EC-Graph: A Distributed Graph Neural Network System with Error-Compensated Compression

Zhen Song[1], Yu Gu[1], Jianzhong Qi[2], Zhigang Wang[3], Ge Yu[1]
Northeastern University[1], The University of Melbourne[2], Ocean University of China[3]
songzhen_neu@163.com, {guyu,yuge}@mail.neu.edu.cn, jianzhong.qi@unimelb.edu.au, wangzhigang@ouc.edu.cn

***Abstract*—The high training costs of graph neural networks (GNNs) have limited their applicability on large graphs, e.g., graphs with hundreds of millions of vertices which have become common in the era of big data. A few recent studies propose distributed GNN systems. However, these systems may generate high communication costs due to the extensive message passing among graph vertices stored on different machines. To address such limitations, in the paper, 1) we propose a distributed GNN computation system named EC-Graph for CPU clusters, which drastically reduces the communication costs among the machines by message compression; 2) we design a requesting-end compensation method for the embeddings to mitigate the errors induced by compression in the forward propagation and a Bit-Tuner to adaptively balance the model accuracy and message size; and 3) we propose a responding-end compensation approach for the embedding gradients in the backward propagation. Extensive experiments over large real-world datasets show that EC-Graph outperforms state-of-the-art distributed GNN systems on two CPU clusters of different sizes.**

***Index Terms*—distributed systems, distributed graph neural networks, compression, error compensation**

I. Introduction

Graphs have strong representation power for relationships among entities in the real world. They have been used to represent web page networks, molecular structures, social networks, etc. Many data analysis algorithms for graphs have been proposed, including traditional *iterative graph* (IG) algorithms and more recently *graph neural networks* (GNNs) [1]–[3]. While GNNs have shown high accuracy for predictive tasks (e.g., vertex label or link prediction), they also have expensive computation costs. Meanwhile, graphs in the era of big data have become extremely large. For example, a product co-purchasing network (OGBN-Products) has over 2 million vertices and over 100 million edges, while a citation network (OGBN-Papers100M) has over 100 million vertices and over 3.2 billion edges [1]. Such large graphs bring significant challenges to the computation efficiency of GNNs.

A. Challenges in Distributed Graph Neural Network Training

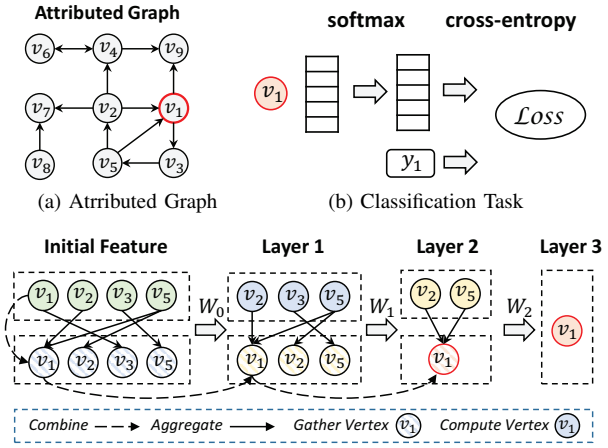
Distributed processing is a natural direction to scale GNNs. To show the challenges in distributed GNN computation (i.e., training), we first brief the general idea of GNN training. As shown in Fig. 1, consider an attributed graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E}, \boldsymbol{X}_{\mathcal{V}}, \boldsymbol{X}_{\mathcal{E}} \rangle$, where $\mathcal{V}$ and $\mathcal{E}$ are the sets of vertices and edges, while $\boldsymbol{X}_{\mathcal{V}}$ and $\boldsymbol{X}_{\mathcal{E}}$ represent the feature (i.e, attribute) sets of the vertices and edges, respectively. A GNN aims to generate an embedding (i.e., a vector) to represent a vertex, an edge, or a graph through aggregating the graph structure and/or feature information. The computed embeddings can be used in downstream tasks, e.g., vertex classification as shown in Fig. 1b. Fig. 1c shows the general processing flow of a 3-layer GNN for a vertex classification task. Consider vertex v_1 from Fig. 1a. Since this is a 3-layer GNN, the vertices that contribute to the embedding of v_1 are the 3-hop neighbors of v_1, i.e., v_2, v_3, and v_5. Each of these vertices v_i fetches the embeddings (feature vectors in the initial layer) of its in-neighbors in each layer. These embeddings, together with the embedding of v_i, go through a GNN computation to produce the updated embedding of v_i which becomes the input of the next GNN layer. After three layers, the final embedding of v_1, $\boldsymbol{h}_{v_1}^{(3,t)}$, is obtained, which is used with the ground truth label y_1 (Fig. 1b) to compute a training loss (e.g., via a fully connected layer). The gradient is computed and propagated backward to update the GNN parameters. This process is repeated for T times or until the GNN parameters converge. Different variants of GNNs have been proposed. They differ in their aggregation functions ($agg()$) to aggregate embeddings of the neighboring vertices and their combination functions ($com()$) to combine the aggregated embedding with the embedding of target vertex v, as summarized by the equation:

$$\boldsymbol{h}_v^{(l,t)} = \sigma(com(\boldsymbol{h}_v^{(l-1,t)}, agg_{u \in \mathcal{N}(v)}(\boldsymbol{h}_u^{(l-1,t)})) \odot \boldsymbol{W}^{(l-1,t)}) \quad (1)$$

Here, $\boldsymbol{W}^{(l-1,t)}$ denotes the GNN parameters (weights) to be learned, and $\sigma()$ denotes an activation function, e.g., ReLU.

Existing distributed IG frameworks [4]–[7] cannot process GNN computations efficiently. This is because each vertex in GNN needs to propagate a high dimensional vector (i.e., its embedding) to its neighbors in each GNN layer, while only a scalar is propagated in traditional IG tasks. More importantly, GNN has the forward and backward propagations in model training which are not inherently supported by IG frameworks. Distributed *machine learning* (ML) frameworks support model training. However, they mostly use Parameter Server (PS) [8]–[14] or All-Reduce [15], [16] frameworks, which assume independent training samples. The samples in GNNs, i.e., the vertices, are not independent. Thus, such distributed ML frameworks are not applicable directly.

[1]https://ogb.stanford.edu/docs/nodeprop/



(a) Atrributed Graph (b) Classification Task

(c) Forward Propagation

Fig. 1: An Example of a 3-Layer GNN

Recently, several studies [17], [18], [20] built distributed GNN systems on top of ML frameworks. These systems locally cache L-hop (L corresponds to the number of layers) neighboring vertices to enforce the independence among physical computing nodes, leading to redundant computations. Besides, it is difficult for these systems to support GNN training on large graphs in the full-batch mode (which is preferred due to a faster convergence), because a huge memory space will be required for caching L-hop neighbors. The problem becomes even worse when processing graphs with small diameters, where L-hop neighbors may cover a large portion of the graphs.

Other emerging GNN systems such as DistGNN [35] follow distributed graph processing procedures to bypass the data independence requirement, which is the path that we take in this paper. One of the biggest challenges in building such systems is that all vertices need to send (receive) high dimensional embeddings to (from) their neighboring vertices in each GNN layer. Many of such neighbors may reside across computation nodes which need to be transferred, generating high communication costs. Consider a 3-layer GNN model (128-dimensional embeddings for each layer) to be trained on a graph with 10 million vertices and 1 billion edges. Assume that each vertex has 10% neighbors residing in other computation nodes, and each dimension is represented by a 4-byte floating point number. The GNN model can produce roughly 50 GB ($10^9 \times 0.1 \times 128 \times 4/1024^3$) embeddings in each layer, and 250 GB (three forward propagations and two backward propagations) messages need to be communicated in each epoch. Such computation and communication costs pose significant challenges to the distributed system design.

B. Our Proposal

To address these challenges, we propose a distributed GNN system named *EC-Graph* for CPU clusters. Further, we focus on designing optimizations to reduce distributed communication costs. Optimizations for standalone GNN training algorithms are orthogonal to our work. EC-Graph follows the framework of distributed iterative graph computation systems and reduces the communication costs by aggressive compression methods. As a result, errors may be resulted from the compression. To mitigate these errors, we propose error compensation algorithms for both vertex embeddings and embedding gradients. For the vertex embeddings, we maintain extra approximate embeddings and adaptively select the embeddings to be sent based on the compression errors. We also tune the number of bits used in compression based on the communication costs allowed in each computation iteration. For the embedding gradients, we maintain the errors generated by compression in the current iteration and correct these in the next iteration. To the best of our knowledge, EC-Graph is the first effort to combine the error-compensated mechanism with graph-structured GNN analysis tasks. Our main contributions are summarized as follows.

- We design a novel error-compensated distributed GNN system named EC-Graph that takes a distributed iterative graph processing approach to avoid the high computation and memory costs. EC-Graph integrates aggressive lossy compression for the messages of embeddings and embedding gradients to reduce communication costs.
- We present an algorithm named *ReqEC-FP* to compensate for the compression errors at the requesting-end. For faster convergence, a Bit-Tuner is designed to adaptively adjust the compression rate.
- We propose another algorithm named *ResEC-BP* to compensate for the errors generated by the last iteration at the responding-end. We further show an error upper bound for the proposed compensation methods.
- Extensive experiments on real datasets show that EC-Graph outperforms state-of-the-art distributed GNN systems consistently. We achieve a performance improvement in terms of DistGNN and DistDGL (state-of-the-art sampling and non-sampling based systems) by $1.10 \sim 1.48\times$ and $1.35 \sim 6.28\times$ on real datasets.

II. Related Work

We review distributed iterative graph and machine learning frameworks, GNN computation systems, model compression and error compensation techniques.

A. Distributed Frameworks for Traditional Iterative Tasks

Traditional distributed large graph iterative computation systems are mostly developed from Pregel [4], which proposes a vertex-centric programming model. PowerGraph [5] presents a new Gather-Apply-Scatter (GAS) computation model on graphs, together with a vertex-cut partition strategy. PowerSwitch [6] proposes a graph parallel mode with an adaptive switching between synchronous and asynchronous to achieve optimal convergence. These systems are designed for graph analysis tasks such as PageRank, shortest-path finding and connected component computation. They are unsuitable for complex GNN computation and efficient high-dimensional vector propagation.

