GraphTheta: A Distributed Graph Neural Network Learning System With Flexible Training Strategy

Yongchao Liu1,‡, Houyi Li2,†*, Guowei Zhang1, Xintan Zeng1, Yongyong Li1, Bin Huang1, Peng Zhang3,†, Zhao Li4,†, Xiaowei Zhu1, Changhua He1, Wenguang Chen1,†,
1Ant Group, 2Fudan University, 3Guangzhou University, 4Zhejiang University, 5Tsinghua University
China

ABSTRACT
Graph neural networks (GNNs) have been demonstrated as a powerful tool for analyzing non-Euclidean graph data. However, the lack of efficient distributed graph learning systems severely hinders applications of GNNs, especially when graphs are big and GNNs are relatively deep. Herein, we present GraphTheta, the first distributed and scalable graph learning system built upon vertex-centric distributed graph processing with neural network operators implemented as user-defined functions. This system supports multiple training strategies and enables efficient and scalable big-graph learning on distributed (virtual) machines with low memory. To facilitate graph convolutions, GraphTheta puts forward a new graph learning abstraction named NN-TGAR to bridge the gap between graph processing and graph deep learning. A distributed graph engine is proposed to conduct the stochastic gradient descent optimization with a hybrid-parallel execution, and a new cluster-hatched training strategy is supported. We evaluate GraphTheta using several datasets with network sizes ranging from small- to large-scale. Experimental results show that GraphTheta can scale well to 1,024 workers for training an in-house developed GNN on an industry-scale Alipay dataset of 1.4 billion nodes and 4.1 billion attributed edges, with a cluster of CPU virtual machines (dockers) of small memory each (5–12GB). Moreover, GraphTheta can outperform DistDGL by up to 2.02×, with better scalability, and GraphLearn by up to 30.56×. As for model accuracy, GraphTheta is capable of learning as good GNNs as existing frameworks. To the best of our knowledge, this work presents the largest edge-attributed GNN learning task in the literature.

PVldb Reference Format:
Yongchao Liu1,‡, Houyi Li2,†*, Guowei Zhang1, Xintan Zeng1, Yongyong Li1, Bin Huang1, Peng Zhang3,†, Zhao Li4,†, Xiaowei Zhu1, Changhua He1, Wenguang Chen1,†, GraphTheta: A Distributed Graph Neural Network Learning System With Flexible Training Strategy. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVldb Artifact Availability:
This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@pvldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

† Joint first authors; ‡ Contributed to this work when the authors worked in Ant Group or Alibaba Group. Correspond to yongchao.ly, yuanben.cwg@antgroup.com.

The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION
Graph neural networks (GNNs) [15, 29, 43, 60, 73–75, 109] have been popularly used in analyzing graph data and achieved promising results in various applications, such as node and graph classification [16, 24, 95, 97], link prediction [52, 108], program analysis [2, 4, 61], recommendation [19, 55, 100, 102, 116] and quantum chemistry [25]. Currently, GNNs are implemented as proof-of-concept built upon deep learning (DL) frameworks like Tensorflow [1], PyTorch [71] and MXNet [12]. They are either shared-memory [19, 58, 59, 65, 67, 69, 70, 85, 88, 101] or use distributed computing [20, 23, 42, 57, 80, 87, 96, 111–113, 116]. The shared-memory frameworks are restricted to host memory size and difficult to handle big graphs. One approach to overcoming this constraint is to use external storage to store large graphs [70], and another is to use graph storage servers. Most of the distributed frameworks employ graph servers, which either require every server to hold the entire graph or distribute the whole graph over a set of servers. In principle, the architecture of these distributed frameworks can be abstracted as Figure 1, where classical graph computing and MapReduce [17, 104] methods are usually used. In addition, these distributed frameworks usually target only one type of training strategy.

For GNN training, global-batch [16, 49] and mini-batch [31] are two popular strategies. Global-batch performs full graph convolutions across an entire graph by multiplying feature matrices with graph Laplacian. It requires calculations on the entire graph, regardless of graph density. In contrast, mini-batch conducts localized convolutions on a batch of subgraphs, where a subgraph is constructed from a node with its neighbors. Typically, subgraph construction meets the challenge of size explosion, especially when the node degrees and the neighborhood exploration depth (neighborhood exploration depth is equal to the number of graph convolution layers) are very large [10, 53, 54, 98, 99]. This severely limits the scalability of mini-batch in processing dense graphs (dense graphs refer to graphs of high density) or sparse ones with highly skewed node degree distributions, and the situation further deteriorates very much for deep GNNs [10, 53, 54, 98, 99]. Specifically, on one hand, not even mentioning dense graphs, high-degree nodes in a highly skewed graph can often cause a worker to become short of memory space to fully store a subgraph. For example, in the Alipay dataset (see Section 5.1), the degree of a node can reach hundreds of thousands. On the other hand, a deep neighborhood exploration results in the explosion of a subgraph. In the Alipay dataset, a batch of subgraphs obtained from a two-hop neighborhood exploration
The second challenge is to allow for conveniently exploring new training strategies in addition to the existing mini-batch and global-batch methods. For this challenge, existing tensor-based frameworks use sparse tensors\(^1\) to represent subgraphs and full graphs, which poses challenges to support the multiple training strategies described in this paper (especially global-batch) for the following reasons. First, sparse tensors are nodes of the auto-differentiable computational graph and excessive sparse tensors will make the computational graph too large to be efficiently processed by the runtime of these frameworks, as described above. Second, sparse tensors are with shared-memory and their sizes must be considerably constrained in local machines, thus infeasible for global-batch on large graphs. To address global-batch, practitioners usually transform its computation to that of mini-batch. Third, sparse tensors must carefully maintain the node/edge correspondence to the original graph (e.g. by some re-indexing techniques). In contrast, we introduce a distributed subgraph abstraction to transform these strategies into a unified computation (refer to Section 4.2). The technical novelty of our distributed subgraph abstraction lies in that it unifies the processing of multiple training strategies and provides a consistent abstraction for the hybrid-parallel execution of distributed GNN training and inference. To the best of our knowledge, no existing system has ever made this.

The third challenge is to enable deep neighborhood exploration without neighbor sampling. To solve this challenge, with the help of our distributed subgraph abstraction built in the distributed graph training engine, neighborhood exploration only introduces a little extra storage overhead. This overhead comes from the active set data structure that records the active status of nodes and edges, which can be proportional to the number of nodes or edges subject to implementations. Therefore, sampling is no longer a necessity for the sake of reducing space complexity. Furthermore, our distributed subgraph abstraction stores node/edge embedding (or activations) in place along with the graph topology, and thus naturally leads to distributed storage of embeddings (refer to Section 4).

To solve these challenges, we propose a new parallel and distributed GL system named GraphTheta, which is built from the ground up to support GNN training and inference and does not depend on existing DL frameworks like TensorFlow [1] and PyTorch [71]. The rationale behind such a design is explained as follows. On one hand, graph operations can dominate the runtime [111] and thus demand efficient graph processing infrastructures. Built on top of our proprietary high-performance distributed graph processing system (also capable of serving existing DL frameworks), GraphTheta can keep seamless integration with our system and also makes its iterative development process independent of third-party DL frameworks. On the other hand, the training procedure of DL can be modeled as an auto-differentiable computational graph of data and operator (or neural network function) nodes. For each training step, we can build the computational graph by means of treating vertices/edges as data nodes and then building data-operator or operator-operator linkages following the GNN and the input graph topology. In this case, large graphs will make the computational graph inefficiently processed by the runtime

---

\(^1\)Sparse tensors are used to efficiently store and process tensors containing many zero values and typical formats include coordinate list (COO), compressed sparse row (CSR) and compressed sparse column (CSC).

---

Figure 1: A typical architecture of distributed frameworks.

Figure 2: The workflow of GraphTheta.
of existing frameworks. Subject to specific implementations, the size of the computation graph can have a higher order of magnitude than the input graph. Figure 2 illustrates the workflow of our system. There are two roles in our system, i.e. master and worker. The master process coordinates the execution of all workers, monitors their health, manages checkpoints, and directs the learning procedure including training and inference. The workers wait and receive commands from the master, and then execute them. The communication between the master and workers is done using remote procedure calls.

We have conducted extensive experiments on various networks to evaluate the performance of our system. Our experimental results revealed that GraphTheta can learn GNNs as well as existing frameworks in terms of model accuracy. Furthermore, GraphTheta scales well to 1,024 CPU workers in our production Kubernetes [51] CPU cluster and can train an in-house developed GNN on the Alipay dataset of 1.4 billion nodes and 4.1 billion attributed edges using three training strategies. To our knowledge, this is the largest GNN learning task that uses edge attributes of a billion-scale industrial network in the literature.\(^2\) Furthermore, GraphTheta outperforms DistDGL both in terms of training speed (up to 2.02× faster on the same machines) and scalability, and can run 2.61 ∼ 30.56× faster than GraphLearn (the open-source version of AliGraph [116]). Our technical contributions are summarized as follows.

- We present a distributed GL system implemented based on a vertex-centric graph processing programming model. This system proposes a new distributed graph training engine developed with gradient propagation support. This engine implements a new hybrid-parallel execution and supports GNN training and inference via a unified implementation. Compared to existing data-parallel executions, the new engine can scale up to big dense/sparse graphs and enable large neighborhood exploration.
- GraphTheta introduces a new GL abstraction NN-TGAR (Neural Network Transform-Gather-Apply-Reduce), which enables user-friendly programming (supporting distributed training) and bridges the gap between graph processing paradigms and deep learning computational patterns. This abstraction centers around the semantics of nodes and edges, instead of tensors as in conventional DL frameworks, and is used to compose DNNs by orchestrating operations on nodes and edges.
- To alleviate the redundant calculation among batches, GraphTheta supports a new type of training strategy, i.e. cluster-batch [14], which performs graph convolution on a cluster of nodes and can be taken as a generalization of mini-batch or global-batch. In particular, for each training strategy, our system scales well to 1,024 workers on the billion-scale industrial Alipay dataset in a cluster of CPU virtual machines of small memory each.

