big Data) | 978-1-6654-3902-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/BigData52589.2021.9671931

***Abstract*—Graph convolutional network (GCN) has been shown effective in many applications with graph structures. However, training a large-scale GCN is still challenging due to the high computation cost that grows with the size of the graph. In this paper, we propose CM-GCN, a distributed GCN framework using cohesive mini-batches to accelerate large-scale GCN training. The cohesive mini-batches group nodes that are tightly connected in the graph. As a result, CM-GCN can reduce the computation required to train a GCN. We propose a computation cost function to quantify the computation required for mini-batches. By exploring the submodular property of the computation cost function, we develop an efficient algorithm to partition nodes into tightly coupled mini-batches. Based on the computation cost function, we evenly distribute the workloads of mini-batches to workers. We design asynchronous computations between GCN layers to further eliminating the waiting among workers. We implement a CM-GCN framework and evaluate its performance with graphs that contain millions of nodes. Our evaluation shows that CM-GCN can achieve up to 3X speedup without compromising the training accuracy.**

***Index Terms*—graph convolutional networks, mini-batch training, asynchronous computation, graph partitioning**

I. Introduction

Graph convolutional network (GCN) and its variants [18], [4], [26] have achieved state-of-the-art results in many graph-based applications, including node classification [18], link prediction [5], inductive node embedding [9] and recommender systems [28]. The graph convolution operation applies to all the neighbors of nodes to obtain node embeddings. In each convolution layer, the embedding of a node is learned by aggregating its neighborhood embeddings, followed by the same linear transformations and nonlinear activations. After stacking K convolution layers, GCN can learn representative node embeddings by utilizing information from nodes that are within K hops away.

The traditional GCN training on large graphs can be slow because we are only able to update the parameter after processing the full graph. It is also memory-consuming to store all the graph structures especially training on GPUs. Mini-batch stochastic gradient descent (SGD) [9] is proposed to accelerate GCN training from more frequent updates. The updates of the GCN model parameters will focus on the gradients computed from a mini-batch. So mini-batch SGD reduces the memory requirement and conducts several updates per epoch which usually leads to faster convergence. However, mini-batch SGD introduces a significant computational overhead due to the neighborhood expansion. This is because the convolution operation expands the neighborhood nodes in several hops away. The number of dependent nodes grows exponentially when GCN goes deeper.

Several GCN models proposed sampling or clustering strategies to reduce the neighborhood expansion issue. GraphSAGE [9] sampled a fixed number of neighbors to limit the nodes to expand. FastGCN [3] further proposed importance sampling to give different weights for the nodes we sampled. However, the sampling must be done repeatedly in each epoch, and it loses neighborhood information in the convolution operations. Cluster-GCN [5] focused on densely connected clusters in the graph to train GCN. But the edges that are removed between clusters may also lose important neighborhood information. The sampling strategy and removed edges in clustering typically reduce the accuracy of the trained GCNs. Furthermore, although their training often converges in practice, there is no convergence guarantee for trivial sampling methods [4].

GPUs are widely used to train GCNs due to their ability to provide highly parallel computations. But recently CPU clusters have shown appealing training efficiency and low pricing for large graphs [19], [14]. Since the real-world graphs could consist of billions of edges and keep growing [29], [22], it is getting expensive to use multiple GPUs to train GCN. The lowest-configured p3 instance type on AWS has a price of $3.06/h, while an m4.2xlarge instance with 8 vCPU costs only $0.4/h. Not to mention most of the cloud servers provide more flexible pricing options for CPU instances, such as transient resources [31]. NeuGraph [19] and ROC [14] enable multiple GPUs training to improve scalability. However, the memory in GPUs is very limited, which makes it more expensive to scale to billion-edge graphs.

We propose CM-GCN, a distributed GCN framework using cohesive mini-batches to accelerate large-scale GCN training in CPU clusters. The cohesive mini-batches group nodes that are tightly connected in the graph. So these nodes will share the common neighbors to reduce the dependant nodes in GCN training. With the reduction of dependant nodes, we can reduce more than half of the computation required in each epoch. We propose a computation cost function to efficiently calculate

the required computation for each mini-batch. After proving the submodular property of the computation cost function, we develop an efficient algorithm to partition the graph nodes into cohesive mini-batches in polynomial time. Therefore, we can accelerate the large-scale GCN training without compromising the training accuracy.

In CM-GCN, we utilize CPU clusters in GCN training to achieve high scalability and low expense comparing with GPU training. We decompose and perform the computations of embeddings and gradients on CPUs. Since we need to synchronize GCN parameters after each mini-batch, CM-GCN further balances the workloads on workers to avoid waiting. Using balanced workloads, workers can finish processing each mini-batch around the same time. Since processing a node only requires the embeddings or gradients from its neighbors, it is not necessary to synchronize all the embeddings and gradients between GCN layers. Within a mini-batch, CM-GCN enables asynchronous computations to prioritize the processing of nodes that are ready to be processed. So we can parallelize the communication and computation to eliminate the waiting among workers.

We design and implement a framework to support our CM-GCN model and evaluate it with several large-scale graphs with millions of nodes from Reddit and OGB datasets [12]. Comparing with a mini-batch training using the traditional GCN model, CM-GCN achieves comparable training accuracy while we can save more than half of the computation per epoch. With balanced workloads and asynchronous computation between GCN layers, CM-GCN can speedup up to 3X towards GCN.

The remainder of this paper is organized as follows. Section II describes the background of general GCN training and how the mini-batch SGD scheme can be applied in a distributed GCN training. Section III introduces our CM-GCN model on how we partition nodes into cohesive mini-batches and perform the asynchronous computation on the balanced workloads. Section IV reports extensive evaluation results. Section V highlights the related works and Section VI finally concludes our work.

II. Preliminary

In this section, we give a brief background of the graph convolutional network training. We first illustrate the general training procedure of GCN. Then we explain how the mini-batch SGD scheme accelerates the GCN training for large graphs.

A. Graph Convolutional Network Training

The graph convolutional network [24], [18], [24], [6] extends existing neural networks to process data in graph domains. One of the key problems is to learn graph embeddings [10] to represent graph nodes, edges, and sub-graphs. Give a graph $G = (V, E)$, the nodes in V are connected through the edges in E, and each node is associated with a feature vector. A K-layer GCN consists of K convolution layers where each layer k contains parameter $W^{(k)}$. In Layer k, we represent an *embedding* $h_v^k \in \mathbb{R}^{d_k}$ for each node $v \in V$ where d_k is the dimension of embedding. The training of GCN includes forward propagation and backward propagation steps to compute embeddings and gradients to update parameter W. Next, we will discuss the details of forward propagation and backward propagation steps separately.

1) Forward Propagation Step: In neural networks, forward propagation refers to the flow of data which maps input features to the output of the network. In Layer k, each node aggregates its neighborhood embeddings from the previous layer to compute its embedding of current layer as follows.

$$h_v^{(k)} = \sigma(W^{(k)} \sum_{u \in N_v} h_u^{(k-1)}) \tag{1}$$

where N_v is the neighbor set for node v in graph G, and $\sigma(\cdot)$ is an activation function which is usually a Sigmoid or ReLU function. The embeddings are propagated layer by layer through the edges of nodes. In the initial layer 0, the embedding is the input feature as $h_v^0 = x_v$. The final output z_v at Layer K is $z_v = h_v^{(K)}$. The output z_v is used to compute the training loss. The loss function sums up the losses from all labeled nodes V_L as follows.

$$\mathcal{L} = \sum_{v \in V_L} loss(y_v, z_v) \tag{2}$$

where y_v is the given label for node v. In practice, a cross-entropy loss or mean squared error (MSE) is commonly used for node classification in multi-class or multi-label problems.