Many iterative frameworks have been designed for ML tasks. For example, Parameter Server [10] and Petuum [11] are the early general-purpose distributed ML frameworks. FlexPS [8] can dynamically adjust the degree of parallelism to adapt to the changing-workload in different stages of training. All-Reduce [15] can better leverage the bandwidth between GPUs. However, this framework is inapplicable to the heterogeneous cluster. Moreover, TUX^2 [23] transforms ML tasks into a bi-partite graph representation, where the edges connect samples and parameters, which processes MLs with a distributed graph system. In summary, these systems for MLs focus on gradient aggregations and model updates, where no communication is needed among samples. As a result, they do not adapt to distributed GNN tasks.

B. Graph Neural Network Systems

Scarselli et al. [37] propose the first GNN model together with a training algorithm. Many studies have followed and presented new training algorithms and model variants. However, performance optimization for GNN systems is still in its infancy. GNN toolkits, such as PyG [24], DGL [25] and CogDL [26], have implemented many GNN variants to simplify the building of customized GNN models. However, these toolkits optimize the performance only for a single machine. Other systems, such as PaGraph [28], ROC [21], PCGCN [29], CAGNET [22] and NeuGraph [27], are GPU-based systems, which mainly focus on cache policies and parallel acceleration on GPUs.

A few distributed systems for GNNs on CPU-cluster have been proposed, which are divided into two categories: Machine Learning Computation centered (*ML-centered*) and Graph Computation centered (*Graph-centered*) frameworks. In ML-centered based GNN systems, each vertex caches L-hop neighbors to the machine that manages the vertex. Each machine thus stores all the required information for GNN computation. Take a vertex v_1 as an example which has an in-neighbor v_2 ($v_1 \leftarrow v_2$ for convenience). There are two more vertices v_3 and v_4 that are relevant, where $v_2 \leftarrow v_3$, and $v_3 \leftarrow v_4$. For a 3-layer GNN (i.e., L = 3), v_1 will cache the feature vectors of v_1, v_2, v_3 and v_4, as well as the adjacency lists of v_1, v_2 and v_3. In comparison, the Graph-centered based GNN systems only cache the feature vector and adjacency list of v_1 on the machine that manages v_1, and the machine of v_1 only needs to communicate with the machine that manages v_2 for its embedding during training. Through a 3-layer aggregation, the information of v_4 can be eventually propagated and aggregated to v_1. Although the ML-centered systems do not require communication among the workers, they need to cache a large amount of data and have many redundant computations on every machine in the cluster.

EC-Graph is a Graph-centered based GNN system that runs on CPU clusters. Existing GNN systems based on CPUs include PSGraph [17], AGL [20], AliGraph [18], DistDGL [19] and DistGNN [35]. The first three systems are ML-centered, while DistDGL and DistGNN are Graph-centered like EC-Graph. DistGNN's main optimization is a delayed remote aggregation to reduce the communication traffic in each iteration, i.e., only a portion of the vertices request for neighboring vertices embedding vectors from another machine in each iteration. As a result, DistGNN has a slower convergence. EC-Graph does not have this issue, since all vertices will communicate for neighbors' embeddings in each iteration.

TABLE I: Notations

Symbol	Explanation
$\boldsymbol{X}_i$	The feature vector of the ith vertex
$\boldsymbol{W}_l$	The neural network parameters of the lth layer
$\|V\|$	The number of vertices
$\|E\|$	The number of edges
$\overline{g}$	The average degree of each vertex
$\overline{d}$	The average dimension size of each vertex
T	The number of iterations
L	The number of layers
$\boldsymbol{h}_v^{(l,t)}$	The representation of v at the lth layer in the tth iteration
d^l	The output dimension size of the lth layer
$\overline{g}_{rmt}$	The average number of remote 1-hop neighbors of each vertex

C. Compression and Error Compensation for MLs

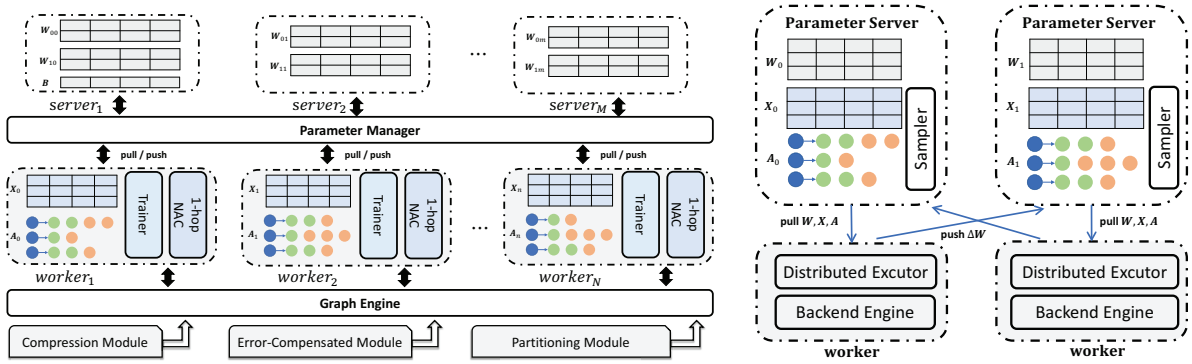
Compression techniques can be categorized into lossless and lossy methods. The latter is usually more efficient at the expense of accuracy, which has been applied to ML tasks. SketchML [30] utilizes sketch to compress gradients and adopts a MinMaxSketch strategy to reduce the compression error. Top-k sparsification [32] keeps the largest k gradient values, while 1-Bit quantization [31] simplifies the gradient values into -1,+1. DoubleSqueeze [33] compresses the gradients transferred from the workers to the servers and those from the servers to the workers, and compensates in the next iteration. Error-Compensation (EC) in ML is designed for communicating compressed gradients $\nabla \boldsymbol{F}$ and parameters $\boldsymbol{W}$ between workers and servers. In comparison, EC-Graph compresses and compensates for the vertex messages (embeddings $\boldsymbol{H}$ in forward propagation and gradients of embeddings $\boldsymbol{G}$ in backward propagation) among workers, which has not been considered in ML since the training samples in ML are considered independent. It is non-trivial to design an effective EC method for distributed GNN systems. To begin with, the compression errors flow between GNN layers following the graph topology, which needs a careful algorithm design to track and compensate for. Further, error compensation for forward and backward propagations needs to be done differently. Neither of these can be addressed by existing ML error compensation methods directly.

III. EC-Graph System

We describe the overall framework and detail an implementation of the *graph convolutional network* (GCN) on EC-Graph. We conclude this section by comparing EC-Graph (a typical Graph-centered system) to ML-centered systems from the memory, computation and communication costs. Table I summarizes our notations.

A. EC-Graph Overview

EC-Graph is a distributed GNN system that takes a graph dataset $\mathcal{G}$ and a GNN (e.g., GCN) as the input, trains (i.e.,



(a) EC-Graph (b) Existing ML-Centered Framework

Fig. 2: System Architectures

optimizes the model parameters) the GNN over $\mathcal{G}$, and outputs the trained network.

EC-Graph modules. Fig. 2a shows the architecture of EC-Graph which follows the general design of distributed iterative graph processing systems. For comparison, we also show the general architecture of existing ML-centered distributed GNN systems, which is detailed in Section III-C.

EC-Graph is based on a CPU cluster with two types of computation nodes: *workers* and *servers*. The workers are responsible for the data-parallel computation to obtain the gradients. They can communicate with each other to fetch embeddings and embedding gradients of the vertices in forward and backward propagations, respectively. The servers maintain and update the GNN parameters to be trained. A physical node can run any number of workers and servers. The workers communicate with the servers by two operators *pull* and *push* for obtaining parameters and sending gradients, respectively. The workers fetch embeddings and embedding gradients by operator *get*.

There are two modules managing the overall computation process: *Graph Engine (GE)* and the *Parameter Manager (PM)*. *GE* manages the vertex communication, partition and storage of graphs, and other optimizations on graphs. *PM* is responsible for the partition and storage of GNN parameters. In fact, both of these two modules are logical layers distributed among various workers.

EC-Graph model training process. Given an input graph $\mathcal{G}$ and a GNN to be trained, *GE* calls the partition module to divide $\mathcal{G}$ into n parts (the number of workers). After partitioning, each worker accesses *NFS* (a general distributed file system) for their local subgraphs, including the subgraph topology and the vertex feature of the subgraphs. Then, *PM* divides GNN parameters onto m servers according to some user-defined partition strategy. By default, we implement a built-in range-based partition method, which divides the weights W and biases B of each layer evenly. The workers then start the model training process by computing both forward and backward propagations. During this process, the workers request *PM* for relevant weights and biases. Then, *PM* locates and pulls the required parameters from servers, and returns the parameters to the requesting workers. The 1-hop NAC (Neighbor Access Controller) component in each worker is in charge of accessing the 1-hop neighbors from the (same or other) workers. Specifically, the local neighboring vertices (residing in the same workers) are obtained from the shared memory, while neighboring vertices on other workers are fetched by the *GE*.

A well-designed partition strategy can help reduce the communication costs, and we have implemented the *Hash* and *METIS* partitioning algorithms in EC-Graph. Some streaming methods [38] can partition graphs with low space and time costs, which will be left in future work. Although *METIS* aims to minimize the across-node communication during iterative computation, it still does not avoid the communication costs triggered by message passing inherent to GNN training. To address this issue, EC-Graph has a *compression module* and a *error-compensated module* to compress and compensate the vertex messages respectively, when *GE* fetches the remote neighboring vertices.

We additionally implement two basic optimizations which are common in the existing GNN systems. First, we cache the first-hop remote neighbors in each worker, leading to less communication without inducing extra costs. Second, we adopt the same message-aggregating optimization with DGL (i.e., if in-dimension $>$ out-dimension, EC-Graph will compute the $\boldsymbol{X} \cdot \boldsymbol{W}$ first, and then perform aggregation $\boldsymbol{A} \cdot \boldsymbol{XW}$).

B. Graph Convolutional Network on EC-Graph

We show how GCN is implemented on EC-Graph for its two key steps, i.e., forward and backward propagations.