2 PRELIMINARY

2.1 Notations
A graph can be defined as \(G = \{V, E\}\), where \(V\) and \(E\) are nodes and edges respectively. For simplicity, we define \(N = |V|\) and \(M = |E|\). Each node \(v_i \in V\) is associated with a feature vector \(h_i^0\), and each edge \(e(i,j) \in E\) has a feature vector \(e_{i,j}\), a weight value \(a_{i,j}\). To represent the structure of \(G\), we use an adjacency matrix \(A \in \mathbb{R}^{N \times N}\), where \(A(i,j) = a_{i,j}\) if there exists an edge between nodes \(i\) and \(j\), otherwise 0.

2.2 Graph Neural Networks
Existing graph learning tasks are often comprised of encoders and decoders as stated in [32]. Encoders map high-dimensional graph information into low-dimensional embeddings. Decoders are application-driven. The essential difference between GNNs relies on the encoders. Encoder computation can be performed by two established methods [110]: spectral and propagation. The spectral method generalizes convolution operations on two-dimensional grid data to graph-structured data as defined in [18, 49], which uses sparse matrix multiplications to perform graph convolutions. The propagation method describes graph convolutions as a message propagation operation, which is equivalent to the spectral method (refer to Section A.1 in the supplementary material\(^3\) for the proof).

In this paper, we present an algorithmic framework of Message Propagation based Graph Neural Network (MPGNN) as a typical use case to elaborate the compute pattern of NN-TGAR and our new system GraphTheta. As shown in Algorithm 1, MPGNN can unify existing GNN algorithms under different training strategies. Recent work [26, 31, 84] focuses on propagating and aggregating messages and aims to deliver a general framework for different message propagation methods. The core difference between existing propagation methods lies in the projection (Line 6), the message propagation (Line 7), and the aggregation (Line 8) functions.

2.3 Cluster-batched Training
We study a new training strategy, namely cluster-batched gradient descent, to address the challenge of redundant neighbor embedding computation under the mini-batch strategy. This training strategy was first used by the Cluster-GCN [14] algorithm and shows superior performance to mini-batch in some applications. This method can maximize per-batch neighborhood sharing by taking advantage of community detection (graph clustering) algorithms.

Cluster-batch first partitions a big graph into a set of smaller clusters and then generates a batch of data either based on one cluster or a combination of multiple clusters. Similar to mini-batch, cluster-batch also performs localized graph convolutions. However, cluster-batch restricts the neighbors of a target node into only one cluster, which is equivalent to conducting a full graph convolution on a cluster of nodes. Typically, cluster-batch generates clusters

---

\(^2\)The network in [102] has 3 billion nodes and 18 billion edges with no edge attributes.

\(^3\)https://github.com/yongchao-liu/graphtheta
by using a community detection algorithm based on maximizing intra-community edges and minimizing inter-community connections [5]. Note that community detection can run either beforehand or at runtime, based on the requirements. Moreover, cluster sizes are often irregular, resulting in varied batch sizes. It needs to be noted that differing from the cluster-batch in Cluster-GCN, our cluster-batch optionally allows for target nodes to access boundary neighbors outside the clusters. In the supplementary material, Figure A1 illustrates an example for cluster-batched computation and Table A1 compares the pros and cons of the three training strategies, implying the necessity to design a new GNN learning system that enables the exploration of different training strategies and a solution to address the limitations of existing architectures.

3 COMPUTE PATTERN

3.1 NN-TGAR

To solve the GNN learning in a hybrid-parallel fashion, we present a general computation pattern abstraction, namely NN-TGAR, which can perform the forward and backward computation of GNNs on a big (sub-)graph distributionally. This abstraction decomposes an encoding layer (explained in the context of encoders and decoders as described in Section 2.2) into a sequence of independent stages, i.e., NN-Transform (NN-T), NN-Gather (NN-G), Sum, NN-Apply (NN-A), and Reduce. Our method allows distributing the computation over a cluster of machines. NN-T is executed on each node to transform the results and generate messages. NN-G is applied to each edge, where the inputs are edge values, source node, destination node, and the messages generated in the previous stage. After each iteration, this stage updates edge values and sends messages to the destination node (maybe on other workers). In Sum, each node accumulates the received messages by means of a non-parameterized method like averaging, concatenation or a parameterized one such as LSTM [63]. The resulting summation is updated to the node by NN-A.

Different from the vertex-centric programming abstraction Gather-Sum-Apply-Scatter (GAS) proposed by PowerGraph [27] for conventional graph processing applications, NN-T, NN-G and NN-A are implemented as neural network functions. In forwards, the trainable parameters also join in the calculation of the three stages, but kept unchanged. In backwards, the gradients of these parameters are generated in NN-T, NN-G and NN-A stages, which is used in the final stage NN-Reduce for parameter updating. NN-TGAR can be executed either on the entire graph or subgraphs, subject to the training strategy used.

Intuitively, we can understand NN-TGAR as follows. If having only operations on individual nodes (e.g., imagining a node is an image), an NN can be realized only by NN-T. The forward and backward computation logics are almost identical to the NNs in the tensor-based DL frameworks. If needding message passing along with edges, NN-G, Sum and NN-A will be used. If a node aggregates information from its neighbor along every out-edge in the forward computation, this node will aggregates gradient from its neighbor along every in-edge in the backward, and vice versa.

3.2 Forward

The forward of a GNN model can be described as $K + 2$ passes of NN-TGA, as each encoding layer can be described as one pass of NN-TGAR. The decoder and loss functions can be separately described as a single NN-T operation. Recall that the decoder functions are application-driven. A decoder function can be described by a single NN-T operation in node classification, and a combination of NN-T and NN-G in link prediction. Without loss of generality, we use node classification as the default task in this paper. The forward is comprised of $K$ passes of NN-TGA from the first to $K$-th encoding layer, one NN-T operation for the decoder function, and one for the loss calculation. In the forward of each encoding layer, the projection function is executed on NN-T and the propagation function on NN-G. However, the aggregation function is implemented by a combination of $\text{Sum}$ and $\text{NN-A}$, which corresponds to the accumulate part and the apply part defined as follows:

$$M^k_i = \text{Acc}_{\mathcal{K}}(\{m_{j-i}^{k-1}\}_{j \in \mathcal{N}(i)}) h^{k(1)}_i, \quad (1a)$$

$$h^k_i = \text{Apply}_k(h^k_i, M^k_{\mathcal{N}(i)}|\mu^k_i^{(2)}), \quad (1b)$$

where $\mu_k = [\mu^{(1)}_k, \mu^{(2)}_k]$. If the accumulate part is not parameterized by mean-pooling, $\mu^{(1)}_k = 0$ and $\mu^{(2)}_k = 1$.

As shown in Figure 3(a), the embedding of the $(k-1)$-layer $h^{k-1}_i$ is transformed to $n^{k}_i$ using a neural network $\text{Proj}_k(\cdot|W_k)$ for all the three nodes in NN-T. The NN-G stage collects messages both from the neighboring nodes $\{v_1, v_2\}$ (and $v_1$) and from adjacent edges $\{e_1, e_2\}$ (and $e_2$) to the central node $v_1$ (and $v_2$) through the propagation function $\text{Prop}_k(\cdot|\theta_k)$. The output of this stage is automatically accumulated by $\text{Acc}_{\mathcal{K}}(\cdot)$, resulting in the summed message $M^k_{\mathcal{N}(i)}$ and $M^k_{\mathcal{N}(2)}$. The NN-A stage computes the new embeddings of nodes $\{v_1, v_2, v_3\}$, i.e., $h^k_1$, $h^k_2$ and $h^k_3$, as the output of this encoding layer.

3.3 Auto-Differentiation and Backward

Auto-differentiation is a prominent feature in deep learning and GraphTheta also implements auto-differentiation to simplify GNN programming by automating the backward gradient propagation during training. Like existing deep learning training systems, a primitive operation has two implementations: a forward version and a backward one. For instance, a data transformation function can implement its forward computation by a sequence of primitive operations. In this case, its backward computation can be automatically interpreted as a reverse sequence of the backward versions of these primitive operations. Assuming that $y = f(x, W)$ is a user-defined function composed of a sequence of built-in operations, GraphTheta can generate the two derivative functions automatically: $\frac{dy}{dx} = f'_r(W)$ and $\frac{dy}{dW} = f'_r(x)$. NN-TGAR will organize all these derivative functions to implement the backward progress of the whole GNN model. Refer to Section A.2 in supplementary material for a general proof of backward computation with message propagation as well as Section A.3 for the derivatives of MPGNN.

In the task of node classification, the backward of a GNN model can be described as $K + 2$ passes of NN-TGAR, but in a reverse order. First, the differential of a loss function $\frac{dL}{d\gamma_i} = l'(y_i)$ is executed on each labeled node by a single NN-T stage. Then, the two stages NN-T and Reduce are used to compute the differential of the decoder function. In this phase, the gradients of the final embedding for each node are calculated as $\frac{dL}{d\gamma_i^k} = \frac{dL}{d\gamma_i} \cdot \text{Dec}'(\omega)$ and updated to the corresponding node. Meanwhile, the gradients of the
decoder parameters are calculated as $\frac{dL}{d\omega} = \frac{dL}{d\theta} \cdot \text{Dec}(h^K)$, and sent to the optimizer. The $K$ passes of NN-TGAR are executed backwards from the $K$-th to the first encoding layer.