2) Backward Propagation Step: In the backward propagation step, we compute the gradient of the loss function with respect to the embeddings h and parameter W. The gradient is computed based on the chain rule from the opposite propagation direction in the forward step. The parameter W is then updated using gradient descent. Specifically, we first calculate the gradients in the last layer from the final loss. Take MSE as an example, the gradient of node v is computed as follows.

$$\frac{\partial \mathcal{L}}{\partial z_v} = 2(z_v - y_v) \tag{3}$$

Based on the chain rule, the gradient of the embedding for node v at layer $k < K$ is computed as follows.

$$\frac{\partial \mathcal{L}}{\partial h_v^{(k)}} = \sum_{u \in N_v} \left(\frac{\partial \mathcal{L}}{\partial h_u^{(k+1)}} \times \frac{\partial h_u^{(k+1)}}{\partial h_v^{(k)}} \right) \tag{4}$$

To update $W^{(k)}$, we aggregate the gradients from all the nodes in Layer k and apply them in the gradient descent algorithm.

$$\frac{\partial \mathcal{L}}{\partial W^{(k)}} = \sum_{v \in V} \left(\frac{\partial \mathcal{L}}{\partial h_v^{(k)}} \times \frac{\partial h_v^{(k)}}{\partial W^{(k)}} \right) \tag{5}$$

B. Mini-Batch GCN training

The full-batch gradient descent training scheme was commonly used in GCN training when the graph size is small.

However, full-batch requires storing all the intermediate embeddings, which is not scalable for large graphs. The convergence can be slow since the parameters are updated only once per epoch. Mini-batch SGD scheme [9], [28] is proposed to accelerate the training. The mini-batch GCN training mainly takes three meta steps. First, we randomly select a subset of nodes $B \subset V$ as a mini-batch. Then, we expand the full set or sample a subset of neighbor nodes on each layer to specify the nodes to be trained. At last, we propagate forward and backward among the nodes in each GCN layer. The model parameters in the GCN are updated iteratively via stochastic gradient descent.

III. CM-GCN Model

We propose the CM-GCN model to accelerate the GCN training with distributed cohesive mini-batches. A cohesive mini-batch groups nodes that are tightly connected in the original graph. So we increase the reusability of embedding computation results and reduce the required computation for processing the same number of nodes in the mini-batch. Figure 1 shows an overview of our model with a 2-layer GCN. We first partition nodes into cohesive mini-batches to minimize the computation required for each epoch. Then we distribute the computation of each mini-batch to workers. After each mini-batch, we synchronize the GCN parameters and process the next mini-batch until the model converges.

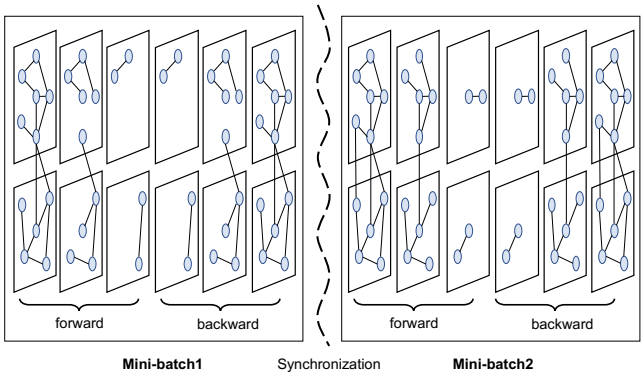


Fig. 1: Overview of CM-GCN

Besides cohesive mini-batches, CM-GCN also balances the workloads on workers and enables asynchronous computation to reduce the waiting caused by synchronizations between GCN layers. As shown in Figure 1, the two workers in the top and bottom have almost the same amount of computation. So they can finish at the same time to avoid waiting at the synchronization barrier. Meanwhile, we break the synchronization between GCN layers. So we can parallelize the communication for passing embeddings and gradients with the node processing to further accelerate the training.

A. Partition Nodes into Cohesive Mini-batches

Using cohesive mini-batches, we increase the utilization of the computed embeddings and gradients in each mini-bath to reduce the computation per epoch. In Figure 2, we show the nodes that are required to be processed in cohesive mini-batches and non-cohesive mini-batches. We use the color red and blue to distinguish different mini-batches. In the cohesive mini-batches, the nodes are tightly coupled in the mini-batch and share neighbors. So the computation of embeddings and gradients can be reused for more nodes. For example, we need to process 13 nodes for the red mini-batch and 16 nodes for the blue mini-batch in the cohesive mini-batches. For simplicity, we assume the computation for each node is the same as 1. Then, the total computation required in cohesive mini-batches is 29 per epoch. In the non-cohesive mini-batches, we need to expend more neighbors in order to compute the same number of training losses. For example, the non-cohesive mini-batches in Figure 2 require the total computation as 35. So the cohesive mini-batches in this simple example can save 17% of the computation. When the GCN goes deeper, we can save much more computation time using cohesive mini-batches.

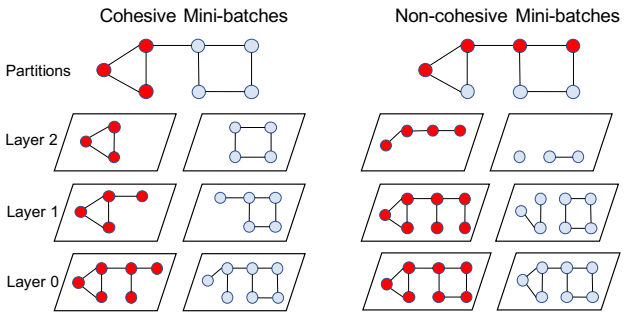


Fig. 2: Cohesive mini-batches and non-cohesive mini-batches GCN training. Color Red and Blue represent the nodes from two mini-batches

Finding the optimal cohesive mini-batches is none-trivial. Clustering nodes that are close to each other into mini-batches can reduce the required computation. But the computation required for each node varies because it is related to the number of its neighbors. Thus, not only the number of nodes but also the computation required determines the quality of mini-batches. In CM-GCN, we propose a computation cost function and partition nodes into mini-batches that minimizes the computation cost.

The computation cost of a mini-batch B is the time required to compute all the embeddings and gradients during the GCN training. The mathematical operations in the training can be represented as a *computational graph* $\mathcal{G}_B$ for mini-batch B. For example, in Figure 3, it is the computational graph from a mini-batch with node A. Given the input graph structure on the left, the computational graph on the right indicates the nodes to be processed in each layer and the directions to propagate the embeddings and gradients. The nodes in the forward step represent the computations of embeddings. The nodes in the backward step indicate the computations of gradients towards the training loss for each corresponding embedding.