Forward Propagation (FP). In each layer of GCN, a vertex v updates its embedding based on its 1-hop neighbors' embeddings from the last layer. Some of the 1-hop neighbors may reside in a different physical node, which are requested by the *get* operator through network communications. These embeddings, together with the embeddings of the 1-hop neighbors stored with v, form a matrix which is used to compute the new embedding of v for the current layer. Eq. 2 and Eq. 3 formalize this process, where $\boldsymbol{A} = \boldsymbol{D}^{-\frac{1}{2}}(\boldsymbol{A}+\boldsymbol{I})\boldsymbol{D}^{-\frac{1}{2}}$ is a

normalized adjacency matrix, $\boldsymbol{D}$ is the degree matrix and $\boldsymbol{I}$ is the identity matrix. Note that the initial embeddings are set as the feature vectors of the vertices, i.e., $\boldsymbol{H}^0 = \mathbf{X}_{\mathcal{V}}$.

$$\boldsymbol{Z}^l = \boldsymbol{A}^T \boldsymbol{H}^{l-1} \boldsymbol{W}^{l-1} \tag{2}$$

$$\boldsymbol{H}^l = \sigma(\boldsymbol{Z}^l) \tag{3}$$

Algorithm 1 shows the process of FP on EC-Graph. We train the model for T iterations. For each iteration, each vertex needs to propagate and aggregate L times, assuming a GCN of L layers. For computing layer l, the workers pull the weights $\boldsymbol{W}^{l-1}$ from the servers (line 4). If l is not the last layer, the vertex embeddings need to be propagated and aggregated for the next layer as follows. Each worker first accesses other workers and its local memory to obtain embeddings of all 1-hop neighbors for the vertices managed by the worker (lines 5 and 6). Then, the workers concatenate the local and remote embeddings into a new matrix $\boldsymbol{H}_{cat}^{l-1}$ (line 7). Finally, we aggregate neighbors for each vertex by a dot-product with the adjacency matrix A^T, and transform the embeddings by another dot-product with the model weight matrix W^{l-1} (lines 9 and 11). If l is the last layer of the current iteration, the embedding of each vertex is fed into the loss function to compute the gradients (lines 12 and 13).

Algorithm 1: Forward Propagation on EC-Graph

Input: Feature Matrix $\boldsymbol{X}_v$, Adjacency Matrix $\boldsymbol{A}$

1 **Worker** $n = 1, 2, ..., N$ in Parallel:
2 **for** t **from** 1 **to** T **do**
3 **for** l **from** 1 **to** L **do**
4 $\boldsymbol{W}^{l-1}$ = pull(l-1)
5 *locNeiEmbs* = getLocEmbs(*locNodeSet*,$l-1$)
6 *rmtNeiEmbs* = getRmtEmbs(*rmtNodeSet*,$l-1$)
7 $\boldsymbol{H}_{cat}^{l-1}$ = concatenate(*locNeiEmbs*, *rmtNeiEmbs*)
8 **if** l != L **then**
9 $\boldsymbol{H}^l = \sigma(\boldsymbol{A}^T \boldsymbol{H}_{cat}^{l-1} \boldsymbol{W}^{l-1})$
10 **else**
11 $\boldsymbol{H}^l = \boldsymbol{A}^T \boldsymbol{H}_{cat}^{l-1} \boldsymbol{W}^{l-1}$
12 $score$ = softmax($\boldsymbol{H}^l$)
13 $\mathcal{L}$ = etropyloss(*score*, *label*)

Backward Propagation (BP). The equations of BP for GCN have been derived in CAGNET [22], which are listed as Eqs. 4 to 6. Here, $\boldsymbol{G}^l$ represents the partial derivative of loss $\mathcal{L}$ over the output embeddings of the lth layer $\boldsymbol{Z}^l$. Through the chain rule, we can obtain the gradients for the weights in the previous layer, i.e., $\boldsymbol{Y}^{l-1}$. Note that $\boldsymbol{G}^{l-1}$ is flowed from $\boldsymbol{G}^l$ following the adjacency matrix $\boldsymbol{A}$.

$$\boldsymbol{G}^L = \frac{\partial \mathcal{L}}{\partial \boldsymbol{Z}^L} = \nabla_{\boldsymbol{H}^L} \mathcal{L} \odot \sigma'(\boldsymbol{Z}^L) \tag{4}$$

$$\boldsymbol{G}^{l-1} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{Z}^{l-1}} = \boldsymbol{A}\boldsymbol{G}^l (\boldsymbol{W}^l)^T \odot \sigma'(\boldsymbol{Z}^{l-1}) \tag{5}$$

$$\boldsymbol{Y}^{l-1} = (\boldsymbol{H}^{l-1})^T \boldsymbol{A}\boldsymbol{G}^l \tag{6}$$

Since the vertices only communicate with their 1-hop neighbors in each layer, the dataflow from 2-hop to L-hop cannot be obtained locally. We compute gradients gradually in BP by sending $\boldsymbol{G}^l$ in each layer, as summarized in Algorithm 2. The gradients of the weights $\boldsymbol{W}^{L-1}$ are the first to be computed (lines 7, 13 and 14), which is done locally on each worker. From layers $L-1$ to 1, each worker needs to request $\boldsymbol{G}^{l+1}$ to compute $\boldsymbol{G}^l$ (lines 9 to 12). After computing $\boldsymbol{G}^l$, the gradients $\boldsymbol{Y}^{l-1}$ can be obtained (lines 13 and 14). Finally, each worker sends the gradients to the servers based on the parameter addressing maps (line 15). The servers receive gradients from each worker, add them up to obtain the global gradients, and update the weights with the global gradients. EC-Graph can also support other GNN models as long as they exchange the same types of information (i.e., embeddings and embedding gradients of neighboring vertices) and have the same communication topologies. For example, *Graph Attention Networks* (GAT) fetches embeddings from in-neighbors in FP and embedding gradients from out-neighbors in BP. Other computations such as embedding aggregation and weight updates are computed locally on each machine, which do not interfere with the processing paradigm of EC-Graph and hence can be integrated into EC-Graph straightforwardly.

Algorithm 2: Backward Propagation on EC-Graph

Input: Weights $\boldsymbol{W}$, Adjacency Matrix $\boldsymbol{A}$, embeddings $\boldsymbol{Z}$

1 **Server** $m = 1, 2, ..., M$ in Parallel:
2 $grads += grad_i$
3 $AdamOptimizer(\boldsymbol{W}, \boldsymbol{grads})$

4 **Worker** $n = 1, 2, ..., N$ in Parallel:
5 **for** l **from** L **to** 1 **do**
6 **if** l==L **then**
7 $\boldsymbol{G}^l = \nabla_{\boldsymbol{H}^l} \mathcal{L} \odot \sigma'(\boldsymbol{Z}^l)$
8 **else**
9 *locNeiG*=getLocG(*locNodeSet*,$l+1$)
10 *rmtNeiG*=getRmtG(*rmtNodeSet*,$l+1$)
11 $\boldsymbol{G}_{cat}^{l+1}$=concatenate(*locNeiG*, *rmtNeiG*)
12 $\boldsymbol{G}^l = \boldsymbol{A}\boldsymbol{G}_{cat}^{l+1} (\boldsymbol{W}^{l+1})^T \odot \sigma'(\boldsymbol{Z}^l)$
13 $\boldsymbol{Y}^{l-1} = (\boldsymbol{H}^{l-1})^T \boldsymbol{A}\boldsymbol{G}^l$
14 $grad$.append($\boldsymbol{Y}^{l-1}$)
15 push($grad$)

C. Architecture Comparison and Analysis

We compare EC-Graph with ML-centered frameworks in Fig. 2b. Systems based on such frameworks (e.g., AliGraph [18]) store the neural network models and all the information of an input graph in the parameter servers. They usually take a mini-batch training mode with a sampling approach, to suit the processing capability of each individual worker. At start, each worker sends out a sampling request to obtain mini-batches consisting of target vertices. It then samples L-hop neighbors of each target vertex assigned to the server, and pulls all the needed information (features and adjacency lists) of the L-hop neighbors. Local training is done on the worker with the pulled information for the target vertices. The workers do not need to communicate with each other for updating the

TABLE II: Algorithm Costs

	ML-centered framework	EC-Graph
Memory Space	$O(\overline{g}^L \cdot \overline{d})$	$O(\overline{g} \cdot \overline{d})$
Computation Cost	$O(\overline{g}^{L-1} \cdot \overline{d}^2)$	$O(L \cdot \overline{d}^2)$
Communication Cost	$O(\overline{g}^L \cdot d_0)$	$O(\frac{T \cdot L \cdot \overline{g}_{rmt} \cdot \overline{d}}{32/B})$

vertex embeddings, at the expense of extra memory space and computation costs for the L-hop neighbors.

Since the neighboring vertices are only sampled, such systems suffer in model training accuracy, and they are difficult to run in the full-batch mode without sampling. In contrast, our system shows favorable scalability, and we run in a full-batch without sampling, which has been shown to converge faster with higher accuracy [21], [22]. Further, EC-Graph also supports a sampling-based training mode.

We further analyze the communication, computation and memory space costs of ML-centered frameworks and EC-Graph, assuming a full-batch mode without sampling. In terms of memory space, ML-centered framework based systems need to cache L-hop neighbors, and hence each vertex generates a neighbor set of size $(1 + \overline{g}^1 + \cdots + \overline{g}^L) = \frac{1-\overline{g}^{L+1}}{1-\overline{g}} \approx \overline{g}^L$, where $\overline{g}^i$ denotes the average degree of the ith layer. The memory space for caching the feature vector of each vertex is $\overline{g}^L \cdot \overline{d}$, where $\overline{d}$ denotes the average dimension size. The cost for caching the adjacency list is $\overline{g}^{L-1} \cdot \overline{g} = \overline{g}^L$ as the Lth-hop neighbors do not generate new embeddings. The total memory space cost is then $O(\overline{g}^L \cdot \overline{d})$. In comparison, EC-Graph only needs $O(\overline{g} \cdot \overline{d})$ space for each target vertex. For simplicity, we do not take the same cost into account, e.g., the weights.

The computation complexity of a matrix dot-product computation of $M_{|V| \times \overline{d}} \odot M_{\overline{d} \times \overline{d}}$ is $O(|V| \cdot \overline{d}^2)$. As a result, the ML-centered framework needs $O(\overline{g}^{L-1} \cdot \overline{d}^2)$ time for each target vertex in each iteration, while EC-Graph takes $O(L \cdot \overline{d}^2)$ time.