Differing from forward that runs the apply part at the last step, backward executes the differential of the apply part on nodes $\{v_1, v_2, v_3\}$ in the NN-T stage, as shown in Figure 3(b). This stage calculates $\frac{dL}{d\theta}$ that will be sent to the optimizer, $\{\frac{dL}{d\theta^{k-1}}\}$, $\frac{d\text{Prop}_k}{d\theta}$ $\{i = 1, 2, 3\}$ that will be updated to the nodes as values, and $\{\frac{dL}{dM_i^k}\} | i = 1, 2 \}$ that will be consumed by the next stage $\text{NN-G}$. Besides receiving messages from the previous stage, stage $\text{NN-G}$ takes as input the values of the corresponding adjacent edges and centric nodes, and sends the result to neighbors and centric nodes, as well as the optimizer.

In $\text{NN-G}$, taking $\text{Gather}_{1,3}$ as an example, the differential of the accumulation function calculates $\frac{dL}{dM_{k-1}}$. Subsequently, the differential of the propagation function computes $\frac{d\text{Prop}_k}{dM_i}$, $\frac{d\text{Prop}_k}{dM_i}$, and $\frac{d\text{Prop}_k}{d\theta}$, all of which are multiplied by $\frac{dL}{dM_{k-1}}$ and then sent to the source node $v_3$, destination node $v_1$, and the optimizer. The $\text{Sum}$ stage receives gradient vectors computed in $\text{NN-G}$, as well as new node values computed in $\text{NN-T}$, and then element-wisely adds the gradients by node values for each of the three nodes, resulting in $\{\frac{dL}{dM_i}\} | i = 1, 2, 3\}$. These three results will be passed to the next stage. The $\text{NN-A}$ computes the gradients of the $(k-1)$-th layer embeddings $\frac{dL}{dh_{k-1}}$ as $\frac{dL}{dM_{k-1}} \cdot \text{Prop}_k(W_k)$, where the gradients of $W_k$ are calculated similarly. The optimizer invokes $\text{Reduce}$ to aggregate all the gradients of parameters (i.e., $\mu_k$, $\theta_k$, and $W_k$), which are generated in stages $\text{NN-T}$, $\text{NN-G}$, and $\text{NN-A}$ and distributed over nodes/edges, and updates the parameters with this gradient estimation.

### 4 IMPLEMENTATION

Inspired by distributed graph processing systems, we present a new GNN training system which can simultaneously support all of the three training strategies. Our new system enables deep GNN exploration without pruning graphs. Moreover, our system can balance memory footprint, time cost per epoch, and convergence speed.

Figure 4 shows the architecture of our system. It consists of five components: (i) a graph storage component with distributed partitioning and heterogeneous features and attributes management, (ii) a subgraph generation component with sampling methods, (iii) graph operators which manipulate nodes and edges, and (iv) learning core operations including neural network operators (including fully-connected layer, attention layer, batch normalization, concat, mean/attention pooling layers and etc.), typical loss functions (including softmax cross-entropy loss and binary cross-entropy loss), and optimizers (including SGD, Adam [48] and AdamW [64]).

#### 4.1 Distributed Graph Representation

Our graph programming abstraction follows the vertex-program paradigm and fits the computational pattern of GNNs which centers around nodes. In our system, the underlying graphs are stored distributively which require fast graph partitioning algorithms for efficient processing. A number of graph partitioning approaches have been proposed, such as vertex-cut [7, 27, 40], edge-cut [47, 77, 82], and hybrid-cut [11] solutions. Typical graph partitioning algorithms are included in our system to support popular graph processing methods (The effect of different partitioning methods on performance will be elaborated in Section 5.4).

To efficiently run GNN-oriented vertex programs, we propose a new distributed graph representation format which evenly distributes nodes to partitions and cuts off cross-partition edges. Similar to PowerGraph, we use master and mirror nodes, where a master node is assigned to one partition and its mirrors are created in other partitions. For each edge, our method assigns it to the partition in which its source node is a master (target nodes also can be used as the indicator). This way, any edge contains at least one master node.
The vertex-cut approach used by PowerGraph has the disadvantage of duplicating mirror nodes, resulting in memory overhead for multi-layer GNN learning. To address this problem, our method allows mirror nodes to act as placeholders and only hold node states instead of the actual values. With this strategy, our method not only reduces memory overhead but also lowers communication overhead for two reasons. On one hand, by allowing mirror nodes not to hold actual values, our method can significantly reduce memory overhead. On the other hand, at the end of each superstep, PowerGraph will synchronize all master nodes to their mirrors. Instead, our method removes this global synchronization, and only synchronizes the masters used. This means that the communication overhead of PowerGraph is an upper bound of ours. Moreover, our method can reduce the replica factor to 1 from \((N_{\text{master}} + N_{\text{mirror}})/N_{\text{master}}\), where \(N_{\text{master}}\) and \(N_{\text{mirror}}\) are the number of master and mirror nodes. With the partitioning method, the implementation of each primitive in GAS abstraction is composed of several phases. Figure 5(b) illustrates the computation of the Gather primitive.

To traverse the graph efficiently, GraphTheta organizes outgoing edges in CSR and incoming edges in CSC, and stores node and edge values separately. Our distributed graph traversal is completed in two concurrent operations: one traverses nodes with CSR and the other with CSC. For the operation with CSR, each master node sends its related values to all the mirrors and then gathers its outgoing edges with master neighbors, where the edges with mirror neighbors are directly skipped. For the operation with CSC, each mirror node gathers incoming edges with master neighbors. Mirror nodes receiving values from their corresponding masters will be passively gathered by their neighbors.

Figure 5(c) illustrates an example of traversing all outgoing edges of the master node 3 in partition 0 held by worker 0. For the operation with CSR, worker 0 processes the outgoing edge of node 3 to the master node 0. As node 1 in worker 0 is a mirror, the edge from node 3 to node 1 is skipped. Meanwhile, worker 1 sends the value of its master node 1 to the mirror in worker 0. Concurrently, for the operation with CSC, worker 0 processes the incoming edge of mirror node 1 from node 3. Herein, the mirror node 1 receives its value sent from worker 1 and will be passively gathered by node 3 in worker 0. Overall, we can see that communication only occurs between master and mirror nodes.

The abstraction can address the local message bombing problem. On one hand, for a master-mirror pair, we only need one time of message propagation of node values and the results, which can reduce the traffic load from \(O(M)\) to \(O(N)\). This is because in a partition, a master/mirror node is shared by all of its edges located in the partition, and our method that merely synchronizes master nodes with mirrors makes the traffic load proportional to \(N\) rather than \(M\). On the other hand, we only synchronize node values involved in the computation per neural network layer, during the dataflow execution of a GNN model. Moreover, we remove the implicit synchronization phase after the original Apply primitive, instead of introducing implicit master-mirror synchronization to overlap computation and communication. Additionally, heuristic graph partitioning algorithms, such as METIS [46] and Louvain [5], are also supported to adapt cluster-batched training.

### 4.2 Subgraph Training

To unify the processing of all three types of training strategies, our new system uses subgraphs as the abstraction of graph structures and the GNN operations are applied. Both mini-batch and cluster-batch train a model on the subgraphs generated from the initial batches of target nodes, whereas global-batch does on the entire graph. The size of a subgraph can vary from one node to the entire graph based on three factors, i.e., the number of GNN layers, graph topology, and community detection algorithms. The number of GNN layers determines the neighbor exploration depth and has an exponential growth of subgraph sizes. For graph topology, node degree determines the exponential factor of subgraph growth.

Figure 6 shows a mini-batch training example, where a GNN model containing two graph convolution layers is interleaved by two fully-connected layers. The right part shows a subgraph constructed from a batch of initial target nodes \(\{1, 2, 3\}\), which has one-hop neighbors \(\{4, 5, 6\}\) and two-hop neighbors \(\{7, 8\}\). The left part shows the forward and backward computation of this subgraph, with arrows indicating the propagation direction. For subgraph computation, a straightforward method is to load the subgraph structure and the related data into memory and perform matrix/tensor operations on the subgraph located on the same machine. As the memory overhead of a subgraph may exceed the memory limit, this method has an inherent limitation in generalization. Instead, GraphTheta completes the training procedure with distributed computing by only spanning the structure of the distributed subgraph. More specifically, our system distributively stores the input and intermediate data of nodes/edges following the distributed subgraph, instead of transporting and placing all of them into a shared memory space.

To construct a subgraph, our system introduces a breadth-first-search traversal operation. For each target node, this operation initializes a minimal number of layers per node, which are involved in the computation, to reduce unnecessary propagation of graph computing. Furthermore, to avoid the cost of subgraph structure construction and preserve graph access efficiency, we build a vertex-ID mapping between the subgraph and the local graph within each process/worker to reuse CSR/CSC indexing. In addition, our system has implemented a few sampling methods, including random neighbor sampling [31], which can be applied to subgraph construction.

### 4.3 Parallel Execution Model

Training subgraphs sequentially cannot fully unleash the power of a distributed system. GraphTheta is designed to concurrently train multiple subgraphs with multi-versioned parameters in a distributed environment, and support concurrent lookups and updates,
To alleviate peak memory pressure, the memory can be allocated, calculated and released immediately after use. As the allocation in the forward/backward phase, output tensors for each layer are stored in a stack of consecutive resident memory, storing raw data and tensors. For each underlying implementation details, the memory layout of nodes is structured from different batches of target nodes, especially for the same node can be incorporated in different subgraphs constructed on the fly. Instead, the global indexing of the whole graph environment to construct and release the indexing data for each graph is reused, and a private cache-friendly vertex-ID mapping is adopted to improve load balance and efficiency.