For computing embeddings in the forward step, we can directly analyze the required computation in Equation (1). In the backward step, the gradients are computed for both

TABLE I: Notations of constant factors in computation cost function

Factor	Computation	Cost
α	Summation of 2 d_k vectors	αd_k
β	Multiplication between a $d_k \times d_{k-1}$ matrix and a d_{k-1} vector	$\beta d_k d_{k-1}$
γ	Applying activation function $\sigma(\cdot)$ to a d_k vector	γd_k
η	Multiplication between a $d_k \times d_{k-1}$ matrix and a scalar	$\eta d_k d_{k-1}$
λ	Computing the gradient of a loss function towards a d_k vector	λd_k

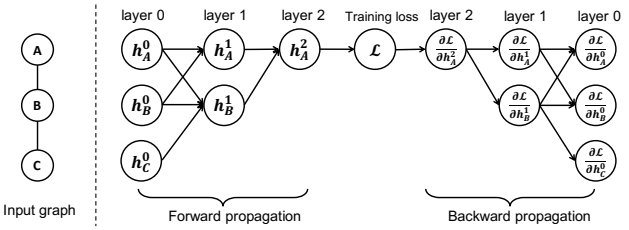


Fig. 3: Computational graph of a mini-batch $\{A\}$

embeddings $h^{(k)}$ and the parameters $W^{(k)}$ in Layer k. To efficiently compute the gradients, we distribute the computation in Equation (4) into different nodes in the computational graph. In this subsection, we will first discuss how to derive the computation cost function in the forward step, backward step, and the whole mini-batches separately. Then we show how CM-GCN partitions nodes into cohesive mini-batches using the computation cost.

1) Computation Cost in Forward Step: In the forward step, the computation in Equation (1) consists of aggregating the embeddings from neighbors $\sum_{u \in N_v} h_u^{(k-1)}$, multiplying the parameter $W^{(k)}$, and applying the activation function $\sigma(\cdot)$. Since the nodes in Layer 0 using the input features as embeddings, the computation cost for node v is in layer 0 is $c_f(v, 0) = 0$.

From Layer 1 to Layer K, we divide the computation cost into three parts. (1) The aggregation of embeddings $\sum_{u \in N_v} h_u^{(k-1)}$ is linear to the number of neighbors $|N_v|$ and the dimension d_{k-1} of previous layer $k-1$. So the computation time as $\alpha |N_v| d_{k-1}$, where α indicates the constant factor of the time for adding two vectors. We show all the constant factors for the analysis of computation cost in Table I. (2) The computation time for multiplying $W^{(k)}$ is linear to matrix size $d_k \times d_{k-1}$ and aggregated embedding vector size d_{k-1}, which is $\beta d_k d_{k-1}$. (3) Applying the activation function $\sigma(\cdot)$ is linear to embedding vector size d_k, which takes γd_k time. The total computation cost to compute embedding h_v^k is as follows.

$$c_f(v, k) = \alpha |N_v| d_{k-1} + \beta d_k d_{k-1} + \gamma d_k \tag{6}$$

2) Computation Cost in Backward Step: In the backward step, we compute the gradients of embedding $h^{(k)}$ and the parameter $W^{(k)}$. For the example in Figure 3, each node in backward propagation of the computational graph computes its gradient $\frac{\partial \mathcal{L}}{\partial h_v^{(k)}}$. Based on the chain rule, the gradients are propagated to the neighbors in the next layer. The gradients are computed using Equation (4) as $\sum_{u \in N_v} \left(\frac{\partial \mathcal{L}}{\partial h_u^{(k+1)}} \times \frac{\partial h_u^{(k+1)}}{\partial h_v^{(k)}} \right)$, where we aggregate all the multiplication of gradients $\frac{\partial \mathcal{L}}{\partial h_u^{(k+1)}} \times \frac{\partial h_u^{(k+1)}}{\partial h_v^{(k)}}$ from previous layer. However, the multiplication of gradients should be done in layer $k+1$ instead of layer k for efficiency. Because for any neighbor v of node u, the gradient $\frac{\partial h_u^{(k+1)}}{\partial h_v^{(k)}}$ is the same for embedding $h_u^{(k+1)}$. So we can compute the multiplication of gradients once and reuse the results for all the neighbors of u in Layer k.

In Layer K, we represent the computation $g(v, K)$ in node v as the multiplication between the gradient of output $z_v = h_v^{(K)}$ and the embedding gradient as follows.

$$g(v, K) = \frac{\partial \mathcal{L}}{\partial z_v} \times \frac{\partial z_v}{\partial h_u^{(K-1)}} \tag{7}$$

where u is one neighbor of v in layer $K-1$. The computation time for $g(v, K)$ contains three parts: (1) Computing the gradient towards loss function $\frac{\partial \mathcal{L}}{\partial z_v}$ takes λd_K time. (2) Computing the embedding gradient $\frac{\partial z_v}{\partial h_u^{(K-1)}}$ is the gradient of the activations function $\sigma'(\cdot)$ multiplying the parameter matrix $W^{(K)}$. So it is linear to the matrix size $d_K \times d_{K-1}$ which takes $\eta d_K d_{K-1}$ time. (3) The multiplication between the two result gradients takes $\beta d_K d_{K-1}$ time. Therefore, the total computation cost to compute $g(v, K)$ is as follows.

$$c_g(v, K) = \lambda d_K + (\eta + \beta) d_K d_{K-1} \tag{8}$$

From Layer 1 to Layer $K-1$, the gradient of node v is calculated as $\frac{\partial \mathcal{L}}{\partial h_v^{(k)}} = \sum_{u \in N_v} g(u, k+1)$. So the computation of $g(v, k)$ in node v is computed as follows.

$$g(v, k) = \left(\sum_{u \in N_v} g(u, k+1) \right) \times \frac{\partial h_v^{(k)}}{\partial h_u^{(k-1)}} \tag{9}$$

The computation includes three parts: (1) the aggregation from neighbor nodes $g(u, k+1)$ takes $\alpha |N_v| d_k$ time, (2) computing the embedding gradient $\frac{\partial h_v^{(k)}}{\partial h_u^{(k-1)}}$ is the same as Layer K which takes $\eta d_k d_{k-1}$ time, and (3) the multiplication between the two result gradients also takes $\beta d_k d_{k-1}$ time. Therefore, the total computation cost to compute $g(v, k)$ is as follows.

$$c_g(v, k) = \alpha |N_v| d_k + (\eta + \beta) d_k d_{k-1} \tag{10}$$

Since the goal of computing gradients is to update the parameter $W^{(k)}$, we only compute the gradients of embeddings from Layer 1 to Layer K. So $c_g(v, 0) = 0$.