For an ML-centered framework, all L-hop information needs to be pulled from the parameter servers, and the communication cost is $O(\overline{g}^L \cdot d_0)$, where d_0 is the dimensionality of the initial feature vectors. This process is performed once in the preparation phase. Our EC-Graph needs a communication cost of $O(T \cdot L \cdot \overline{g}_{rmt} \cdot \overline{d})$, where $\overline{g}_{rmt}$ denotes the average number of remote 1-hop neighbors of each vertex. In the next section, we will present a compression technique to further reduce the communication cost of EC-Graph by a factor of $32/B$, i.e., to $O(\frac{T \cdot L \cdot \overline{g}_{rmt} \cdot \overline{d}}{32/B})$, where B is the number of bits for compressing. We summarize the algorithm costs in Table II.

IV. Error-Compensated Compression for Forward and Backward Propagations

We first outline our compression technique that comes with error-compensation to mitigate its impact on model accuracy. Then, we detail the requesting-end compensated model (*ReqEC-FP*) for FP, together with a Selector and a Bit-Tuner. Finally, we detail the responding-end compensated model (*ResEC-BP*) for BP, and we provide a theoretical error bound.

A. Compression and Error Compensation

EC-Graph requires information exchanging among different physical nodes which incurs communication costs. Consider an L-layer GNN, the intermediate embeddings $\boldsymbol{H}^0$ to $\boldsymbol{H}^{L-1}$ need to be exchanged in FP, while the intermediate gradient results $\boldsymbol{G}^L$ to $\boldsymbol{G}^2$ need to be exchanged in BP. The vertex information exchanged is the main communication bottleneck, because a large number of vector messages may need to be transmitted along the graph topology. To reduce the communication costs, we take a compression approach. Since the vertex information exchanged in FP and BP is in the form of a matrix, we compress each matrix by mapping its elements into N_B buckets. We then send the bucket values and the encoded matrix to the requesting workers.

Fig. 3 shows an example of the compression process for FP. Compression for BP is similar and hence omitted. Consider a worker w_i with two local vertices v_3 and v_4 which needs to fetch neighbor embeddings $\boldsymbol{h}_5$ and $\boldsymbol{h}_6$ from another worker w_j. Worker w_j (i.e., the *responding end*) compresses $\boldsymbol{h}_5$ and $\boldsymbol{h}_6$ before sending them to w_i. In this example, we set the number of bits B for the compressed embedding as 2. This means that we partition the data domain (i.e., $[0, 1]$) for each embedding dimension into $2^B = 4$ buckets. This enables sending the bucket ID (B bits) for each embedding dimension instead of the coordinate (a 32-bit floating point number) in that dimension, reducing the communication cost per dimension from 32 to B=2. Each bucket has its lower and upper bounds (e.g., 0.6 and 1 for bucket 2), the value of which is set to the average value of both bounds (0.8 for bucket 3).

As the figure shows, the compression is first done with a *map* stage which maps an embedding coordinate to the bucket that contains the value, e.g., 0.7 is mapped to bucket 2. Since this example uses 8-dimensional embeddings, we obtain two 16-bit mapped values, which are concatenated into a 32-bit unsigned integer (3,388,157,968). This compressed embedding matrix is sent together with the average values of each bucket ($\{0.2, 0.5, 0.8, 0.0\}$) to worker w_i (i.e., the *requesting end*). Worker w_i then decompresses the embeddings $\boldsymbol{h}_5$ and $\boldsymbol{h}_6$ by reading the bucket ID for each dimension and using the average value of the corresponding bucket as the coordinate of the dimension.

In general, given d-dimensional embeddings where each dimension takes b bits and B bits to represent a bucket ID, this compression technique reduces the message size per embedding from $(d \cdot b)$ to $(d \cdot B + 2^B b)$. Note that the cost $(2^B b)$ to send the bucket averages will be amortized with more embeddings to be compressed and sent.

Our design provides optimization opportunities to balance the communication costs and the model accuracy by tuning the number of bits B.

B. Compensation in Forward Propagation (ReqEC-FP)

We first compensate for the errors in $\boldsymbol{H}$ at the requesting end in FP. We also design an adaptive bit-tuner to make a trade-off between efficiency and accuracy.

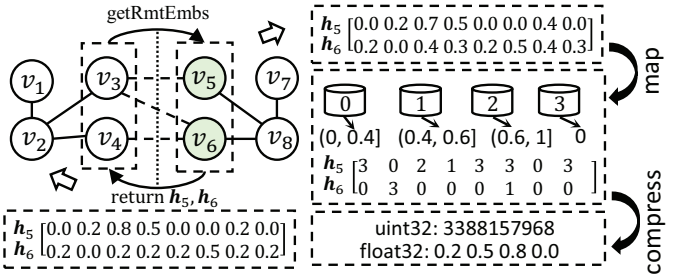


Fig. 3: Compression Example

Selector for Foward Compensation. To compensate for the errors in the embeddings sent from a responding end worker w_j to a requesting end worker w_i, we send not only the compressed embeddings ($C_{bits}(\boldsymbol{H})$), but also the non-compressed embeddings periodically to w_i in every T_{tr} iterations (defined as a trend-group), where T_{tr} is a system parameter. On the responding end w_j, we further record the changing rate for each embedding coordinate in a matrix $\boldsymbol{M}_{cr}^{l,t} = (\boldsymbol{H}^{l,t+T_{tr}} - \boldsymbol{H}^{l,t})/T_{tr}$, which is sent together with the non-compressed embeddings $\boldsymbol{H}$ to the requesting end w_i. The messages sent by each responding worker at the tth iteration are defined as:

$$\boldsymbol{M} = \begin{cases} C_{bits}(\boldsymbol{H}), & 0 \le t \bmod T_{tr} < T_{tr} - 1 \\ \boldsymbol{H}, \boldsymbol{M}_{cr}, & t \bmod T_{tr} = T_{tr} - 1 \end{cases}$$

Then, at each iteration t, a worker has three embedding matrices, $\hat{\boldsymbol{H}}_{pdt}^{l,t}$, $\hat{\boldsymbol{H}}_{cps}^{l,t}$, and $\hat{\boldsymbol{H}}_{avg}^{l,t}$, which are calculated by Eqs. 7 to 9 below. Here, $\hat{\boldsymbol{H}}_{pdt}^{l,t}$ is an estimation based on the last non-compressed embeddings $\boldsymbol{H}^{l,\lfloor t/T_{tr} \rfloor \cdot T_{tr}}$ sent from the responding ends together with the changing ratio matrix $\boldsymbol{M}_{cr}$, where $\lfloor \cdot \rfloor$ represents the "floor" function. $\hat{\boldsymbol{H}}_{cps}^{l,t}$ is the compressed embeddings as computed following the procedure described in Section IV-A; and $\hat{\boldsymbol{H}}_{avg}^{l,t}$ is the average of $\hat{\boldsymbol{H}}_{pdt}^{l,t}$ and $\hat{\boldsymbol{H}}_{cps}^{l,t}$.

$$\hat{\boldsymbol{H}}_{pdt}^{l,t} = \boldsymbol{H}^{l,\lfloor t/T_{tr} \rfloor \cdot T_{tr}} + \boldsymbol{M}_{cr}^{l} \times (t \bmod T_{tr} + 1) \quad (7)$$

$$\hat{\boldsymbol{H}}_{cps}^{l,t} = C_{bits}(\boldsymbol{H}) \quad (8)$$

$$\hat{\boldsymbol{H}}_{avg}^{l,t} = (\hat{\boldsymbol{H}}_{pdt}^{l,t} + \hat{\boldsymbol{H}}_{cps}^{l,t})/2 \quad (9)$$

To further reduce the communication costs, we maintain these three approximate representations on the responding ends, and only send the most accurate approximation upon each request. In particular, we compute the L_1 distances $\boldsymbol{S}$ between the an original embedding and its three approximate embeddings by Eq. 10, and we send the approximate embedding with the smallest L_1 distance to the requesting end, i.e., $\hat{\boldsymbol{H}}_v[k]$ with $k = \arg\min\ \boldsymbol{S}_v$.

$$\boldsymbol{S}_v = \sum_{i=1}^{d^l} |\hat{\boldsymbol{h}}_{(v,i)}^{l,t} - \boldsymbol{h}_{(v,i)}^{l,t}| \quad (10)$$

Continue with the example in Fig. 3, we illustrate the error compensation process in Fig. 4, where $\boldsymbol{S}$ is a matrix with N_{rmt} rows and three columns, and N_{rmt} denotes the number of neighbors in *nodes*. Each row corresponds to a vertex embedding to be sent to a requesting end, and each column represents the L_1 distance of an approximate representation. The predicted approximation of $\boldsymbol{h}_5$ and the average approximation of $\boldsymbol{h}_6$ yield the smallest L_1 distances, respectively.

Note that we do not need to send the compressed values for v_5 because such embeddings can be predicted by the requesting workers. These non-sent embedding messages offset part of the communication costs induced by sending accurate embeddings at the last iteration of every trend-group.

We send the ID of each of these three approximations which only takes 2-bits, i.e., 00, 01 and 10 for compressed, predicted, and average approximations, respectively. We also send the corresponding compressed embedding when compressed or average approximations are selected. When a predicted embedding is selected, we can compute the embedding directly on the requesting end. There are three kinds of granularity for the approximate representations, including element-wise, vertex-wise and matrix-wise schemas. We use vertex-wise approximations, which yields the best balance between the message size and the accuracy empirically.