GraphView and multi-versioned parameters. To train multiple subgraphs concurrently, we design the abstraction GraphView, which maintains all key features of the underlying parallel graph storage including reused indexing, embedding lookup, and the distributed graph representation. Implemented as a light-weighted logic view of the global graph, the GraphView exposes a set of interfaces necessary to all training strategies, and allows to conveniently communicate with storage. Besides the global-, mini-, and cluster-batch, other training strategies can also be implemented based on GraphView. And training tasks with GraphViews are scheduled in parallel. That enables concurrently assigning the separated forward, backward, and aggregation phases to a training worker. Due to the varied workloads of subgraphs, a work-stealing scheduling strategy is adopted.

Figure 7 depicts the parallel batched training paradigm with graph view and multi-versioned parameter management. In the figure, ParameterManager manages multiple versions of trainable parameters. In a training step, workers can fetch parameters of a specific version from ParameterManager, and use these parameters within the step. For each worker, it computes on a local slice of the target subgraph being trained and uses a task queue (i.e., TaskQueue) to manage all the tasks assigned to itself and then execute them concurrently. At the end of a training step, parameter gradients are aggregated and sent to ParameterManager for a version update. UpdateParam performs the actual parameter update operations either in a synchronous or an asynchronous mode [35]. It needs to be stressed that we will only use synchronous training for each test throughout this paper.

5 EXPERIMENTS AND RESULTS

5.1 Experiment Setups

Datasets Information. We evaluate the performance of our system by training node classification models using 7 datasets. As shown in Table 1, the network sizes vary from small-, modest-, to large-scale. The first 6 datasets including Cora, Citeseer, Pubmed, Reddit, Amazon and ogbn-papers100M (Papers) [37] are publicly available and have only node attributes, without edge attributes or types. Cora, Citeseer, and Pubmed [76] are three citation networks with nodes representing documents and edges indicating citation relationships between documents. In the three datasets, the attribute of a node is a bag-of-words vector, which is sparse and high-dimensional, while the label of the node is the category of the corresponding document. Reddit is a post-to-post graph in which
we would like to emphasize that in these tests, our purpose is not to
produce node embeddings to a Softmax layer. In all tests, cross-
validation is used for both training and the other for testing. We train Reddit for 600 steps and Amazon for 2,750 steps with respect to mini-batch. Note that as Reddit (and Amazon) is a dense co-comment (and co-purchasing) network, the two-hop neighbors of only 1% (and 0.1%) labeled nodes almost touch 80% (and 65%) of all nodes. For cluster-batch, each training step randomly chooses 1% clusters to form the initial batch for Reddit and Amazon, where clusters are created beforehand. In addition, cluster-batch applies the same early stop strategy as mini-batch.

5.2 Accuracy Assessment

5.2.1 Evaluation on Public Datasets. On the public datasets, we train a two-layer GCN model [49] using our system and compare the performance with some well-known counterparts, including a TensorFlow-based implementation (TF-GCN) from [49], a DGL-based one (DGL), Cluster-GCN and 3 sampling-based ones: VR-GCN, GraphSAGE [31], and GraphSAIN T [105]. We exclude Fast-GCN from our comparisons because it is outperformed by Cluster-GCN and GraphSAIN T as shown in [14, 105]. Note that our system is designed for non-sampling graph learning, and our performance evaluation does not intend to compete in terms of state-of-the-art accuracy on these public datasets, but to demonstrate that our system can train as good GNN models as existing works.

Comparison with non-sampling-based methods. First, we compare our system with TF-GCN, DGL, and Cluster-GCN to demonstrate that our system can achieve highly competitive or superior generalization performance on the same datasets and models, even without using sampling. Table 2 shows the performance comparison on the three small-scale citation networks, i.e., Cora, Citeseer, and PubMed [76]. For DGL and Cluster-GCN, since they did not completely expose their hyper-parameters, this poses challenges for result reproduction. Both of them have been re-evaluated with the hidden layer size set to 16 for each dataset as described in Section 5.1. It is observed that DGL almost reproduces its original result in [88] for each dataset, but Cluster-GCN yields far inferior results to those in [14] for Cora and PubMed (note that Citeseer is not used in [14]). Both GraphTheta and TF-GCN use the same set of hyper-parameter values, including learning rate, dropout keep probability, regularization coefficient, and batch size, as proposed in [49], and train the model 300 steps for mini-batch. From Table 2, global-batch yields the best accuracy for each dataset, with the exception that its performance is neck-by-neck with that of mini-batch on Citeseer. Mini-batch also outperforms DGL, TF-GCN and Cluster-GCN for each case.

Comparison with sampling-based methods. Second, we compare our implementations (without sampling) with VR-GCN, GraphSAGE, GraphSAIN T, and Cluster-GCN on the two modest-scale datasets: Reddit and Amazon. VR-GCN adopts variance reduction, GraphSAGE uses random sampling, GraphSAIN T introduces sub-graph sampling, whereas Cluster-GCN does not use sampling same as GraphTheta. All of these GNNs train models in a mini-batched

| Name   | #Nodes | #Node attr. | #Edges | #Edge attr. | Max cluster | #clusters |
|--------|--------|-------------|--------|-------------|-------------|-----------|
| Cora   | 2.7K   | 1.4K        | 5.4K   | 0           | -           | -         |
| Citeseer | 3.3K  | 3.7K        | 4.7K   | 0           | -           | -         |
| Pubmed | 19K    | 500         | 44K    | 0           | -           | -         |
| Reddit | 233K   | 602         | 11M    | 0           | 30K         | 4.3K      |
| Amazon | 2.4M   | 100         | 61M    | 0           | 36K         | 8.5K      |
| Papers | 111M   | 128         | 1.6B   | 0           | -           | -         |
| Alipay | 1.40B  | 575         | 4.14B  | 57          | 24M         | 4.9M      |

Table 1: Information of the datasets used.

For global-batch, we set 500 epochs for Reddit at maximum, and activate early stop as long as validation accuracy reaches stable. Meanwhile, a maximum number of 750 epochs is used for Amazon. For mini-batch, each training step randomly chooses 1% labeled nodes to form the initial batch for Reddit, and 0.1% for Amazon. This results in a batch size of 1,500 for the former and 1,710 for the latter. We train Reddit for 600 steps and Amazon for 2,750 steps with respect to mini-batch. Note that as Reddit (and Amazon) is a dense co-comment (and co-purchasing) network, the two-hop neighbors of only 1% (and 0.1%) labeled nodes almost touch 80% (and 65%) of all nodes. For cluster-batch, each training step randomly chooses 1% clusters to form the initial batch for Reddit and Amazon, where clusters are created beforehand. In addition, cluster-batch applies the same early stop strategy as mini-batch.

5.2.2 Evaluation on industrial datasets. In the interviews, we equably split the data set into two parts, one for training and the other for testing.

GNN Training Settings. We employ the node classification task as the application and use the popular GCN [49] model and our in-house developed GAT-E model for performance comparison. Herein, we would like to emphasize that in these tests, our purpose is not to conclude which DL framework or GNN implementation is superior to others, but to demonstrate that GraphTheta is capable of learning GNNs as well as existing frameworks (Section C in supplementary material also gives another showcase with GAT [84] model). Note that although there are only node classification tasks in our tests, our system can support other types of tasks with moderate changes, such as revising the decoders to accommodate specific tasks and keeping the graph embedding encoder part unchanged.

Different datasets are configured to have varied hidden layer sizes. Specifically, hidden layer sizes are 16, 128, and 200 for the three citation networks (i.e., Cora, Citeseer, and PubMed), Reddit, and Amazon. Except for Amazon, all the others have their sub-sets for validation. Moreover, the latter enables dropout for each layer, whereas the former does not. In terms of global-batch, we train the model up to 1,000 epochs and select the trained model with the highest validation accuracy to test generalization performance for the latter. With respect to mini-batch, early stop is exerted once stable convergence has reached for each dataset. In all tests, cross-entropy loss is used to measure convergence, and prediction results are produced by feeding node embeddings to a Softmax layer.

For global-batch, we set 500 epochs for Reddit at maximum, and activate early stop as long as validation accuracy reaches stable. Meanwhile, a maximum number of 750 epochs is used for Amazon. For mini-batch, each training step randomly chooses 1% labeled nodes to form the initial batch for Reddit, and 0.1% for Amazon. This results in a batch size of 1,500 for the former and 1,710 for the latter. We train Reddit for 600 steps and Amazon for 2,750 steps with respect to mini-batch. Note that as Reddit (and Amazon) is a dense co-comment (and co-purchasing) network, the two-hop neighbors of only 1% (and 0.1%) labeled nodes almost touch 80% (and 65%) of all nodes. For cluster-batch, each training step randomly chooses 1% clusters to form the initial batch for Reddit and Amazon, where clusters are created beforehand. In addition, cluster-batch applies the same early stop strategy as mini-batch.

5.2 Accuracy Assessment

5.2.1 Evaluation on Public Datasets. On the public datasets, we train a two-layer GCN model [49] using our system and compare the performance with some well-known counterparts, including a TensorFlow-based implementation (TF-GCN) from [49], a DGL-based one (DGL), Cluster-GCN and 3 sampling-based ones: VR-GCN, GraphSAGE [31], and GraphSAIN T [105]. We exclude Fast-GCN from our comparisons because it is outperformed by Cluster-GCN and GraphSAIN T as shown in [14, 105]. Note that our system is designed for non-sampling graph learning, and our performance evaluation does not intend to compete in terms of state-of-the-art accuracy on these public datasets, but to demonstrate that our system can train as good GNN models as existing works.