For computing the gradients of $W^{(k)}$ in Equation (5), we need to compute $\frac{\partial \mathcal{L}}{\partial h_v^{(k)}} \times \frac{\partial h_v^{(k)}}{\partial W^{(k)}}$ for each node v. We amortize the computation cost to each node in the backward propagation of the computational graph. (1) Computing $\frac{\partial h_v^{(k)}}{\partial W^{(k)}}$

is multiplying a scalar gradient of the activation function $\sigma'(\cdot)$ with the aggregated embedding which takes ηd_k time. (2) The product between a $d_k \times 1$ vector and a $1 \times d_{k-1}$ vector takes $\beta d_k d_{k-1}$ time. So each node v from Layer 1 to Layer K requires the additional computation cost as follow.

$$c_W(v,k) = \eta d_k + \beta d_k d_{k-1} \tag{11}$$

Summing up all the required computation cost c_g and c_W based on layers, we have the computation cost for any node v in backward propagation of the computational graph as follows.

$$c_b(v,k) = \begin{cases} (\lambda+\eta)d_k + (2\beta+\eta)d_k d_{k-1} & k = K \\ \alpha|N_v|d_k + (\beta+\eta)d_k d_{k-1} + \eta d_k & 0 < k < K \end{cases} \tag{12}$$

3) Computation Cost of Mini-batches: The computation cost for a mini-batch B is computed by summing up the computation cost from all the nodes in the $\mathcal{G}_B$ from Layer 1 to Layer K.

$$C(B) = \sum_{k=0}^{K-1} \sum_{v \in \bigcup_{u \in B} N_u^k} (c_f(v,k) + c_b(v,k)) \tag{13}$$

where N_u^k represents the nodes in graph G that are k-hop away from node u. For the 0-hop neighbor, we have $N_u^0 = u$. The computation cost function $C(B)$ in CM-GCN has a submodular property. So we can find polynomial-time algorithms to partition mini-batches using submodular function minimization.

Theorem 1. *Given a mini-batch B for a K-layer GCN, the computation cost $C(B) = \sum_{k=0}^{K-1} \sum_{v \in \bigcup_{u \in B} N_u^k} (c_f(v,k) + c_b(v,k))$ is a submodular function.*

Due to the page limit, the proof of Theorem 1 is shown in the document[1].

Give a graph G, we aim to partition all the nodes V into cohesive mini-batch partitions P to minimize the total computation required for each epoch. Since mini-batches are independent, the objective function is computed as the summation of the computation cost of each mini-batch $B \in P$.

$$J(P) = \sum_{B \in P} C(B) \tag{14}$$

4) Partition Nodes into Mini-batches: Using the submodular property of computation cost function $C(B)$, we can partition the nodes into the optimal two mini-batches in polynomial time [17], [21]. However, finding the optimal M mini-batches, where $M > 2$, is none-trivial. Inspired by the multi-way graph partitioning algorithms in [8], [32], we can format our mini-batch partitioning as a multi-way graph partitioning problem. We can achieve a $2 - \frac{2}{M}$ approximation for M partitions through recursive bisection [17]. That is, we iteratively split one of the existing mini-batches into 2 mini-batches to minimize $J(P)$. So each iteration we increase the number of partitions by one until we get M partitions. Using this way, the M partitions P may not reach the optimal $J(P)$. But we can bound the quality of the partition which will not exceed the $2 - \frac{2}{M}$ of the optimal $J(P)$.

We show our algorithm to partition nodes into cohesive mini-batches in Algorithm 1. Initially, the whole graph G forms one mini-batch as $P = \{V\}$. Then we split P into two mini-batches that have the minimum $J(P)$ from any possible two mini-batches. To efficiently find the optimal two-way partitions, we use the idea in Queyranne's algorithm [21]. The key observation in the algorithm is that we can identify a special ordered node pair (t,u) for an arbitrary subset $U \subset V$. The identification of (t,u) reduces the search space because we can group t and u together as a new inseparable node and get a new candidate as $P = \{\{u\}, V \setminus \{u\}\}$. We will explain the details of how to generate pendent pair and find the candidate in the next paragraph. After examining all the possible candidates of two-way partitions, the one that gives the minimum $J(P)$ is set as the new mini-batch partition. We iteratively repeat this process until we obtain M mini-batches.

Algorithm 1: Partition nodes into cohesive mini-batches

Input : A graph $G = (V, E)$, number of mini-batch M
Output: A list of mini-batch partition P

1 Initialize one single partition as $P = V$
2 **for** $i = 1, ..., M-1$ **do**
3 **for** *each mini-batch* $B_j \in P$ **do**
4 Partition B_j into two mini-batches with minimum total computation cost
5 Group the two new mini-batches with other mini-batches to form a partition P_j
6 Compute the total cost $J(P_j)$
7 **end for**
8 $P = \arg\min_{j=1}^{i} J(P_j)$
9 **end for**
10 Return mini-batch partition P

For splitting an existing mini-batch B_j in line 4 of Algorithm 1, we find the partition that minimizes the symmetric computation cost function as follows.

$$U = \arg\min_{U \subset B_j} (C(U) + C(B_j \setminus U)) \tag{15}$$

Then we replace B_j in P with U and $B_j \setminus U$. Since the computation cost function $C(B_j)$ has the submodular property, we can restrict the search space of all the possible two-way partitions to $|B_j|$ candidates. The key idea is to generate a special ordered series of nodes from a given subset $U \subset B_j$. The last two nodes in that ordered series form a pair (t,u) that is called *pendent pair*. The $C(\{u\})$ takes the minimum u from all subsets of B_j which separate node t from u. Using the pendent pair, we can determine one candidate partition as $\{u\}$ and $B_j \setminus \{u\}$. Meantime, we can reduce the search space by grouping two nodes t and u together as a new inseparable

[1]https://drive.google.com/file/d/1LusLZHLon3lUtMYSLIUUNRsrJg6psmPg

node u'. So the remaining node set $S' = B_j \setminus \{u, v\} \cup \{u'\}$ for generating new candidates.

Algorithm 2: Partition one mini-batch into two

Input : A graph $G = (V, E)$, one mini-batch B
Output: Two mini-batches P

Initialize one partition as $B_1 = B$
for $i = 1, ..., |B|$ **do**
  Randomly select a node v_1 in B_i as a initial node.
  Group $W_1 = \{v_1\}$
  for $j = 2, ..., |B|$ **do**
    Find node v_j that minimize $C(W_{j-1} \cup \{v_j\}) - C(\{v_j\})$
    $W_j = W_{j-1} \cup \{v_j\}$
  end for
  Denote the last two nodes added to W as t and u
  Compute the computation cost $J(P_i)$ for candidate partition $P_i = \{B_i \setminus \{u\}, \{u\}\}$
  Group t and u as one node (u, t)
  $B_{i+1} = B_i \setminus \{u, t\} \cup \{(u, t)\}$
end for
$P = \arg\min_{i=1}^{|B|} J(P_i)$
Return mini-batch partition P

The total running time for partitioning nodes into mini-batches takes M iterations of finding the best split from current mini-batches. Each split takes at most M two-way partitions for each mini-batch partition. Since the total nodes in all the mini-batch are always V, the running time for each split can be bounded in $O(|V|^3)$ time. Then, the total running time to partition nodes into mini-batches is in $O(M|V|^3)$ time.

B. Balance Workers' Loads

During the GCN training, the embeddings and gradients are computed from Layer 0 to Layer K following the directions in the computational graph $\mathcal{G}_B$. When we need to synchronize the embeddings or gradients between consecutive GCN layers, the workers that require more computation will delay the computation on other workers in the next layer. However, balancing the workloads in each GCN layer will be very time-consuming and not necessary. In CM-GCN, we propose asynchronous computation between GCN layers, and only synchronize the GCN parameters W between mini-batches. As long as the computation in one mini-batch is balanced among workers, we can achieve no waiting in training. In this subsection, we will focus on balancing the workloads and discuss the asynchronous computation in the next subsection.