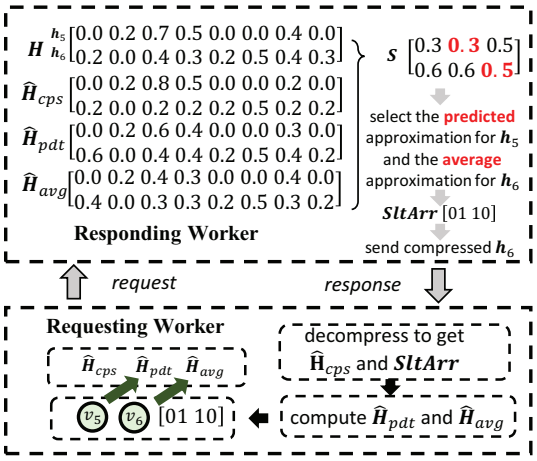


Fig. 4: Error Compensation in Forward Propagation

Adaptive Bit-Tuner. Further, we design an adaptive bit tuner to compression bits B. Our idea is to monitor the proportion P_{pred} of predicted approximate embeddings selected by the responders. When more predicted approximate embeddings are selected, there are smaller communication costs because such embeddings do not need to be sent, i.e., $P_{pred} \propto \frac{1}{Comm}$. This is also an indicator to suggest that the compressed embeddings are too lossy, where a larger number of bits B should be used in the forthcoming iterations. On the other hand, if the predicted approximate embeddings are rarely selected, we may use fewer bits for compression to reduce the communication costs in the forthcoming iterations. Empirically, we increase B when the predicted approximate embeddings are selected for more than 60% of the vertices, while we decrease B when the ratio drops below 40%.

Algorithm 3: *ReqEC-FP* in Requesting Workers

Input: Layer ID l, Remote node set *rmtNodeSet*, Number of bits B, Trend parameter T_{tr}, Iteration number t
Output: Remote neighbor embeddings *rmtNeiEmbs*

// get embeddings from other workers
1 *rm* = getEmbsRpcClient (l, *rmtNodeSet*, B, T_{tr}, t)
2 **if** $(t+1)$ % T_{tr} == *0* **then**
3 $\quad \boldsymbol{H}^{l,[(t+1)/T_{tr}]}$, $\boldsymbol{M}_{cr}^{l}$ = *rm*.parse()
4 $\quad \boldsymbol{H}_{rmt} = \boldsymbol{H}^{l,[(t+1)/T_{tr}]}$
5 **else**
6 $\quad \hat{\boldsymbol{H}}_{cps}^{l,t}$, $SltArr$ = *rm*.decompress()
7 $\quad \hat{\boldsymbol{H}}_{pdt}^{l,t} = \boldsymbol{H}^{l,[t/T_{tr}]\cdot T_{tr}} + \boldsymbol{M}_{cr}^{l} \times (t\%T_{tr}+1)$
8 $\quad \hat{\boldsymbol{H}}_{avg}^{l,t} = (\hat{\boldsymbol{H}}_{pdt}^{l,t} + \hat{\boldsymbol{H}}_{cps}^{l,t})/2$
9 $\quad \boldsymbol{H}$=[$\hat{\boldsymbol{H}}_{cps}^{l,t}$, $\hat{\boldsymbol{H}}_{pdt}^{l,t}$, $\hat{\boldsymbol{H}}_{avg}^{l,t}$]
10 $\quad$ **for** v ***in*** *rmtNodeSet* **do**
11 $\quad\quad sid$=*SltArr*[v]
12 $\quad\quad \boldsymbol{H}_{rmt}[v] = \boldsymbol{H}[sid][v]$
13 $\quad$ **if** l == L **then**
14 $\quad\quad proportion$ = *rm*.getProportion()
15 $\quad\quad$ **if** $proportion > 0.6$ **then**
16 $\quad\quad\quad B = (B < 16)\ ?\ B = B*2 : B$
17 $\quad\quad$ **if** $proportion < 0.4$ **then**
18 $\quad\quad\quad B = (B > 1)\ ?\ B = B/2 : B$
19 **return** $\boldsymbol{H}_{rmt}$

Algorithm 4: *ReqEC-FP* in Responding Workers

Input: Layer ID l, Remote node set *rmtNodeSet*, Number of bits B, Trend parameter T_{tr}, Iteration number t
Result: Integrating Messages *rm*

// respond embeddings to other workers
1 ReplyMessage rm
2 **if** $(t+1)$ % T_{tr} == *0* **then**
3 $\quad \boldsymbol{H}_{res}$ = $\boldsymbol{H}^{l}$[*rmtNodeSet*]
4 $\quad \boldsymbol{M}_{cr}^{l} = (\boldsymbol{H}_{res} - \boldsymbol{H}_{last}^{l})/T_{tr}$
5 $\quad \boldsymbol{H}_{last}^{l} = \boldsymbol{H}_{res}^{l}$
6 $\quad rm$.buildMessage($\boldsymbol{H}_{res}$, $\boldsymbol{M}_{cr}$)
7 **else**
8 $\quad \boldsymbol{H}^{l,t} = \boldsymbol{H}$[*rmtNodeSet*]
9 $\quad \hat{\boldsymbol{H}}_{cps}^{l,t} = compress(\boldsymbol{H}^{l,t})$
10 $\quad \hat{\boldsymbol{H}}_{pdt}^{l,t} = \boldsymbol{H}^{l,[t/T_{tr}]\cdot T_{tr}} + \boldsymbol{M}_{cr}^{l} \times (t\%T_{tr}+1)$
11 $\quad \hat{\boldsymbol{H}}_{avg}^{l,t} = (\hat{\boldsymbol{H}}_{pdt}^{l,t} + \hat{\boldsymbol{H}}_{cps}^{l,t})/2$
12 $\quad \boldsymbol{S} = [\sum|\hat{\boldsymbol{H}}_{cps}^{l,t} - \boldsymbol{H}^{l,t}|, \sum|\hat{\boldsymbol{H}}_{pdt}^{l,t} - \boldsymbol{H}^{l,t}|, \sum|\hat{\boldsymbol{H}}_{avg}^{l,t} - \boldsymbol{H}^{l,t}|]$
13 $\quad SltArr$ = argmin(S^T, $axis = 1$)
$\quad$ // filter out the predicted embedding
14 $\quad \hat{\boldsymbol{H}}_{cps}^{l,t}$ = filter($\hat{\boldsymbol{H}}_{cps}^{l,t}$, $SltArr$)
$\quad$ // coutX couts the elements valued 1
15 $\quad proportion$ = coutX($SltArr$, $value$=1) / $SltArr$.size()
16 $\quad rm$.buildMessage($SltArr$, $\hat{\boldsymbol{H}}_{cps}^{l,t}$, $proportion$)
17 **return** rm

Algorithm 3 and Algorithm 4 summarize the embedding requesting and responding process with compression and error compensation in FP. To start the process, a worker sends a request to get the embeddings of the neighboring vertices (line 1, Algorithm 3). A responding worker receives a request and computes the reply message rm. There are two cases. The first is to return non-compressed embeddings and a matrix of changing rates (lines 2 to 6, Algorithm 4); the second is to return compressed embeddings and supporting information for embedding selection and bit-tuning (lines 8 to 16, Algorithm 4). Next, the requesting workers parse rm , which also has two different cases (lines 2 to 18, Algorithm 3). The first is to parse the matrix of changing rates and the non-compressed embeddings (lines 3 and 4). The second is to parse and decompress the embeddings from rm (lines 6 to 9). Then, the embeddings of the neighboring vertices are reconstructed based on the selection indicators (lines 10 to 12). We also update B and choose its value from $\{1, 2, 4, 8, 16\}$. When the proportion of predicted embeddings is larger than 0.6 and $B < 16$, we double B for less aggressive compression. When the proportion is smaller than 0.4 and $B > 1$, we reduce B by half to reduce the communication costs. The proportion of predicted embeddings on a responding worker is computed on the choosing results of all the requesting workers. Note that different numbers of bits may be used for responding workers.

C. Compensation in Backward Propagation (ResEC-BP)

During BP, we use a similar compression for the embedding gradients of $L-1$ layers ($\boldsymbol{G}^2, \ldots, \boldsymbol{G}^L$). We compensate for the compression errors in BP in this subsection. At the tth iteration, compressed embeddings are responded in each layer. At the (t+1)-th iteration, before the embeddings are sent to the requesting workers in each layer, the compression error of the tth iteration is added to the embeddings of the (t+1)-th iteration, which are compressed and sent afterwards.

The left half of Fig. 5 shows the message accessing of the lth layer at the tth iteration, while the right half shows that of the lth layer at the (t+1)-th iteration. There are four steps of embedding requests in the two iterations. In the first step, v_3 and v_4 in the ith worker request for the embeddings of v_5 and v_6 of the lth layer. The second step computes the errors $\boldsymbol{\delta}_5^{l,t}$ and $\boldsymbol{\delta}_6^{l,t}$ generated by compression based on Eq. 11, which will be transferred to an error compensator. The third step occurs in the (t+1)-th iteration when v_3 and v_4 request for the neighboring vertices' embeddings again. The responding worker j prepares the embeddings of v_5 and v_6, which are sent to the error compensator. In the fourth step, the errors generated in the tth iteration are compensated into the current embeddings. The compensated embeddings will be compressed and sent to the requesting worker i, which are calculated by Eq. 12.

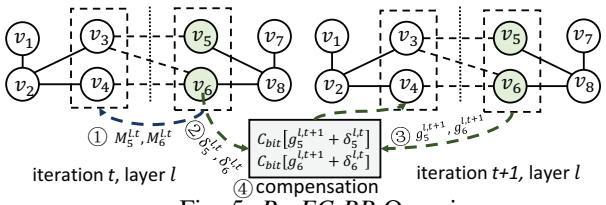


Fig. 5: *ResEC-BP* Overview

$$\boldsymbol{\delta}_v^{l,t} = \boldsymbol{g}_v^{(l,t)} + \boldsymbol{\delta}_v^{l,t-1} - C_{bit}[\boldsymbol{g}_v^{(l,t)} + \boldsymbol{\delta}_v^{l,t-1}] \qquad (11)$$

$$\boldsymbol{M}_v^{l,t+1} = C_{bit}[\boldsymbol{g}_v^{l,t+1} + \boldsymbol{\delta}_v^{l,t}] \tag{12}$$

We summarize the process of *ResEC-BP* in Algorithm 5 and Algorithm 6. In Algorithm 5, the requesting workers request for the neighboring vertices' embedding gradients (line 1), decompress (line 2) and return them (line 3). When the responding workers receive the requests, they start to build the replying message rm. As shown in Algorithm 6, first, they identify their embedding gradients needed by the requesting workers based on *rmtNodeSet* (line 2), which will be compensated by the error generated in the last iteration (line 3). Before compressing the compensated embedding gradients, we should compute the maximum and minimum values since they will not be normalized into a unit ball (lines 4 and 5). After building the replying messages, we compute the new error of this iteration, and return the messages to the requesting workers (lines 9 to 11).