Comparison with non-sampling-based methods. First, we compare our system with TF-GCN, DGL, and Cluster-GCN to demonstrate that our system can achieve highly competitive or superior generalization performance on the same datasets and models, even without using sampling. Table 2 shows the performance comparison on the three small-scale citation networks, i.e., Cora, Citeseer, and PubMed [76]. For DGL and Cluster-GCN, since they did not completely expose their hyper-parameters, this poses challenges for result reproduction. Both of them have been re-evaluated with the hidden layer size set to 16 for each dataset as described in Section 5.1. It is observed that DGL almost reproduces its original result in [88] for each dataset, but Cluster-GCN yields far inferior results to those in [14] for Cora and PubMed (note that Citeseer is not used in [14]). Both GraphTheta and TF-GCN use the same set of hyper-parameter values, including learning rate, dropout keep probability, regularization coefficient, and batch size, as proposed in [49], and train the model 300 steps for mini-batch. From Table 2, global-batch yields the best accuracy for each dataset, with the exception that its performance is neck-by-neck with that of mini-batch on Citeseer. Mini-batch also outperforms DGL, TF-GCN and Cluster-GCN for each case.

Comparison with sampling-based methods. Second, we compare our implementations (without sampling) with VR-GCN, GraphSAGE, GraphSAIN T, and Cluster-GCN on the two modest-scale datasets: Reddit and Amazon. VR-GCN adopts variance reduction, GraphSAGE uses random sampling, GraphSAIN T introduces sub-graph sampling, whereas Cluster-GCN does not use sampling same as GraphTheta. All of these GNNs train models in a mini-batched
128 for Reddit and 200 for Amazon as stated in Section 5.1. It needs to be stressed that since GraphSAINT offers three random samplers, we have assessed all of the samplers and drawn the conclusion that sampling-based training methods are not always better than non-sampling-based ones.

Table 2: Comparison to counterparts without sampling.

| Dataset | Accuracy in Test Set (%) |
|---------|--------------------------|
|         | GCN w/ GB | GCN w/ MB | GCN On DGL | GCN On TF | Cluster-GCN |
| Cora    | 82.70     | 82.40     | 80.50      | 81.50     | 70.47       |
| Citeseer| 71.90     | 71.90     | 70.36      | 70.30     | 59.43       |
| Pubmed  | 80.00     | 79.50     | 79.06      | 79.00     | 75.14       |

Table 3: Comparison to counterparts with sampling.

| Dataset | Accuracy in Test Set (%) |
|---------|--------------------------|
|         | GB: global-batch, MB: mini-batch, and CB: cluster-batch |
| Reddit  | 96.44 95.87 96.44 96.44 |
| Amazon  | 89.77 88.34 26 |

Table 4: Accuracy comparison on Alipay dataset.

| Strategies | Performance in Test Set (%) | Time (h) |
|------------|-----------------------------|----------|
|            | F1 Score | AUC      |          |
| Global-batch | 12.18 | 87.64 | 30 |
| Mini-batch  | 13.33  | 88.12  | 36 |
| Cluster-batch  | 13.51 | 88.36  | 26 |

In terms of both F1 score and accuracy. In terms of convergence speed in the same distributed environment, we can observe that cluster-batch converges the fastest. The second fastest method is global-batch, and the third one is mini-batch. Specifically, the overall training time of 1,024 workers is 30 hours for global-batch, 36 hours for mini-batch, and 26 hours for cluster-batch, and the peak memory footprint per worker is 12 GB, 5 GB, and 6 GB respectively.

5.3 Scalability Assessment

5.3.1 Comparison of Different Training Strategies. We evaluate the capability of strong scaling to big graphs of GraphTheta with respect to the number of workers using Alipay dataset. Each worker is equipped with one computing thread and runs in a Linux CPU Docker. Due to the large memory footprint of Alipay dataset, we start with 256 workers and use this performance as the baseline.

Figures 8(a), 8(b) and 8(c) illustrate the speedup results in the function of the number of workers for global-batch, cluster-batch, and mini-batch respectively. From the figures, we can see that each of the three training strategies can scale to 1,024 workers, and the speedups for the forward, backward, and full training steps are consistent in terms of the scaling size for each training strategy. This can be explained by our observation that the neural network functions are usually compute-intensive and thus lead to better computation and communication overlap, significantly lowering the impact of communication. By investigating the speedups of the three training strategies, it can be observed that global-batch has the best scalability, followed by cluster-batch and mini-batch in decreasing order. This is because (i) global-batch has relatively balanced workloads among workers, since all nodes in the graph participate in the computation simultaneously, and (ii) cluster-batch has better data locality among distributed machines, resulting in less inter-machine communication than mini-batch.

In the following, we will analyze the speedups and parallel efficiency gained as the number of workers varies. For global-batch, by increasing the number of workers from 256 to 512, the forward runs 1.66× faster, the backward 1.75× faster, and the full training step 1.72× faster. Furthermore, when increasing the number of workers further to 1,024, the speedups become 2.81, 3.21, and 3.09 times respectively. In terms of parallel efficiency, the forward achieves 83% (and 70%), backward 87% (and 80%), and full training steps 86% (and 77%) by using 512 workers (and 1,024 workers), respectively.

In terms of cluster-batch, when moving from 256 to 512 (and 1,024) workers, the speedup is 1.70 (and 1.75) for the forward, 1.74 (and 2.45) for the backward, and 1.52 (and 1.80) for the full training step. In this case, the corresponding parallel efficiency becomes 85% (and 44%) for the forward, 87% (and 61%) for the backward, and 76% (and 45%) for the full training step.

With respect to mini-batch, compared to the baseline performance at 256 workers, the speedup and parallel efficiency at 512 workers is 1.83 and 91% for the forward, 1.77 and 88% for the backward, and 1.71 and 86% for the full training step, respectively. Meanwhile, the corresponding values at 1,024 workers are 1.84 and 46% for the forward, 2.85 and 71% for the backward, and 2.23 and 56% for the full training step, respectively.

5.3.2 Comparison with DistDGL. In this section, we will compare with DistDGL [112, 113] in terms of scalability, as well as the best
For scalability assessment, we set to use 4 cores for each worker/trainer and keep the overall batch size of value 24K unchanged (i.e. the batch size per worker/trainer times the number of workers/trainers is equal to 24K). Considering that DistDGL runs a distributed graph storage server within each machine, the computational resources of each machine have to be split among the servers and all trainers running in the same machine. Herein, we set the number of threads per server to be \(\text{max}\{16, 64 - 4 \times p\}\), where \(p\) represents the number of trainers per machine.

Figure 9(a) shows the scalability of GraphTheta, which demonstrates consistent scalability for all models. Specifically, each model scales to 64 workers with a slight performance degradation at 128 workers. Section D.1 in supplementary material shows that DistDGL does not scale at all for each model. We analyze that this phenomenon is caused by the increasing amount of redundant computation among batches as the number of trainers grows. Specifically, as more trainers are launched, the batch size per trainer is reduced proportionally since the overall batch size is invariant in our synchronous training. In this case, some neighbor nodes shared by the input target nodes of a large batch have to be replicated within the set of small batches. These replicated nodes will be computed multiple times between batches and thus lower training speed and scalability. However, GraphTheta does not suffer from this problem, because the subgraph constructed from the target nodes is independent of the number of workers and its overall amount of computation does not grow as more workers join. In addition, our hybrid-parallel execution can further reduce runtime by using more workers.

Besides scalability, we also compare with the best performance of DistDGL gained on this cluster of virtual machines. As suggested by one DistDGL developer, we launch only one trainer in each machine but tune the number of threads assigned to the trainer and the distributed server. Refer to Section D.2 in the supplementary material for more details. Figure 9(b) illustrates the speedup of GraphTheta over DistDGL for each model. More specifically, the speedup is 1.09, 1.53, 2.02, and 1.81 for the 2-layered, 3-layered, 4-layered, and 5-layered model, respectively.

5.3.3 Comparison with GraphLearn. In this section, we will compare with GraphLearn in terms of scalability on the same CPU cluster as in Section 5.3.2. We use four GCNs of the different number of layers (i.e. 2, 3, and 4) on the Reddit and Papers datasets, but exclude the 5-layered GCN, because GraphLearn does not work on it even with a small sampling setting to be described below. Our synchronous distributed training keeps the overall batch size of 24K for Reddit and 12K for Papers, and sets hidden layer sizes to 128. Same as in Section 5.3.2, GraphTheta still configures 4 cores for each worker. Figure 9(c) illustrates the scalability of GraphTheta on Papers, where our performance gets better as the number \(w\) of workers increases for the 3- and 4-layered GCNs, but encounters marginal degradation from 64 to 128 workers for the 2-layered one.

GraphLearn provides multiple sampling strategies, among which the “full” sampling strategy is chosen. This strategy returns all neighbors if the number of neighbors for a hop is less than or equal to the threshold \(\text{nbr\_num}\) and otherwise truncates by \(\text{nbr\_num}\). We have attempted to disable sampling for GraphLearn by setting \(\text{nbr\_num}\) to a large value under the “full” strategy but encountered socket errors. Hence, we keep sampling enabled and employ two sampling settings concerning \(\text{nbr\_num}\). One configures \(\text{nbr\_num}\) to relatively small values 10, 5, and 3 (following ByteGNN [111]), which correspond to hops from 1 to 3, while the other sets \(\text{nbr\_num}\) to relatively large values 20, 10, and 2.