The workload can be measured by the running time to process the nodes in the computational graph. In CM-GCN, we use our computation cost function to estimate the workloads on workers. As long as every worker has the same total computation cost, we balance the workloads. The computation of gradient for a node v at layer k requires the embeddings h_v^k. So both the forward and backward steps of node v should be processed at the same worker. Meanwhile, a graph node v usually exists in multiple layers in the computational graph. For example, node A need to be processed in every GCN layer in Figure 3. We put the processing of these nodes in the same worker so that there is no duplicate copy of nodes in different workers. Then, each node in the graph is categorized as a whole computation as follows.

$$nc(v) = \sum_{k=1}^{K} c_f(v, k) + c_b(v, k)$$

Given a mini-batch B, we assign every node v in G that is shown in the computational graph $\mathcal{G}_B$ to workers one by one since the computation costs $nc(v)$ are independent. Initially, all the P workers have the total computation cost as zero. We follow the order of breadth-first search starting from nodes in B to assign nodes to workers. Each time, we assign a node v to the worker that has the lowest total computation cost so far. So the workload differences among workers will not exceed the computation cost of one node. At last, we generate the assignments as a mapping table from a node ID to a worker.

For each worker p, we construct a list of nodes N_p that are required to be processed in that worker. During the training, we first load the input features of N_p into worker p and subgraph associated to N_p. Although the mini-batches are isolated, the dependent nodes may appear in multiple workers. For efficiency, we load the input features of node v to all the workers that will process v in any layer. When we train a mini-batch B, every worker p loads the mapping table that is generated for worker p.

C. Asynchronous Computation between GCN Layers

Traditionally, it is required to synchronize the embeddings and gradients between two consecutive GCN layers. However, the synchronization between GCN layers may incur considerable overhead to wait for embeddings and gradients from other workers. Since processing a node only requires the embeddings and gradients from its neighbors, it is not necessary to synchronize all the embeddings and gradients between two GCN layers. We can process nodes that have their neighborhood embeddings or gradients available in an asynchronous fashion. CM-GCN prioritizes the processing of nodes that are ready to be processed to enable asynchronous GCN training. Since we always compute the embeddings and gradients from current processing results, we can guarantee the same output as synchronous GCN training.

During the training, we decompose the computation of nodes in the computational graph and process each node asynchronously based on its availability. In Figure 4, we illustrate an example of our asynchronous computations with two workers. Each worker stores one partition of the computational graph based on the balanced workload. For example, the mini-batch in Figure 4 is $\{A, C\}$. Worker 1 processes every node in $\mathcal{G}_{\{A,C\}}$ for graph node A and B. A_f^1 indicates the computation of embedding $h_A^{(1)}$ for node A at Layer 1. A_b^1 indicates the computation of gradient for $h_A^{(1)}$ in the backward propagation. Worker 2 focuses on graph nodes C and D.

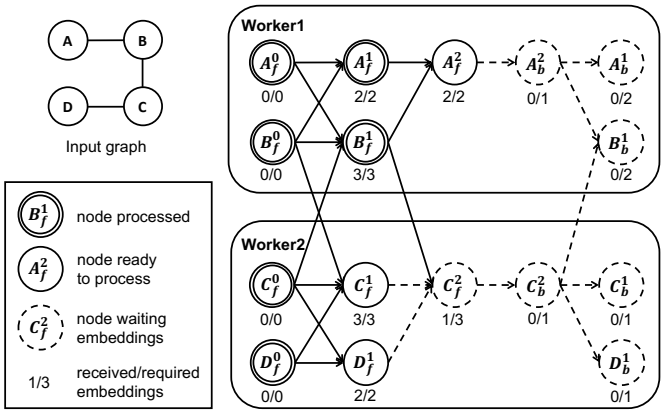


Fig. 4: Asynchronous computations in CM-GCN

The availability represents whether the dependent embeddings or gradients are ready for each node. We maintain a counter for each node in the computational graph to indicate the received embeddings or gradients. Initially, the counter of a node equals 0. In Figure 4, we illustrate the counter under each node. The first number is the current value of the counter while the second number is the number of its neighbors from the previous layer in $\mathcal{G}_B$. Since the nodes in layer 0 using the input features as embeddings, they are always available. So the counter for node v_f^0 is $0/0$.

In CM-GCN, we maintain a *processing queue* in each worker for the nodes that are ready to be processed. Each worker keeps processing the nodes in its processing queue while updating the counter for each node that is not processed. So workers are able to process the nodes without synchronizations between GCN layers. When the counter of a node v reaches the required counter value, we enqueue v into the processing queue. For example, after processing node A_f^1 and B_f^1, node A_f^2 will be ready to be processed in worker 1. There is no need to wait for node C_f^1 and D_f^1 in worker 2. Meanwhile, the embeddings of node A_f^1 and B_f^1 are available for node A_f^2 and the embedding of B_f^1 will be sent to node C_f^2 through message passing. The counter of node A_f^2 will be updated to $2/2$ and added to the processing queue of worker 1. When worker 2 receives the embedding, it will update the counter of node C_f^2 to $1/3$. There are still two more embeddings that are required for node C_f^2 to be ready.

IV. Experiments

In this section, we evaluate our CM-GCN model to show its performance with large-scale datasets. We focus on the node classification tasks in the evaluation. The GCNs for other tasks such as link prediction mostly differ in the objective function while sharing most of the GCN architectures, so we omit them in this paper. Comparing with the well-known GCN models, our CM-GCN using cohesive mini-batches reduces the required computation as well as the total training time. We also examine the effectiveness of using balanced workloads and asynchronous computation in reducing the waiting time. Finally, the scalability test shows that CM-GCN can scale to large clusters.

A. Experiment Setting

We first describe our experiment settings including the datasets, benchmark model, and testing environment. We evaluate our CM-GCN model for training GCN on multi-class classification on four public datasets. The statistic of the datasets from Cora, Reddit, and Open Graph Benchmark (OGB) datasets [12] are shown in Table II. The dimension of the hidden layers for Cora is 256, while we set the dimension as 128 for Reddit and 64 for other OGB datasets. We split the train, validation, and test data ratio as 70%, 20% and 10%.

TABLE II: Dataset Summary

Dataset	# Nodes	# Edges	Node features
Cora	2,708	5,429	1,433
Reddit	232,965	11,606,919	602
OGBN-arxiv	169,343	1,166,243	128
OGBN-product	2,449,029	61,859,140	100

We use the state-of-the-art GCN as the benchmark model. We use mean pooling architecture and ReLU activation function per graph convolution layer. We set weight decay as zero, dropout rate as 0.1. We randomly select 1% of the nodes in the graph for each mini-batch, which is equivalent to 100 mini-batches in CM-GCN. Since Cora data is small, we use 5% as the mini-batches size and 4 workers for training.