Algorithm 5: *ResEC-BP* in Requesting Workers

Input: Layer ID l, Remote node set *rmtNodeSet*, Number of bits B

Output: Remote neighbor embeddings $\boldsymbol{G}_{rmt}$

1 $message$ = getRmtG(l,*rmtNodeSet*,B)
2 $\boldsymbol{G}_{rmt}$ = $message$.decompress()
3 **return** $\boldsymbol{G}_{rmt}$

Algorithm 6: *ResEC-BP* in Responding Workers

Input: Layer ID l, Remote node set *rmtNodeSet*, Number of bits B

Output: Responded message rm

1 ReplyMessage rm
2 $\boldsymbol{G}^{l,t}$ = $\boldsymbol{G}[rmtNodeSet]$
3 $\boldsymbol{G}_{cpt}^{l,t} = \boldsymbol{G}^{l,t} + \boldsymbol{\delta}^{l,t-1}$
4 max, min = getMaxMin($\boldsymbol{G}_{cpt}^{l,t}$)
5 $\boldsymbol{M}^{l,t}$ = compress($\boldsymbol{G}_{cpt}^{l,t}$,$max$,$min$)
6 rm.buildMessage($\boldsymbol{M}^{l,t}$)
7 $\boldsymbol{\delta}^{l,t} = \boldsymbol{G}^{l,t} - \boldsymbol{M}^{l,t}$
8 **return** rm

To analyze the theoretical error bounds of *ResEC-BP*, Eq. 13 and Eq. 14 are given to constrain the upper bounds of the compression errors and the embedding gradients, respectively, where $||\cdot||$ denotes the L_2 norm for matrices, and α is a constant. These two inequalities are common conditions for analyzing the compression error of iterative tasks [32]. Based on them, we give an error bound for each layer l at the tth iteration.

$$\mathbb{E}||\boldsymbol{x} - C(\boldsymbol{x})||^2 \leq \alpha^2||\boldsymbol{x}||^2 \tag{13}$$

$$\mathbb{E}||\boldsymbol{G}_{t,l}||^2 \leq G^2 \tag{14}$$

Theorem 1: ResEC-BP bounds the expected compression error of the embedding gradients $\boldsymbol{G}$ by $\mathbb{E}||\boldsymbol{\delta}_{t,l}||^2 \leq \frac{(1+\alpha)^{L-l}\cdot G^2}{1-\alpha^2(1+\frac{1}{\rho})}(0 < \alpha < \frac{\sqrt{2}}{2})$.

Proof:

$$\begin{aligned}
&\mathbb{E}||\boldsymbol{\delta}_{t,l}||^2 = \mathbb{E}||\boldsymbol{G}_{t,l} + \boldsymbol{\delta}_{t-1,l} - C[\boldsymbol{G}_{t,l} + \boldsymbol{\delta}_{t-1,l}]||^2 \\
&\leq \alpha^2\mathbb{E}||\boldsymbol{G}_{t,l} + \boldsymbol{\delta}_{t-1,l}||^2 \\
&\leq \alpha^2\mathbb{E}||\boldsymbol{G}_{t,l}||^2 + \alpha^2\mathbb{E}||\boldsymbol{\delta}_{t-1,l}||^2 \\
&+ \alpha^2(\rho\cdot\mathbb{E}||\boldsymbol{G}_{t,l}||^2 + \frac{1}{\rho}\cdot\mathbb{E}||\boldsymbol{\delta}_{t-1,l}||^2),\ (\rho > 0) \\
&\leq \alpha^2(1+\rho)\mathbb{E}||\boldsymbol{G}_{t,l}||^2 + \alpha^2(1+\frac{1}{\rho})(\alpha^2(1+\rho) \\
&\cdot\mathbb{E}||\boldsymbol{G}_{t-1,l}||^2 + \alpha^2(1+\frac{1}{\rho})\mathbb{E}||\boldsymbol{\delta}_{t-2,l}||^2) \\
&= \sum_{s=1}^{t}\alpha^{2(t-s+1)}\cdot(1+\frac{1}{\rho})^{t-s}\cdot(1+\rho) \\
&\cdot(1+\alpha)^{L-l}\cdot\mathbb{E}||\boldsymbol{G}_{s,L}||^2 \\
&\leq \frac{\alpha^2(1-[\alpha^2(1+\frac{1}{\rho})]^t)}{1-\alpha^2(1+\frac{1}{\rho})}\cdot(1+\rho)\cdot(1+\alpha)^{L-l}\cdot G^2 \\
&\leq \frac{\cdot(1+\alpha)^{L-l}\cdot G^2}{1-\alpha^2(1+\frac{1}{\rho})}(\alpha < \frac{1}{\sqrt{1+\rho}}, \rho > 1)
\end{aligned} \tag{15}$$

System implementation. We implement EC-Graph in C++ and Python. To reuse the linear algebraic operator optimization, we use PyTorch as the computation backend, which is responsible for the graph neural network model definition and the computation of FP and BP. We use *gRPC*[2] as the communication framework with the *protobuf*[3] serialization technique. Data transformation between C++ and Python is implemented using *Pybind11* [4], which supports definitions of C++ objects and uses memory address for data transformation and conversion. EC-Graph is open-sourced[5].

V. Evaluation

We compare EC-Graph with state-of-the-art distributed GNN systems by their performance on real datasets.

A. Experimental Setup

GNN models and datasets. We use *Graph Convolutional Networks* (GCN) [1] to evaluate the performance of EC-Graph, which is a typical GNN model. GCN is designed for semi-supervised learning and uses a localized first-order approximation of spectral graph convolutions for aggregating the information of neighboring vertices. Moreover, *GraphSAGE* [2] is a general inductive framework and aggregates the neighboring information in the spatial domain. Since GCN and GraphSAGE enjoy similar performance improvements from our optimizations, we only show the results of GCN for conciseness. The number of layers in each GNN model ranges from 2 to 4, and each layer contains a full-connected layer and an activation function. We use *Adam* optimization, and the same learning rates are used for both EC-Graph and all baseline systems.

[2]https://github.com/grpc/grpc
[3]https://github.com/protocolbuffers/protobuf
[4]https://github.com/pybind/pybind11
[5]https://github.com/songzhen-neu/ecgraph

TABLE III: Datasets

Datasets	#Vertex	#Edges	#Features	#Classes	Degree
Cora	2,708	10,556	1,433	7	3.90
Pubmed	19,717	88,654	500	3	4.50
Reddit	232,965	114,615,892	602	41	491.99
OGBN-Products	2,449,029	123,718,024	100	47	50.52
OGBN-Papers	111,059,956	3,231,371,744	128	172	29.10

We use five real public datasets from PyTorch Geometric [24] and Open Graph Benchmark [36], which are commonly used graph datasets. The datasets are split into train/val/test subsets with sizes of 1,408/300/1,000, 12,816/1,971/4,930, 153,932/23,699/55,334, 196,615/39,323/2,213,091 and 1,207,179/125,265/214,338, respectively.

Environments and baselines. We use two CPU clusters. The first cluster consists of 13 machines, each of which has 32 GB DRAM and an Intel(R) Xeon(R) 4-core CPU E3-1226 v3 @ 3.30 GHz. The second cluster consists of 6 machines, each of which is equipped with 250 GB DRAM and an Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz with 32 cores. The second cluster is used for the largest dataset OGBN-Papers with billions of edges. The machines in each cluster are connected with a Gigabit Ethernet.

We compare with PyG [24], DGL [25], AliGraph [18], AGL [20], DistGNN [35] and DistDGL [19]. The first two are the most commonly used GNN libraries, which are for observing the distributed speedups. Others are distributed GNN systems grouped into two categories for fair comparison:

- **Sampling-based systems:** AliGraph, AGL, and DistDGL. AliGraph and AGL are the typical ML-centered based GNN systems, both of which are sampling-based GNN systems. DistDGL is a Graph-centered based system and also follows a sampling-based training mode. However, it adopts an online-sampling that chooses different neighbors for a vertex in each iteration.
- **Non-sampling-based systems:** DistGNN. It is a Graph-centered based system with a full-batch scheme, which reduces the communication traffic by the proposed delayed remote partial aggregation.

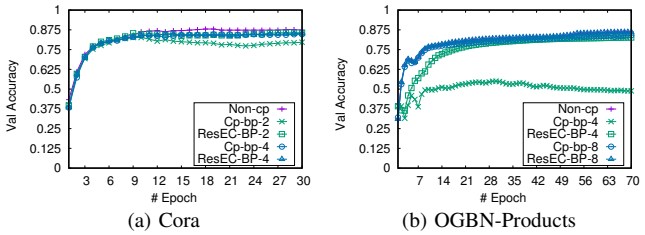
Since AGL [20] and DistGNN [35] are not open-sourced, we implement them following their original proposals. For AGL, its overall framework and main optimizations have been implemented, including GraphFlat, Sub-Graph Vectorization, and Graph Pruning. We ignore the time of disk I/O and graph vectorization since it can be hidden in AGL [20] by pipelining. For DistGNN, we implement its only distributed optimization *delayed remote partial aggregation* on our system, which is extended for an edge-cut partitioning. Other optimization tricks for single machines have been omitted for fair comparison since they are not implemented on the other systems.

Training Mode. We use the full-batch (full-graph) mode for training instead of mini-batch as this yields faster convergence as shown in previous studies [21], [22]. The numbers of layers, if unspecified, are set to 2, 2, 2, 3, and 3 for the five datasets, while the hidden layer sizes are set to 16, 16, 16, 256, and 256 respectively. Six machines are used for test except for scalability. We set $T_{tr} = 10$ empirically, which achieves a satisfactory performance for all datasets. We set the delayed round $r = 5$ for DistGNN following the original paper [35]. AliGraph-FG enables the full graph mode for AliGraph. No implementation details have been reported for this mode. Empirically, we found that AliGraph can achieve better performance when using the full graph mode. We use AliGraph-FG in all comparisons. Finally, we run EC-Graph on OGBN-Papers to evaluate the performance improvement of EC-Graph on large-scale graphs.