In GraphLearn, each graph server creates a thread pool of 32 threads by default to process concurrent sampling queries. We have tried to reduce the thread pool size to some small values but encountered errors. Therefore, we keep using 32 concurrent thread settings throughout our tests. Table 5 gives the performance with \(w\) varying from 8 to 32. It needs to be stressed that launching \(w > 32\)
Table 5: Average runtimes per mini-batch for GraphLearn.

| Sampling setting | GCNs | Reddit (#workers) | Papers (#workers) | 8 | 16 | 32 | 8 | 16 | 32 |
|------------------|------|-------------------|------------------|---|---|----|---|---|----|
| 2-layer          | 13.6 | 4.4               | 1.6              | 3.6| 1.2| 0.5|   |    |     |
| 3-layer          | 614.8| 153.6             | 40.8             | 109.8|30.0 |8.5|   |    |     |
| 4-layer          | 8863.5|2301.1             | 565.6            | 1759.4|433.8 |112.8|   |    |     |

| Sampling setting | GCNs | Reddit (#workers) | Papers (#workers) | 25 | 10 | 10.2 | 8 | 16 | 32 |
|------------------|------|-------------------|------------------|----|----|------|---|---|----|
| 2-layer          | 32.2 | 11.1              | 5.7              | 9.9| 3.7| 1.8  |   |    |     |
| 3-layer          | –    | –                 | –                | 984.7|253.2 |72.3|   |    |     |
| 4-layer          | –    | –                 | –                | –  | –  | –    |   |    |     |

- indicates the results are unavailable caused by socket errors.

workers will result in socket errors. From the table, for every combination of the dataset and sampling setting, it is surprising to observe super-linear speedups along with a doubling increase of $w$, with the performance of 8 workers as the baseline. After communicating with a GraphLearn developer, we analyze that this phenomenon is due to the following two reasons. One is that the thread pool can run 32 concurrent threads at maximum. Therefore, the processing capability of concurrent queries could ideally increase at the same rate as $w$ grows. The other is that multiple workers concurrently run on the same machine and thereby can benefit from faster intra-machine communication than inter-machine one.

Even though sampling reduces the computational overhead of GraphLearn and thus results in unfair comparisons with sampling-free systems, we still compare with GraphLearn in terms of the best performance on the same machines as done in Section 5.3.2. Our evaluation shows that on Reddit, GraphTheta yields a speedup of 2.61 (and 30.56) for the 3-layered (and 4-layered) GCN. As for Papers, GraphTheta does not demonstrate speed advantages to GraphLearn, as expected, but the latter completely failed in the 4-layered GCN when using sampling setting 25.10.10.2. In this sense, GraphTheta is still superior. Finally, note that GraphLearn uses a Python user-defined function (UDF) to create sparse tensors for a mini-batch. Hence, we expect that its performance could get further improved if the UDF was programmed in some more efficient language.

5.4 Effect of Graph Partitioning Methods

In this section, we will evaluate the effect of the following two graph partitioning methods: vertex-cut and 1D-edge partition, on the execution of different training strategies using the Amazon dataset.

vertex-cut is a 2D-grid partitioning method, which targets to evenly distribute edges to all workers. Given an edge, vertex-cut determines the worker it is assigned to by computing a hash value from the source and destination nodes of the edge. 1D-edge partition aims to evenly distribute source (or destination) nodes to all workers. Given an edge, 1D-edge partition determines its destination worker by computing a hash value from the source (or destination) node. In our implementation, we compute the hash value from the source node, which is also the master node, but allow for users to configure to use the destination. vertex-cut has the advantage of addressing the highly skewed node degree distribution problem, while 1D-edge partition leads to better edge locality for the perspective of source (or destination) nodes.

In GraphTheta, we set 1D-edge partition as the default partitioning method, based on the following two considerations. On one hand, both the use of edge attributes and the computation of edge attention and embedding has been becoming more and more popular in GNNs [8, 94, 110]. With vertex-cut, the edges of a given master node are likely distributed among multiple workers. In this case, both the loading of edge attributes and the computation of edge attention and embedding will incur considerable extra communication. On the contrary, 1D-edge partition always places a master node and all of its edges together in the same worker. This way, we can load the edge attributes and compute the edge attention and embedding for the master node locally, with no need of extra communication. On the other hand, in practice, we observe that vertex-cut usually has a higher peak memory footprint than 1D-edge partition. Taking the benchmark on Amazon as an example, the peak memory of vertex-cut is about 20% larger than that of 1D-edge partition on average. Nevertheless, users can still specify their preferred partitioning method at runtime.

Figures 10(a), 10(b) and 10(c) show the normalized average runtimes of the forward, the backward and the full epoch steps, respectively in terms of global-batch, cluster-batch, and mini-batch. The normalization uses the corresponding runtime of 1D-edge partition as the baseline. From the figures, vertex-cut is superior to 1D-edge partition in terms of global-batch and mini-batch, and inferior to the latter with respect to cluster-batch. Based on these observations, we would like to make the following suggestion. If there is sufficient memory in distributed machines, we recommend users try vertex-cut first in an optimistic manner. Otherwise, 1D-edge partition is supposed to be in preference.

6 RELATED WORK

Existing GL frameworks for GNNs employ either shared-memory systems [19, 58, 88, 102] or distributed computing systems [23, 80, 85, 87, 102, 112, 113]. Most are designed for mini-batched training [106, 111, 116], while some others are for global-batch [42, 57, 65, 85, 102]. Herein, we will mainly analyze the pros and cons of those works that are capable of processing billion-scale networks.

PinSage [102] trains GNNs with sampling enabled and uses MapReduce to implement non-sampling GNN inference. However, MapReduce incurs considerable manual GNN translation overhead. Specifically, users have to re-write and translate GNN models into MapReduce procedures and make endeavors to guarantee the correctness of this translation. AGL [106] extends the idea of PinSage by additionally employing MapReduce to extract training subgraphs offline and save them on disk for future use, while still using MapReduce to implement inference. AGL can consume a huge amount of disk to store subgraphs and still undergoes GNN translation burden with respect to inference.

AliGraph [116] introduces distributed graph servers to perform sampling. Compared to MapReduce-based methods, AliGraph does not require separating training from inference, thus no need of GNN translation, but demands sampling for inference. One good thing is that AliGraph generates subgraphs at runtime and discards them once they have been consumed, thus not demanding extra persistent storage of these subgraphs. Nonetheless, AliGraph does not always work well because DL workers have to pull all necessary neighbor
Figure 10: Comparison between vertex-cut and 1D-edge partitioning methods on Amazon for each training strategy.

7 CONCLUSION

In this paper, we have presented GraphTheta, a distributed and scalable GNN learning system, which is implemented based on a graph processing programming model and introduces an NN-TGAR abstraction to ease graph convolution implementations. Differing from conventional data-parallel and model-parallel paradigms, GraphTheta adopts a hybrid parallel execution, which distributively computes each batch of graph data by a group of processes/workers. Moreover, our system supports flexible training strategies and performs inference through a unified implementation with training. These features distinguish our system from existing frameworks.

Extensive experiments show that GraphTheta can scale to 1,024 workers in distributed CPU virtual machines. Considering the affordable access to CPU virtual machine clusters in public clouds like Aliyun, Amazon Web Services, Google Cloud, and Azure, our system shows promising potential to achieve low-cost graph learning on big graphs. Moreover, our system demonstrates comparable or better generalization than the well-known GCN implementations on a diverse set of networks. Compared with DistDGL, GraphTheta yields much better scalability and runs up to 2.02× faster in terms of the best performance on the Reddit dataset using the same machine. On the same dataset, meanwhile, GraphTheta can run up to 2.61× (and 30.56×) faster than GraphLearn when training the 3-layered (and 4-layered) GCN. In particular, GraphTheta can scale well with good parallel efficiency on the large edge-attributed Alipay dataset of 1.4 billion nodes and 4.1 billion edges. Meanwhile, cluster-batch is observed capable of obtaining the best generalization and the fastest convergence speed on Alipay. This suggests that it is worthwhile of developing new graph learning systems to allow for exploring a diversity of training strategies to meet the different requirements of various applications.

It is worth mentioning that the internal code name of GraphTheta is GeoLearning (Graph Extended and Accelerated Learning) and it has been deployed in production and used by several businesses in mobile Alipay App including Zhima Credit and risk management. Currently, GraphTheta merely runs on CPUs and the performance of its implementation based on fine-grained vertex programs still has a large room for improvement. Therefore, the performance improvement of GraphTheta in favor of GPUs is part of our future work. Nonetheless, through this system, we have proven the feasibility of developing a distributed graph learning system based on graph processing. This characteristic enables our system to execute graph processing and learning procedures in the same program, thereby opening up more opportunities to address graph intelligence problems involving diverse graph processing techniques (e.g. graph mining [13, 78, 79]).
We thank Kefeng Deng and other members of our GeaLearn team in Ant Group, including Wei Qin, Zhiqiang Guo, Yice Luo, Peng Du, Yue Jin and Xiabao Wu, for their contributions to this project. We would like to thank Yannmin Fang for building the Alipay dataset. Here is the Data Protection Statement: 1. The data used in this research does not involve any Personal Identifiable Information (PII). 2. The data used in this research were all processed by data abstraction and data encryption, and the researchers were unable to restore the original data. 3. Sufficient data protection was carried out during the process of experiments to prevent the data leakage and the data was destroyed after the experiments were finished. 4. The data is only used for academic research and sampled from the original data, therefore it does not represent any real business situation in Ant Financial Services Group.