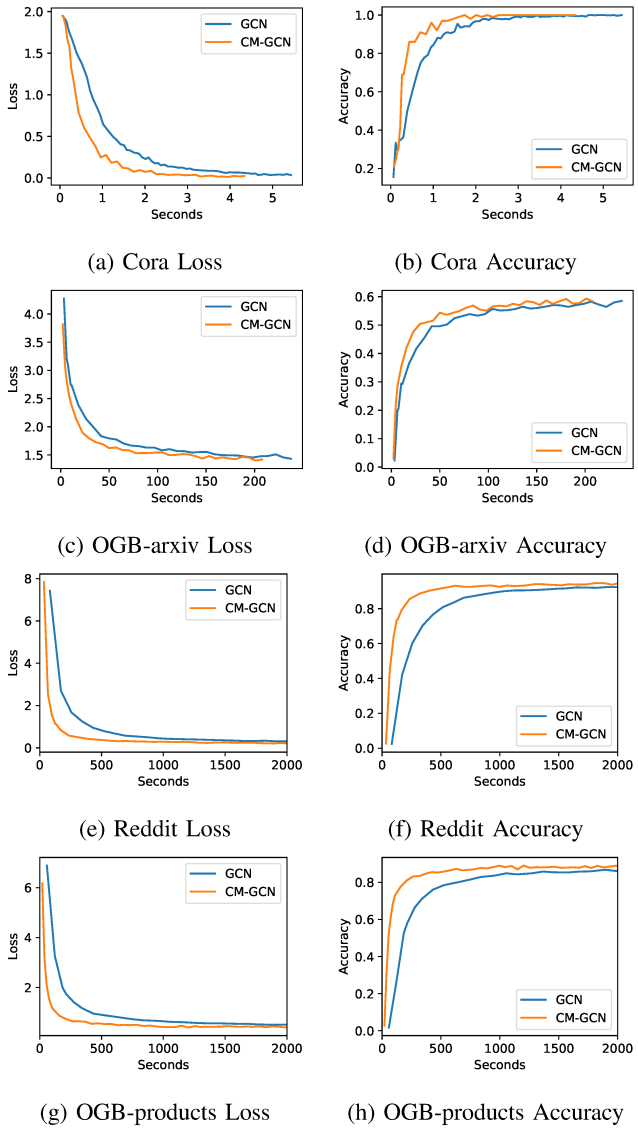
We implement our CM-GCN under DGL v0.6, Pytorch 1.8, and using the existing code of GCN in Dist-DGL [33]. We use a Google cloud cluster of eight n2-highcpu8 with 8 CPU cores at 2.8-GHz and 32GB memory. We evaluate all our experiments using 16 workers on 8 instances and scale to 64 workers in the scalability test. We learn the constant factors in the computation cost function through testing the running time in Pytorch. We measure the factors as $\alpha = 2 \times 10^{-9}$, $\gamma = 2 \times 10^{-9}$, $\eta = 2 \times 10^{-9}$, and $\lambda = 1 \times 10^{-9}$ for MSE function. The factor β in the multiplication between a matrix and a vector is not always constant. When the size of the matrix is large, especially in the first a few GCN layers, β is more stable as $\beta = 8 \times 10^{-9}$. When the size of the matrix is small, such as 32×7 in the last layer, β grows larger. So we set it as $\beta = 5.2 \times 10^{-8}$.

B. Training Time and Accuracy

In this subsection, we show the performance of CM-GCN comparing with GCN in terms of training speed and accuracy. The training time is to run 50 epochs for the Reddit and OGB datasets, and 200 epochs for the Cora dataset. We show the training loss and accuracy towards the training time in Figure 5. The x-axis shows the training time in seconds, and the y-axis shows the training loss and accuracy (F1 score) on the test datasets.

For the Cora and OGB-arxiv datasets, they are all citation networks that have relatively low nodes degrees. The average node degrees are 2 and 6.9. So the nodes in each mini-batch are more closely connected and share more common neighbors.

As shown in Figure 5a and Figure 5c, the Cora dataset reduces more computation which leads to faster convergence using CM-GCN comparing with the OGB-arxiv dataset. We can reach almost the same training accuracy for both datasets as shown in Figure 5b and Figure 5d. In general, we can reduce 20%-40% of the training time using CM-GCN.



(a) Cora Loss (b) Cora Accuracy

(c) OGB-arxiv Loss (d) OGB-arxiv Accuracy

(e) Reddit Loss (f) Reddit Accuracy

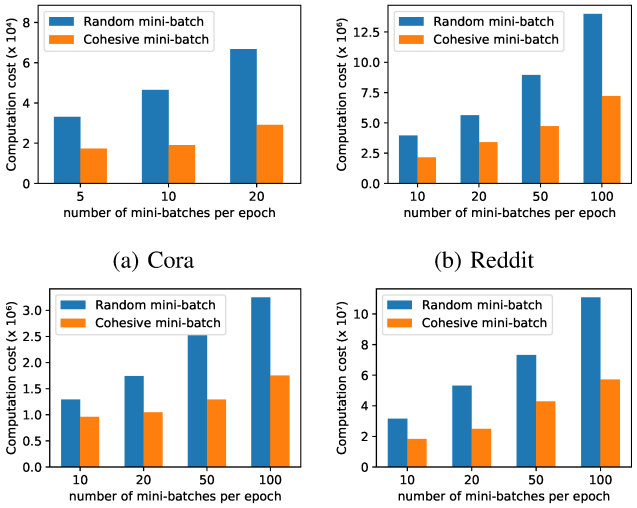
(g) OGB-products Loss (h) OGB-products Accuracy

Fig. 5: Comparison of running time and accuracy for CM-GCN and GCN.

The Reddit dataset has the largest node degree as 50. However, the graph follows a power-law distribution. So the high degree nodes are surrounded by the low degree nodes in each mini-batch. As shown in Figure 5e and Figure 5f, we can reach more than 3X speedup comparing with GCN. The OGB-product dataset has a similar phenomenon. The node degree is around 25, which is smaller than the Reddit dataset. But we can still achieve around 2X speedup which is shown in Figure 5g and Figure 5h.

C. Effect of Cohesive Mini-batches

The improvement from cohesive mini-batches in CM-GCN comes from the reduction of computation per epoch. It is reflected both in the number of nodes we reduced in the computational graph and the total computation costs. In Figure 6, we show the computation reduction from cohesive mini-batches towards random mini-batches. For random mini-batches, we randomly split the graph nodes into M mini-batches. We sum up the computation cost for all the M mini-batches to show the computation reduction. We vary the number M of the mini-batches to show the effectiveness of our cohesive mini-batches.



(a) Cora (b) Reddit

(c) OGB-arxiv (d) OGB-products

Fig. 6: Comparison of computation cost for random mini-batches and cohesive mini-batches.