B. *Effectiveness of ReqEC-FP and ResEC-BP*

We first study the convergence of ReqEC-FP and ResEC-BP using different numbers of bits. Fig. 6 shows the results of ReqEC-FP, where Non-cp denotes no compression, while Cp-fp-i and ReqEC-FP-i denote using only compression and using compression with our compensation algorithms with i bits, respectively. We see that ReqEC-FP achieves a substantial improvement on both accuracy and convergence speed. Take Fig. 6c and Fig. 6h as an example. Given $B < 8$, the compression method cannot converge to a high test accuracy (Fig. 6c). In contrast, ReqEC-FP can reach a near-optimal accuracy in this case (Fig. 6h). Given $B = 8$, the compression method only yields its best accuracy 0.7567 at the 160th epoch, while the accuracy of ReqEC-FP reaches its best both earlier (at 100th epoch) and higher (0.9072). We find that the graphs with a larger average degree are more susceptible to the number of bits used in compression. For example, Reddit (average degree 491.99) cannot converge to a satisfactory accuracy even with an 8-bit compression, while Cora (average degree 3.9) can use a 2-bit compression and converge to an accuracy similar to that without compression. What's more, the compensation methods will oscillate, which is usually caused by an uneven changing of embeddings.

Similarly, we evaluate ResEC-BP with different numbers of bits. For conciseness, we only show a representative subset of the results in Fig. 7, where Non-cp denotes no compression, while Cp-bp-i and ResEC-BP-i denote using compression-only and using ResEC-BP with i bits, respectively. Again, we see that the error compensation helps achieve faster convergence and higher accuracy.

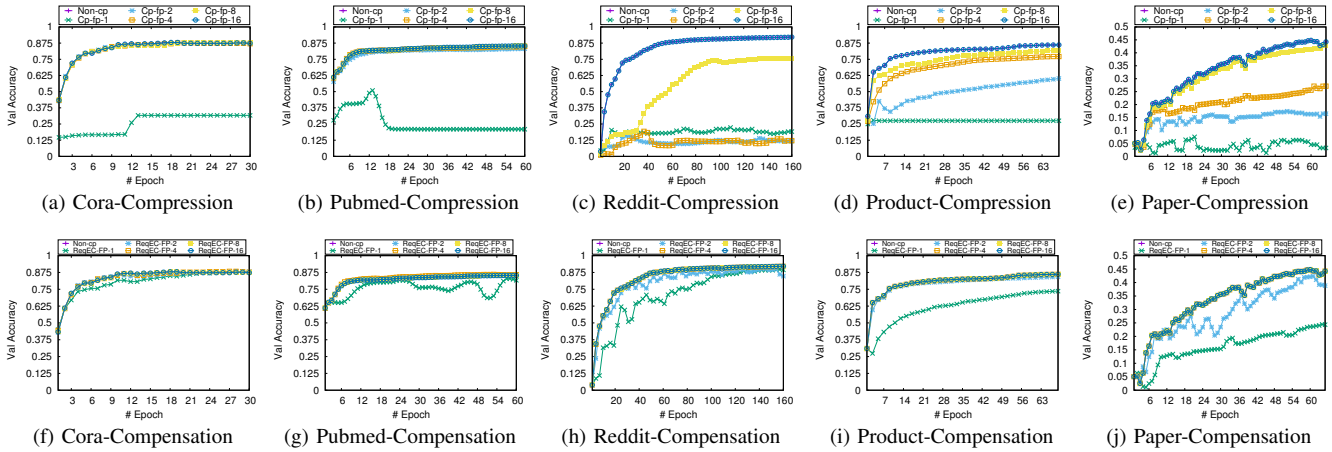


(a) Cora (b) OGBN-Products

Fig. 7: Results of BP with Different Numbers of Bits

C. *Convergence Time*

We carry our an ablation experiment in Fig. 8. In the figure, Cp-fp (Cp-bp) and ReqEC (ResEC) denote using compression-only and using error compensation for the forward (backward) propagation process. We use 2/4/1/2, 4/4/2/2,

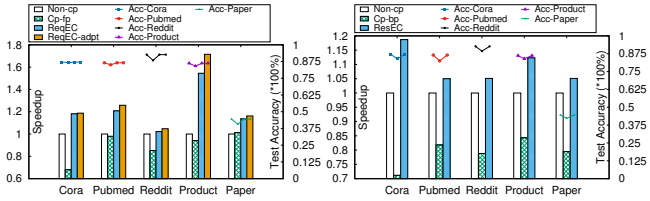


(a) Cora-Compression (b) Pubmed-Compression (c) Reddit-Compression (d) Product-Compression (e) Paper-Compression

(f) Cora-Compensation (g) Pubmed-Compensation (h) Reddit-Compensation (i) Product-Compensation (j) Paper-Compensation

Fig. 6: Results of FP with Different Numbers of Bits

8/8/2/4, 16/8/2/2, and 8/8/4/4 bits on each dataset for Cp-fp/Cp-bp/ReqEC/ResEC respectively such that the models can converge to the near-optimal test accuracy. ReqEC-adapt is the ReqEC-FP algorithm with the proposed adaptive bit tuner. We see that the compression methods without compensation perform even worse than the non-compression methods since a large amount of compressing errors can greatly degrade the convergence rate, leading to more iterations to converge. Besides, EC-Graph shows a smaller speedup in graphs with a large average degree such as Reddit, which are always computationally intensive.



(a) Speedup for FP (b) Speedup for BP

Fig. 8: Ablation Study. Histograms indicate the speedup (y-axis coordinates on the left) and lines represent the test accuracy (y-axis coordinates on the right).

D. End-to-end Performance Comparison

First, we compare EC-Graph with the existing systems on the average epoch time. We list the results of sampling and non-sampling based methods separately in Table IV, with the corresponding sampling ratios. EC-Graph does not outperform the standalone GNN systems DGL and PyG on Cora and Pubmed. This is because parameter updates and delays caused by distributed processing dominate the overall training time on these (relatively) small datasets. Similarly, other distributed systems (even with sampling) also suffer on these two datasets and are slower than DGL and PyG. For larger datasets, we achieve $1.46 \sim 1.57\times$ speedups on Reddit and $1.81 \sim 2.93\times$ speedups on Products over DGL in a full-batch and non-sampling based training mode. DistDGL claims that they have achieved a linear speedup over DGL, since they adopt a high-speed commercial network device (100Gbps), where communication would not be a bottleneck. Moreover, EC-Graph outperforms DistGNN on almost all datasets including both the overall convergence time and the average time per epoch. DistGNN needs more convergence iterations since only part of the vertices are updated by the lastest neighboring messages.

Next, we compare the sampling-based EC-Graph (denoted by EC-Graph-S) with the existing sampling-based distributed GNN systems. EC-Graph-S also achieves the best performance in this comparison. DistDGL adopts an online-sampling strategy, which dominates the overall training time when the computing and network bandwidth resources are constrained. AGL is slow because the disk I/O and sub-graph vectorization costs cannot be overlapped by graph computation under our cluster environment. Besides, AGL is more sensitive to the sampling ratio. For example, on Reddit, a 2-layer GNN under a higher sampling ratio of (10,5) runs even longer than a 3-layer GNN with a sampling ratio of (5,2,2). AliGraph-FG is an ML-centered system, which can generate many redundant computations. Further, we find that EC-Graph-S outperforms AliGraph-FG more as the number of layers increases, which shows the excellent layer-scalability of EC-Graph-S.

Fig. 9 reports the end-to-end times, which include preprocessing times and model training times. The preprocessing time of EC-Graph-S, AGL and AliGraph-FG consists of sampling, logical partitioning, physical assignment and data preprocessing, while that of DistDGL does not include sampling time (DistDGL uses online-sampling). EC-Graph (including EC-Graph-S) uses an equal-vertex partitioning strategy with *Hash*, where the logical partition time is almost negligible (e.g., 2.05 seconds on OGBN-Products with a single-threaded program). The training time reported is the full convergence time (time per epoch $\times$ the number of epochs till convergence). We list the speedups of EC-Graph over the existing systems on OGBN-Products for a clearer comparison. EC-Graph achieves up to $1.68 \sim 1.98\times$, $1.23 \sim 1.48\times$, $1.06 \sim 2.22\times$, $1.35 \sim 1.72\times$ and $4.34 \sim 5.39\times$ speedups over Non-cp,

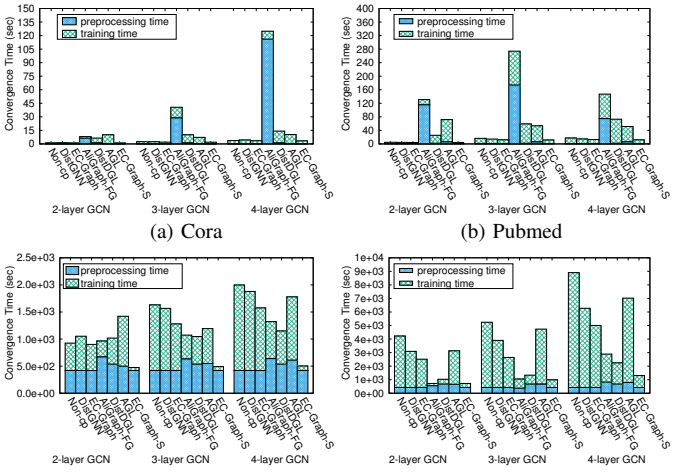
TABLE IV: Training Time Per Epoch (s)

Method	Cora			Pubmed			Reddit			OGBN-Products			OGBN-Papers		
	2-layer	3-layer	4-layer	2-layer	3-layer	4-layer	2-layer	3-layer	4-layer	2-layer	3-layer	4-layer	2-layer	3-layer	4-layer
DGL	**0.011**	**0.013**	**0.015**	**0.023**	**0.029**	**0.036**	6.056	8.954	12.05	44.39	92.72	-	-	-	-
PyG	0.012	0.018	0.023	0.066	0.105	0.144	-	-	-	-	-	-	-	-	-
DistGNN	0.034	0.047	0.074	0.049	0.091	0.109	4.218	6.744	8.584	26.68	33.12	50.84	-	-	-
EC-Graph	0.036	0.049	0.072	0.047	0.082	0.113	**3.854**	**6.142**	**8.255**	**24.58**	**31.68**	**48.24**	68.23	79.12	89.54
(sampling)	(full)	(20,10,5)	(10,5,5,5)	(full)	(10,10,5)	(5,5,5,1)	(10,5)	(5,2,2)	(5,5,1,1)	(20,5)	(10,5,1)	(10,5,2,2)	(10,10)	(10,10,10)	(10,10,10,10)
DistDGL	0.260	0.245	0.288	0.390	0.424	0.678	3.416	3.640	4.375	5.055	7.720	16.47	-	-	-
AGL	0.490	0.198	0.222	1.088	0.348	0.429	7.684	4.589	8.346	35.35	47.67	-	-	-	-
AliGraph-FG	0.061	0.278	0.546	0.232	0.514	0.683	2.417	3.124	4.864	1.987	7.153	24.35	-	-	-
EC-Graph-S	**0.036**	**0.047**	**0.069**	**0.047**	**0.078**	**0.104**	**0.461**	**0.513**	**0.596**	**4.239**	**6.640**	**9.243**	23.25	31.06	42.16