ACKNOWLEDGMENTS

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A system for large-scale machine learning. In 12th (USENIX) Symposium on Operating Systems Design and Implementation (OSDI) 16:265–283.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khadem. 2018. Learning to Represent Programs with Graphs. In International Conference on Learning Representations.

[3] Adam Auten, Matthew Tomei, and Rakesh Kumar. 2020. Hardware acceleration of graph neural networks. In 2020 57th ACM/IEEE Design Automation Conference (DAC) IEEE, 1–6.

[4] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In International Conference on Learning Representations.

[5] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment P10008 (Oct 2008), 1–12.

[6] Aleksandar Bojchevski, Johannes Kleiber, Bryan Perozzi, Amol Kapoor, Martin Blais, Benedek Rümmer, Michael Lukasz, and Stephan Günnemann. 2020. Scaling graph neural networks with approximate pagerank. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2464–2473.

[7] Umit V Catalyurek and Cevdet Aykanat. 1996. Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. In International Workshop on Parallel Algorithms for Irregularly Structured Problems. Springer, 75–86.

[8] Snehu Chaudhuri, Varun Mithal, Gungor Polatkan, and Rohan Ramanathan. 2021. An Attentive Survival Conference. Machine Learning: Transactions on Intelligent Systems and Technology (TIST) 12, 5 (2021), 1–32.

[9] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. Fast: fast learning with graph convolutional networks via importance sampling. arXiv preprint arXiv:1801.10247 (2018).

[10] Ming Chen, Zhenwei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. 2020. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In 33rd Conference on Neural Information Processing Systems (NIPS) 2019.

[11] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Cupach, Garth A Gibbons, and Eric F Xing. 2013. More effective distributed mVIA via a stale synchronous parallel parameter server. Advances in neural information processing systems 26 (2013).

[12] Weihua Hu, Matthias Fey, Honygu Ren, Malo Nakata, Yuxiao Dong, and Jure Leskovec. 2021. OGB-LG: A Large-Scale Challenge for Machine Learning on Graphs. In Thirty-Fifth Conference on Neural Information Processing Systems (NIPS) 2021.

REFERENCES
Husong Liu, Shengliang Lu, Xinyu Chen, and Bingsheng He. 2020. G3: When
Kubernetes. 2022. https://kubernetes.io.
George Karypis and Vipin Kumar. 1999. Parallel multilevel series k-way parti-
Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2015.
Zhihao Jia, Sina Lin, Rex Ying, Jiaxuan You, Jure Leskovec, and Alex Aiken. 2020.
Nilesh Jain, Guangdeng Liao, and Theodore L. Willke. 2013. Graphbuilder:
Qi Liu, Maximilian Nickel, and Douwe Kiela. 2019. Hyperbolic Graph Neural Net-
Renjie Liao, Marc Brockschmidt, Daniel Tarlow, Alexander L Gaunt, Raquel
Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with
Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic opti-
Nilesh Jain, Guangdeng Liao, and Theodore L. Willke. 2013. Graphbuilder: scalable graph etl framework. In First international workshop on graph data management experiences and systems: 1–6.
Yuguang Ji, Mingyang Yin, Hongxia Yang, Jingen Zhou, Vincent W Zheng, Chuan Shi, and Yuan Fang. 2020. Accelerating Large-Scale Heterogeneous Interaction Graph Embedding Learning via Importance Sampling. ACM Transactions on Knowledge Discovery from Data (TKDD) 15, 1 (2020), 1–23.
Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Im-
Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In Proceedings of the 2017 Symposium on Cloud Computing. 445–451.
George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM J. Scientific Computing 20, 1 (1998), 359–392.
Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016).
Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Koyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation. 427–444.
Kubernetes. 2022. https://kubernetes.io.
Kai Lei, Meng Qin, Bo Bai, Gong Zhang, and Minzhuo Huang. 2018. Adaptive sampling towards fast graph representation learning. In Advances in neural information processing systems. 4558–4567.
Nilesh Jain, Guangdeng Liao, and Theodore L. Willke. 2013. Graphbuilder:
Yingzhe Li, Sheng Liu, Binbin Liu, Xinbo Gao, Fei-Chen Chen, and Yuan Su. 2020. G3: When Graph Neural Networks Meet Parallel graph Processing Systems on GPUs. Proc. VLDB Endow. 13, (2020), 2813–2816.
Qi Liu, Maximilian Nickel, and Douwe Kiela. 2019. Hyperbolic Graph Neural Net-
[37] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In Proceedings of the 34th International Conference on Neural Information Processing Systems. Article 1855, 16 pages.
[38] Chengyong Han, Hang Liu, Mengxing Liu, Yongchao Liu, Changhai He, Kang Cheng, Jinlei Jiang, Yongyuei Wu, and Shuaizhen Leon Song. 2022. TeGraph: A Neural General-Purpose Temporal Graph Computing Engine. In 2022 IEEE 26th International Conference on Data Engineering (ICDE). IEEE, 578–592.
[39] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive
[40] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In Proceedings of the 34th International Conference on Neural Information Processing Systems. Article 1855, 16 pages.
[38] Chengyong Han, Hang Liu, Mengxing Liu, Yongchao Liu, Changhai He, Kang Cheng, Jinlei Jiang, Yongyuei Wu, and Shuaizhen Leon Song. 2022. TeGraph: A Neural General-Purpose Temporal Graph Computing Engine. In 2022 IEEE 26th International Conference on Data Engineering (ICDE). IEEE, 578–592.
[39] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive
towards fast graph representation learning. In Advances in neural information processing systems. 4558–4567.
[37] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In Proceedings of the 34th International Conference on Neural Information Processing Systems. Article 1855, 16 pages.
A THEORETICAL ANALYSIS

A.1 Equivalence Relation

Indeed, the propagation on graphs and sparse matrix multiplication is equivalent in the forward. The general convolutional operation on graph $G$ can be defined [49][18] as,

$$x * G \mathcal{K} = U((U^T x) \otimes (U^T \mathcal{K})), \quad (2)$$

where $U = [u_0, u_1, \ldots, u_{N-1}] \in \mathbb{R}^{N \times N}$ is the complete set of orthonormal eigenvectors of normalized graph Laplacian $L$, $\otimes$ is the element-wise Hadamard product, $x \in \mathbb{R}^{N \times 1}$ is a 1-dimension signal on graph $G$ and $\mathcal{K}$ is the convolutional kernel on the graph.

The graph Laplacian defined as $L = I_N - \frac{1}{\lambda_M}AD^{-1/2}$, where $D = \text{diag}(d_1, d_2, \ldots, d_N)$ and $d_i = \sum_{j=0}^{N} A(i,j)$. Equation (2) can be approximated by a truncated expansion with $K$-order Chebyshev polynomials as follows (more details are given in [33][49]).

$$x * G \mathcal{K} \approx \sum_{k=0}^{K} \theta_k T_k(L)x = \theta_0 T_0(L)x + \theta_1 T_1(L)x + \cdots + \theta_K T_K(L)x,$$

where the Chebyshev polynomials are recursively defined by,

$$T_0(L) = 2; \quad T_1(L) = L; \quad T_k(L) = 2LT_{k-1}(L) - T_{k-2}(L) \quad (4a)$$

and $T_0(L) = I_N$, $T_1(L) = L$, $\Lambda = \frac{2L}{\lambda_M} - I_N$, $\lambda_M$ is the the largest eigenvalue of $L$ and $\theta_0, \theta_1, \ldots, \theta_k$ are learnable parameters, which parameterize the convolutional kernel $\mathcal{K}$. Substituting (4a) into (3), and considering $\Lambda M$ as a learnable parameters, the truncated expansion can be rewritten as,

$$\sum_{k=0}^{K} \eta_k L^k x = \eta_0 I_N x + \eta_1 Lx + \cdots + \eta_K L^K x,$$

where $\eta_0, \eta_1, \ldots, \eta_K$ are learnable parameters. This polynomials can also be written in a recursive form of

$$\sum_{k=0}^{K} \eta_k L^k x = \sum_{k=0}^{K} \eta_k' T_k'(x) \quad (6a)$$

with $T_0'(x) = L_0 = 2L$, $T_1'(x) = L_1 = L$, $T_k'(x) = x$ and

$$\eta_k' = \eta_k / \eta_{k-1}; \quad \eta_0' = \eta_0. \quad (6b)$$

and $\eta_0', \eta_1', \ldots, \eta_K'$ are also considered as a series of learnable parameters. We generalize it to high-dimensional signal $X \in \mathbb{R}^{N \times d_k}$ as follows: each item in (6a) can be written as $H_k = T_k'(X)W_k$. Recalling (6b), the convolutional operation on a graph with a high-dimensional signal can be simplified as $K$-order polynomials and defined as,

$$x * G \mathcal{K} \approx \sum_{k=0}^{K} H_k, \quad \text{with} \quad H_k = LH_{k-1}W_k, H_0 = XW_0. \quad (7)$$

where $H_k \in \mathbb{R}^{N \times d_k}$, and $W_k \in \mathbb{R}^{d_k \times d_k}$, which are also learnable and parameterize the convolutional kernel. Based on the matrix multiplication rule, the $i$-th line $h_i^k$ in $H_k$ can be written as,

$$h_i^k = \sum_{j=1}^{N} L(i,j)(h_j^{k-1}W_k). \quad (8)$$

Thus, we can translate convolutional operation into $K$ rounds of propagation and aggregation on graphs. Specifically, at round $k$, the projection function is $h_i^k = h_i^{k-1}W_k$, the propagation function is $m_j^{k-1} = m_j^{k-1}W_k$. The aggregation function is $\sum_{j=1}^{N} n_j^{k-1}$. In other words, $h_j^{k-1}$ (or $h_i^k$) is the $j$-th line (or $i$-th line) of $H_k$ (or $H_{k-1}$) and also the output embedding of $v_i$ (or the input embedding of $v_j$). Each node propagates its message $n_i^k$ to its neighbors, and also aggregates the received messages sent from neighbors by summing up the values weighted by the corresponding Laplacian weights.