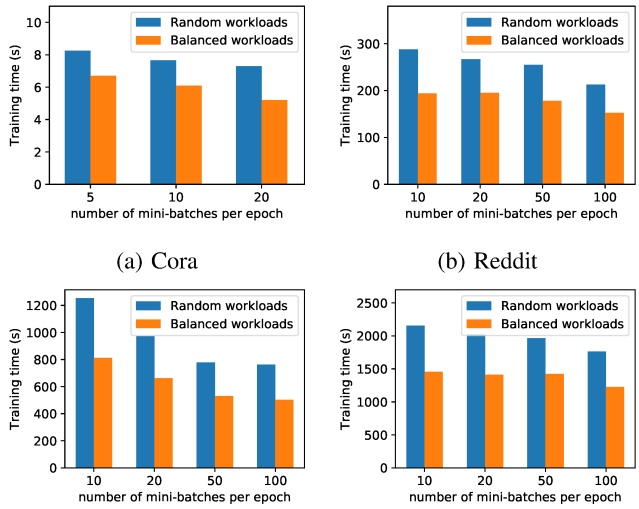
In the Cora dataset, the cohesive mini-batches can save up to 56% of the computation cost towards random mini-batches. The improvement comes from reusing embeddings and gradients in each mini-batch to save the computation. Therefore, the cohesive mini-batches themselves can save around half of the training time towards traditional GCN. When the number of mini-batches is large, we can update the parameter more often, but it required more computation due to the node expansion. Using cohesive mini-batches, we only increase 68% of the computation cost using 20 mini-batches comparing to 5 mini-batches. For other datasets, we observe 46% reduction in OBG-arxiv, 48% reduction in both Reddit and OGB-products.

D. Effect of Balanced Workloads

After balancing the workloads among workers, we can further save the waiting time between mini-batches. We look into the computation time and waiting time for using balanced workloads comparing with random workloads. For random workloads, we evenly assign all nodes to workers based on their nodes ID. So each worker will have the same amount of

nodes to process but there is no guarantee of the computation time. Since the waiting time is hard to directly measure, we compare the total training time. We also vary the number M of the mini-batches to show the effectiveness of balanced workloads.

In Figure 7, we show the improvements from the balanced workloads with different mini-batch sizes. Using our balanced workloads, we can achieve almost no idle time in training. So the computation time is around the same for both balanced workloads and random workloads. The time differences between balanced workloads and random workloads are the time we reduce. When the size of mini-batches is large, we improve the total training time by using more up-to-date parameters. However, more mini-batches leads to fewer nodes to train in each mini-batch. So the workloads will be more unbalanced, which incurs a longer waiting time. Using the balanced workloads can save around 21% of the waiting time for the Cora dataset. We can save similar waiting time for the other datasets as 26% for Reddit, 29 % for OGB-arxiv, and 28% for OGB-products.



(a) Cora (b) Reddit

(c) OGB-arxiv (d) OGB-products

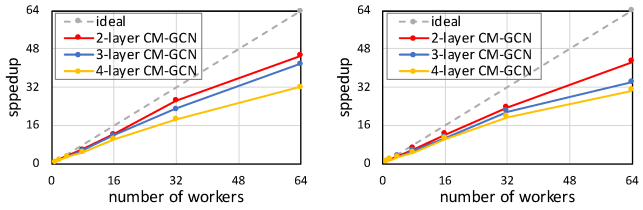
Fig. 7: Comparison of computation time using random workloads and balanced workloads.

E. Training Deeper GCNs

We show the training time and accuracy when we train deeper GCNs in Table III. We train each model 4 times and measure the average training time and accuracy. Comparing to GCN, our CM-GCN can achieve similar or even better accuracy while speedup the training at least 2 times faster. Because CM-GCN carefully generates the cohesive mini-batches and performs asynchronous computation between GCN layers to reduce the possible waiting among workers. Due to the exponential growth of the neighborhood nodes, it is extremely long to train GCN on large datasets. Especially for the dataset with large node degrees like Reddit and OGB-products. Although there is only 1% of the nodes that contribute to the loss computation in the last layer, a 4-layer GCN almost needs to process all the nodes in the first GCN layer for every mini-batch. And the computation in the first GCN layer is usually the most expensive because the dimension of the input feature is the largest. In CM-GCN, we can save more than half of the computation in the first GCN layer by using cohesive mini-batches. Generally, we can save one magnitude of nodes that are required in the computation especially for a graph with a large average node degree like Reddit and OGB-products datasets.

F. Scalability

We further evaluate our CM-GCN on the large-scale clusters to test its scalability. The results are shown in Figure 8 for CM-GCN with different layers. We use two OGB large datasets to show the speedup that is up to 64 workers. For both datasets, we get nearly linear speedup comparing to a single machine version. With the asynchronous computation, the workers process the computation almost with no waiting, so we can achieve around 45X to 50X speedup when we use 64 workers.



(a) OGB-arxiv (b) OGB-products

Fig. 8: Scalability of CM-GCN

V. Related Work

The convolution operation is first introduced to the graph neural network model in [2]. Further, [18], [7] has been proposed to speed up graph convolution computation with localized filters based on Chebyshev expansion. They target relatively small datasets and thus the training proceeds in full batch. In order to scale GCNs to large graphs, sampling techniques such as GraphSAGE [9], FastGCN [3], and VR-GCN [4] have been proposed for efficient mini-batch training.

The sampling algorithms greatly reduce the size of the computational graph in GCN training. GraphSAGE [9] performs uniform node sampling on the neighbors in the previous layer. It enforces a pre-defined budget on the sample size, so as to bound the mini-batch computation complexity. FastGCN [3] enhances the layer sampler by introducing an importance score to each neighbor. The algorithm presumably leads to less information loss due to weighted aggregation. VR-GCN [4] further restricts neighborhood size by requiring only two support nodes in the previous layer. The idea is to use the historical activations in the previous layer to avoid redundant re-evaluation. However, the mini-batches potentially become too sparse to achieve high accuracy. Huang et al. [13] improve FastGCN by an additional sampling neural network. It ensures

TABLE III: Training time and accuracy (F1 scores) for GCN and CM-GCN

		2-Layer		3-Layer		4-Layer	
		Time(s)	Accuracy	Time(s)	Accuracy	Time(s)	Accuracy
Cora	GCN	2.53	80.7 ± 0.88	8.75	81.3 ± 1.19	25.63	82.1 ± 0.61
	CM-GCN	1.85	81.5 ± 0.37	4.21	82.6 ± 0.28	11.34	81.7 ± 0.67
Reddit	GCN	517.8	92.1 ± 0.27	2531.7	91.3 ± 0.42	9632.0	91.9 ± 0.77
	CM-GCN	133.5	91.4 ± 0.6	522.5	91.4 ± 0.51	2809.3	91.8 ± 0.37
OGB-arxiv	GCN	55.2	59.4 ± 0.63	226.8	61.2 ± 0.29	1429.2	60.6 ± 0.36
	CM-GCN	21.9	59.5 ± 0.36	152.3	60.9 ± 0.71	561.2	61.1 ± 0.45
OGB-products	GCN	779.5	85.1 ± 0.26	3895.8	85.7 ± 0.32	23932.4	85.9 ± 0.61
	CM-GCN	268.6	85.5 ± 0.11	1353.5	85.8 ± 0.21	8361.0	86.1 ± 0.55

high accuracy since sampling is conditioned on the selected nodes in the next layer. It may incur significant overhead due to the expensive sampling algorithm and the extra sampler parameters to be learned.

Instead of sampling neighbors, some researchers focus on training from a sub-graph instead of the full graph. ClusterGCN [5] proposes graph clustering based mini-batch training. During pre-processing, the training graph is partitioned into densely connected clusters. During training, clusters are randomly selected to form mini-batches, and intra-cluster edge connections remain unchanged. GraphSAINT [30] samples the training graph first and then builds a full GCN on the subgraph. They analyzed the bias and variance of the mini-batches defined on the subgraphs and proposed normalization techniques and sampling algorithms to improve training quality. Simplified GCN (SGCN) [27], PPRGo [1], and LightGCN [11] explore a linear GCN model that integrates self-connection into graph convolution. The removal of nonlinearities and collapsing the weight matrices to one weight matrix, but generally, they will lose the representation power of GCN models. In CM-GCN, we further optimize the mini-batches by partitioning nodes with our computation cost function. So we can find the cohesive mini-batches that minimize the computation requirements per epoch.