TABLE V: Test Accuracy

	Cora	Pubmed	Reddit	Product	Paper
DGL	87.00%	86.59%	92.64%	**86.23%**	-
PyG	87.02%	86.55%	-	-	-
DistGNN	86.90%	86.41%	92.19%	85.70%	-
EC-Graph	**87.10%**	**86.59%**	**92.66%**	86.18%	44.58%
DistDGL	86.70%	86.43%	**92.57%**	**85.74%**	-
AGL	86.70%	85.89%	92.14%	85.01%	-
AliGraph-FG	76.05%	83.59%	81.47%	82.02%	-
EC-Graph-S	**87.10%**	**86.59%**	92.15%	85.06%	43.56 %

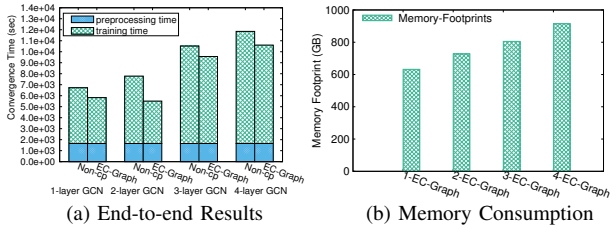
DistGNN, AliGraph-FG, DistDGL and AGL respectively. The accuracy of each approach is shown in Table V).



(a) Cora (b) Pubmed

(c) Reddit (d) OGBN-Products

Fig. 9: End-to-end Performance Evaluation

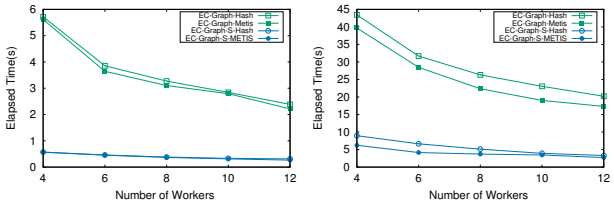


(a) End-to-end Results (b) Memory Consumption

Fig. 10: Results on OGBN-Papers.

E. Scalability with the Number of Machines

We show the scalability of EC-Graph and EC-Graph-S against the number of machines in Fig. 11 under two partitioning strategies *Hash* and *METIS*. Both EC-Graph and EC-Graph-S scale well with the number of machines. *METIS* has lower running times because of its lower communication costs. We did not use it as our default algorithm above because it takes much time to partition on big graphs. Balancing the partitioning time and the quality of partitions is another important research problem beyond the scope of this paper.



(a) Reddit (b) OGBN-Products

Fig. 11: Scalability of EC-Graph

VI. Conclusion

We propose a distributed system named EC-Graph for efficient and scalable GNN computation on CPU clusters. To reduce the computation and communication costs, we take a lossy compression approach to compress the messages communicated among the nodes in the cluster. We further propose two compensation algorithms named *ReqEC-FP* and *ResEC-BP*, respectively, which mitigate the degradation of convergence caused by the lossy compression. Experimental results on real graphs show EC-Graph outperforms DistGNN and DistDGL (state-of-the-art sampling and non-sampling based systems) by $1.10 \sim 1.48\times$ and $1.35 \sim 6.28\times$ on real datasets. EC-Graph runs on CPU clusters at present. We plan to extend EC-Graph to GPU clusters in future work. Meanwhile, since GPUs have stronger computation power than CPUs in general, we expect a GNN to train faster on each machine in a GPU cluster. The communication costs may become a stronger bottleneck, and our EC-Graph system will play a significant role in reducing such costs.

VII. Acknowledgement

This work is supported by the National Natural Science Foundation of China (62072083 and 61902366), the CCF-Huawei Database Innovation Research Funding.

REFERENCES

[1] Michal Defferrard, Xavier Bresson, Pierre Vandergheynst: Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. NIPS 2016: 3837-3845

[2] William L. Hamilton, Zhitao Ying, Jure Leskovec: Inductive Representation Learning on Large Graphs. NIPS 2017: 1024-1034

[3] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Li, Yoshua Bengio: Graph Attention Networks. ICLR (Poster) 2018

[4] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski: Pregel: A System for Large-Scale Graph Processing. SIGMOD Conference 2010: 135-146

[5] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, Carlos Guestrin: PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. OSDI 2012: 17-30

[6] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, Haibo Chen: SYNC or ASYNC: Time to Fuse for Distributed Graph-Parallel Computation. PPOPP 2015: 194-204

[7] Zhigang Wang, Yu Gu, Yubin Bao, Ge Yu, Jeffrey Xu Yu: Hybrid Pulling/Pushing for I/O-Efficient Distributed and Iterative Graph Computing. SIGMOD Conference 2016: 479-494

[8] Yuzhen Huang, Tatiana Jin, Yidi Wu, Zhenkun Cai, Xiao Yan, Fan Yang, Jinfeng Li, Yuying Guo, James Cheng: FlexPS: Flexible Parallelism Control in Parameter Server Architecture. Proc. VLDB Endow. 11(5): 566-579 (2018)

[9] Jiawei Jiang, Bin Cui, Ce Zhang, Lele Yu: Heterogeneity-aware Distributed Parameter Servers. SIGMOD Conference 2017: 463-478

[10] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, Bor-Yiing Su: Scaling Distributed Machine Learning with the Parameter Server. OSDI 2014: 583-598

[11] Eric P. Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, Yaoliang Yu: Petuum: A New Platform for Distributed Machine Learning on Big Data. KDD 2015: 1335-1344

[12] Martn Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, Xiaoqiang Zheng: TensorFlow: A System for Large-Scale Machine Learning. OSDI 2016: 265-283

[13] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, Zheng Zhang: MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. CoRR abs/1512.01274 (2015)

[14] Zhipeng Zhang, Bin Cui, Yingxia Shao, Lele Yu, Jiawei Jiang, Xupeng Miao: PS2: Parameter Server on Spark. SIGMOD Conference 2019: 376-388

[15] Pitch Patarasuk, Xin Yuan: Bandwidth optimal all-reduce algorithms for clusters of workstations. J. Parallel Distributed Comput. 69(2): 117-124 (2009)

[16] Alexander Sergeev and Mike Del Balso. Horovod: Fast and Easy Distributed Deep Learning in TensorFlow. CoRR, 2018.

[17] Jiawei Jiang, Pin Xiao, Lele Yu, Xiaosen Li, Jiefeng Cheng, Xupeng Miao, Zhipeng Zhang, Bin Cui: PSGraph: How Tencent trains extremely large-scale graphs with Spark? ICDE 2020: 1549-1557

[18] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, Jingren Zhou: AliGraph: A Comprehensive Graph Neural Network Platform. Proc. VLDB Endow. 12(12): 2094-2105 (2019)

[19] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, George Karypis: DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. CoRR abs/2010.05337 (2020)

[20] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, Yuan Qi: AGL: A Scalable System for Industrial-purpose Graph Machine Learning. Proc. VLDB Endow. 13(12): 3125-3137 (2020)

[21] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, Alex Aiken: Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. MLSys 2020

[22] Alok Tripathy, Katherine A. Yelick, Aydin Bulu: Reducing Communication in Graph Neural Network Training. CoRR abs/2005.03300 (2020)

[23] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, Lidong Zhou: Tux2: Distributed Graph Computation for Machine Learning. NSDI 2017: 669-682

[24] Fey Matthias, Lenssen Jan E. Fast Graph Representation Learning with PyTorch Geometric[C]. ICLR, 2019. 1-9.

[25] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, Zheng Zhang: Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. CoRR abs/1909.01315 (2019)

[26] CogDL: https://github.com/THUDM/cogdl

[27] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, Yafei Dai: NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. USENIX Annual Technical Conference 2019: 443-458

[28] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, Yinlong Xu: PaGraph: Scaling GNN training on large graphs via computation-aware caching. SoCC 2020: 401-415 ATC, 2019. 443458.

[29] Chao Tian, Lingxiao Ma, Zhi Yang, Yafei Dai: PCGCN: Partition-Centric Processing for Accelerating Graph Convolutional Network. IPDPS 2020: 936-945

[30] Jiawei Jiang, Fangcheng Fu, Tong Yang, Bin Cui: SketchML: Accelerating Distributed Machine Learning with Data Sketches. SIGMOD Conference 2018: 1269-1284

[31] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, Dong Yu: 1-Bit Stochastic Gradient Descent and Its Application to Data-Parallel Distributed Training of Speech DNNs. INTERSPEECH 2014: 1058-1062

[32] Sebastian U. Stich, Jean-Baptiste Cordonnier, Martin Jaggi: Sparsified SGD with Memory. NeurIPS 2018: 4452-4463

[33] Hanlin Tang, Chen Yu, Xiangru Lian, Tong Zhang, Ji Liu: DoubleSqueeze: Parallel Stochastic Gradient Descent with Double-pass Error-Compensated Compression. ICML 2019: 6155-6165

[34] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. International Conference on Parallel Processing, pp. 113-122, 1995.

[35] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj D. Kalamkar, Nesreen K. Ahmed, Sasikanth Avancha: DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks. CoRR abs/2104.06700 (2021)

[36] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, Jure Leskovec: Open Graph Benchmark: Datasets for Machine Learning on Graphs. NeurIPS 2020

[37] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, Gabriele Monfardini: The Graph Neural Network Model. IEEE Transactions on Neural Networks 20(1): 61-80 (2009)

[38] Ali Davoudian, Liu Chen, Hongwei Tu, Mengchi Liu: A Workload-Adaptive Streaming Partitioner for Distributed Graph Stores. Data Sci. Eng. 6(2): 163-179 (2021)