A.2 Backwards of GNN

A general GNN model can be abstracted into $K$ combinations of individual stage and conjunction stage. Each node on the graph can be treated as a data node and transformed separately in a separated stage, which can be written as,

$$n_i^k = f_k(h_i^{k-1}|W_k). \quad (9)$$

But the conjunction stage is related to the node itself and its neighbors, without loss of generality, it can be written as

$$h_i^k = g_k(A(i,1)n_1^k, A(i,2)n_2^k, \ldots, A(i,N)n_N^k|\mu_k). \quad (10)$$

where $A(i,j)$ is the element of the adjacency matrix and is equal to the weight of $e_{i,j}$ (refer to the first paragraph in 2). The forward formula (10) can be implemented by message passing, as $n_i^k$ is propagated from $v_j$ along the edges like $e_{i,j}$ to its neighbor $v_i$. The final summarized loss is related to the final embedding of all the nodes, so can be written as,

$$L = (h_0^k, h_1^k, \ldots, h_N^k). \quad (11)$$

According to the multi-variable chain rule, the derivative of previous embeddings of a certain node is

$$\frac{\partial L}{\partial n_i^k} = \frac{\partial L}{\partial n_i^{k+1}} \frac{\partial n_i^{k+1}}{\partial n_i^k} + \frac{\partial L}{\partial n_i^{k+2}} \frac{\partial n_i^{k+2}}{\partial n_i^k} + \cdots + \frac{\partial L}{\partial n_i^{k+1}} \frac{\partial n_i^{k+1}}{\partial n_i^k}. \quad (12)$$

Thus, (12) can be rewritten as,

$$\frac{\partial L}{\partial n_i^k} = \sum_{j \in N(i)} a_{i,j} \frac{\partial L}{\partial n_j^k} \frac{\partial n_j^k}{\partial n_i^k}. \quad (13)$$

So the backward of a conjunction stage also can be calculated by a message passing, where each node (i) broadcasts the current gradient $\frac{\partial L}{\partial h_i^k}$ to its neighbors along edges $e_{i,j}$; (ii) calculates the differential $\frac{\partial h_j^k}{\partial h_i^k}$ on each edge $e_{i,j}$, multiplies the received gradient and edge weight $a_{i,j}$, and sends the results (vectors) to the destination node $v_j$; (iii) sums up the received derivative vectors, and obtains the gradient $\frac{\partial L}{\partial n_i^k}$. Meanwhile, the forward and backward of each stage are similar to the normal neural network. The above derivation can expand to the edge-attributed graph.
A.3 Derivation in MPGNN

The previous section gives the derivatives for a general GNN model, and this part describes the derivation for the MPGNN framework. As in Algorithm 1, a MPGNN framework contains $K$ (or clusters). The black nodes in Figure A1(c) are in community $A$. The previous section gives the derivatives for a general GNN model, and the black nodes in Figure A1(d) belong to community $A$. Comparisons imply the necessity to design a new GNN learning system and cluster-batched training strategies in Table A1. These comparisons adopt the combination of (1a) and (1b). If the gradients of the $k$-th layer node embeddings $\partial L/\partial \mu_k$ are given, the gradients of $(k-1)$-th layer node embeddings and the corresponding parameters are computed as,

\[
\frac{\partial L}{\partial \mu_k} = \sum_{i=1}^{N} \frac{\partial L}{\partial \epsilon_i} \frac{\partial \epsilon_i}{\partial \mu_k},
\]

\[
\frac{\partial L}{\partial \mu_k} = \sum_{i=1}^{N} \frac{\partial L}{\partial \epsilon_i} \frac{\partial \epsilon_i}{\partial M_k} \frac{\partial M_k}{\partial \mu_k},
\]

\[
\frac{\partial L}{\partial \theta_k} = \sum_{i=1}^{N} \frac{\partial L}{\partial \epsilon_i} \frac{\partial \epsilon_i}{\partial \theta_k},
\]

\[
\frac{\partial L}{\partial \theta_k} = \sum_{i=1}^{N} \frac{\partial L}{\partial \epsilon_i} \frac{\partial \epsilon_i}{\partial M_k} \frac{\partial M_k}{\partial \mu_k} \frac{\partial \mu_k}{\partial \theta_k},
\]

\[
\frac{\partial L}{\partial \theta_k} = \sum_{i=1}^{N} \frac{\partial L}{\partial \epsilon_i} \frac{\partial \epsilon_i}{\partial \theta_k} + \frac{\partial L}{\partial \theta_k},
\]

\[
\frac{\partial L}{\partial \theta_k} = \sum_{i=1}^{N} \frac{\partial L}{\partial \epsilon_i} \frac{\partial \epsilon_i}{\partial \theta_k} + \frac{\partial L}{\partial \theta_k}.
\]

B TRAINING STRATEGY EXAMPLES

Figures A1(c) and A1(d) illustrate an example of cluster-batched computation, where the graph is partitioned into two communities (or clusters). The black nodes in Figure A1(c) are in community $A$ and the black nodes in Figure A1(d) belong to community $B$. The green or blue nodes in both graphs are the 1-hop or 2-hop boundaries of a community. To train a 2-layered GNN model, we can select community $A$ as the first batch and community $B$ as the second. In this case, to achieve more flexibility, our system allows users to configure whether to get their 1-hop or 2-hop boundary neighbors involved in the embedding computation of the black nodes in communities $A$ and $B$. By default, this feature is disabled, i.e., only nodes in the communities participate in the computation as done in Cluster-GCN.

We compare the pros and cons of global-batched, mini-batched, and cluster-batched training strategies in Table A1. These comparisons imply the necessity to design a new GNN learning system that enables the exploration of different training strategies and a solution to address the limitations of existing architectures.

C ACCURACY COMPARISON WITH GAT MODEL

In the main text, we have used the popular GCN algorithm for performance comparison between our system and existing DL frameworks. As stated in the main text, the purpose of these tests is to show that our system can learn GNNs as well as existing frameworks. Herein, we use the GAT model as another example algorithm and three publicly available datasets, i.e. Cora, Citeseer and Pubmed, to compare the performance between our system and DGL for readers’ information. Table A2 shows the accuracy comparison with the GAT model, where it can be observed that our system yields comparable accuracy with DGL.
Figure A1: An example of a two-hop node classification: (a) and (b) are of mini-batch, (c) and (d) are of cluster-batch. The black solid points are the target nodes to generate mini-batches. The green and blue solid nodes are the one-hop and two-hop neighbors of their corresponding target nodes. The hollow nodes are ignored during the embedding computation. The nodes with circles are the shared neighbors between mini-batches.

Table A1: Comparison of three GNN training strategies.

| Strategies   | Advantages                                      | Disadvantages                          |
|--------------|-------------------------------------------------|----------------------------------------|
| Global-batch | • No redundant calculation;                       | • The highest cost in one step.        |
|              | • Stable training convergence.                   |                                        |
| Mini-batch   | • Friendly for parallel computing;               | • Redundant calculation among batches;  |
|              | • Easy to implement on modern DL frameworks.     | • Exponential complexity with depth;    |
| Cluster-batch| • Advantages of mini-batch but with less redundant calculation. | • Power law graph challenge.           |
|              |                                                 | • Limited support for graphs without obvious community structures; |
|              |                                                 | • Instable learning speed and imbalanced batch size. |

Table A2: Accuracy comparison with GAT model.

| Dataset       | Accuracy in Test Set (%) |                  |                  |
|---------------|--------------------------|------------------|------------------|
|               | GraphTheta w/GB          | GraphTheta w/MB  | DGL               |
| Cora          | 81.1                     | 80.0             | 81.4             |
| Citeseer      | 71.2                     | 70.8             | 72.6             |
| Pubmed        | 78.7                     | 78.6             | 78.0             |

GB: global-batch, MB: mini-batch.

Table A3: Runtimes (in seconds) of DistDGL in the function of the number of trainers.

| #Trainers | Number of layers |
|-----------|------------------|
|           | 2                | 3                | 4                | 5                |
| 8         | 22.031           | 55.775           | 88.941           | 123.507          |
| 16        | 22.793           | 60.959           | 99.537           | 137.468          |
| 32        | 25.458           | 73.233           | 124.891          | 174.956          |
| 64        | 30.967           | 105.259          | Socket Error     | Socket Error     |
| 128       | 41.301           | Socket Error     | Socket Error     | Socket Error     |

D.3 Ablation Study

We have conducted an ablation study of GraphTheta on the runtime percentage of each stage of the 2-layered GCN on the ogbn-papers100M dataset. In this test, we use the mini-batched training
strategy and launch 128 workers in our cluster. The training procedure of a mini-batch step is split into six phases for break-down analysis, i.e. preparation, the forward computation of graph convolution (GCNConv) layer 0, the forward computation of GCNConv layer 1, the backward computation of GCNConv layer 0, the backward computation of GCNConv layer 1, and parameter updates for the whole model. Figure A3 illustrates the runtime percentage of each phase. From the figure, we can see that the forward and backward computation of GCNConv layer 0 dominates the overall runtime with a total percentage of up to 76.28%. This is consistent with our expectation, as this layer processes the most nodes and edges.