To explore the mini-batch selection, HAG [15] presents the concept of Hierarchically Aggregated computation Graph to aggregate operations that are repeated when nodes share similar neighborhoods. Joseph [16] develops a mini-batch selection strategy based on submodular function maximization to capture the informativeness of each sample and the diversity of the whole subset. They aimed to select the most relevant samples, but the computation required is not reduced. In CM-GCN, we focused on the computation cost function to optimize the required computation to perform full-graph GCN training.

GPUs have been mainly used to train GCNs due to their ability to provide highly parallel computations. NeuGraph [19] and ROC [14] coordinate multiple GPUs to improve the scalability. However, GPUs still have limited memory which will make the scalability much more expensive since the real-world graphs are routinely billion-edge scale [28], [22]. Dorylus [25] used the distributed CPU servers and serverless threads to tackle the scalability issue of using CPUs. They split the graph operations and the tensor workloads while using the bounded asynchronous model to reduce the waiting of dependency in the training. Although they guarantee the convergence of the training, it may waste the computation by using the stale value and slow down the convergence.

To train GCN in a distributed fashion, frameworks such as DistDGL [33], NeuGraph [19], DistGNN [20] and Dorylus [25] are proposed to support parallel training. NeuGraph [19] combined a dataflow abstraction with the vertex-program abstraction to support multi-GPU training. It performed full graph training on multiple GPUs and distributed memory whose aggregated memory fits the graph data. However, training a GCN model in a large graph will become inefficient because one model update requires a significant amount of computation. In [23], they develop a fully-distributed algorithmic framework for training GCNs. DistDGL [33] first partitioned the graph and stored them in different workers. They balanced the graph partitions to achieve both network communication reduction and load balancing. However, although the graph is balanced partitioned on workers, the random mini-batches may still make the computation in each worker unbalanced. We propose an asynchronous computation that prioritizes the processing of nodes that are ready to be processed. So we can parallelize the communication and computation with balanced workloads to achieve no waiting among workers.

VI. Conclusion

We propose CM-GCN, a novel distributed GCN framework that exploits an efficient mini-batch training. Based on the graph structure we group the nodes that are closely connected into cohesive mini-batches. Therefore, CM-GCN can process the same amount of nodes without any neighborhood sampling for only half of the computation cost. After proposing a computation cost function with submodular property, we develop an efficient algorithm to partition nodes into cohesive mini-batches. To further reduce the waiting time caused by synchronizations, we distribute the computation on workers through balanced workloads. We design asynchronous computations between GCN layers to further eliminate the waiting among workers. We implement a distributed CM-GCN framework and evaluate its performance with graphs that contain millions of nodes. Our evaluation shows that CM-GCN can achieve up to 3X speedup without compromising the training accuracy.

Acknowledgment

This work was supported in part by National Science Foundation Grants CNS-1815412 and CNS-1908536.

REFERENCES

[1] A. Bojchevski, J. Klicpera, B. Perozzi, A. Kapoor, M. Blais, B. Rózemberczki, M. Lukasik, and S. Günnemann. Scaling graph neural networks with approximate pagerank. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2464–2473, 2020.

[2] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and locally connected networks on graphs. *arXiv:1312.6203*, 2013.

[3] J. Chen, T. Ma, and C. Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv:1801.10247*, 2018.

[4] J. Chen, J. Zhu, and L. Song. Stochastic training of graph convolutional networks with variance reduction. *arXiv:1710.10568*, 2017.

[5] W.-L. Chiang, X. Liu, and etc. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD*, pages 257–266, 2019.

[6] P. Cui, X. Wang, J. Pei, and W. Zhu. A survey on network embedding. *IEEE Transactions on Knowledge and Data Engineering*, 31(5):833–852, 2018.

[7] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *arXiv:1606.09375*, 2016.

[8] S. Fujishige. *Submodular functions and optimization*. Elsevier, 2005.

[9] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. *arXiv:1706.02216*, 2017.

[10] W. L. Hamilton, R. Ying, and J. Leskovec. Representation learning on graphs: Methods and applications. *arXiv:1709.05584*, 2017.

[11] X. He, K. Deng, X. Wang, Y. Li, Y. Zhang, and M. Wang. Lightgcn: Simplifying and powering graph convolution network for recommendation. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, pages 639–648, 2020.

[12] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv:2005.00687*, 2020.

[13] W. Huang, T. Zhang, Y. Rong, and J. Huang. Adaptive sampling towards fast graph representation learning. *arXiv:1809.05343*, 2018.

[14] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems*, 2:187–198, 2020.

[15] Z. Jia, S. Lin, R. Ying, J. You, J. Leskovec, and A. Aiken. Redundancy-free computation for graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 997–1005, 2020.

[16] K. Joseph, K. Singh, V. N. Balasubramanian, et al. Submodular batch selection for training deep neural networks. *arXiv preprint arXiv:1906.08771*, 2019.

[17] G. Karypis and V. Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[18] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv:1609.02907*, 2016.

[19] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai. Neugraph: parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference*, pages 443–458, 2019.

[20] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. Kalamkar, N. K. Ahmed, and S. Avancha. Distgnn: Scalable distributed training for large-scale graph neural networks. *arXiv preprint arXiv:2104.06700*, 2021.

[21] M. Queyranne. Minimizing symmetric submodular functions. *Mathematical Programming*, 82(1):3–12, 1998.

[22] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 410–424, 2015.

[23] S. Scardapane, I. Spinelli, and P. Di Lorenzo. Distributed training of graph convolutional networks. *IEEE Transactions on Signal and Information Processing over Networks*, 7:87–100, 2020.

[24] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

[25] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, et al. Dorylus: affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 495–514, 2021.

[26] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *arXiv:1710.10903*, 2017.

[27] F. Wu, A. Souza, T. Zhang, C. Fifty, T. Yu, and K. Weinberger. Simplifying graph convolutional networks. In *International conference on machine learning*, pages 6861–6871. PMLR, 2019.

[28] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD*, pages 974–983, 2018.

[29] S. Yun, M. Jeong, R. Kim, J. Kang, and H. J. Kim. Graph transformer networks. *Advances in Neural Information Processing Systems*, 32:11983–11993, 2019.

[30] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna. Graphsaint: Graph sampling based inductive learning method. *arXiv:1907.04931*, 2019.

[31] G. Zhao, L. Gao, and D. Irwin. Sync-on-the-fly: A parallel framework for gradient descent algorithms on transient resources. In *IEEE International Conference on Big Data*, pages 392–397. IEEE, 2018.

[32] L. Zhao, H. Nagamochi, and T. Ibaraki. Greedy splitting algorithms for approximating multiway partition problems. *Mathematical Programming*, 102(1):167–183, 2005.

[33] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis. Distdgl: Distributed graph neural network training for billion-scale graphs. *arXiv:2010.05337*, 2020.