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 $\mathcal{G}$ can be defined [49][18] as,

$$\boldsymbol{x} *_{\mathcal{G}} \mathcal{K} = \boldsymbol{U}((\boldsymbol{U}^T\boldsymbol{x}) \odot (\boldsymbol{U}^T\mathcal{K})), \tag{2}$$

where $U = [u_0, u_1, ..., u_{N-1}] \in \mathbb{R}^{\mathrm{N}\times\mathrm{N}}$ is the complete set of orthonormal eigenvectors of normalized graph Laplacian $\boldsymbol{L}$, $\odot$ is the element-wise Hadamard product, $\boldsymbol{x} \in \mathbb{R}^{\mathrm{N}\times 1}$ is the 1-dimension signal on graph $\mathcal{G}$ and $\mathcal{K}$ is the convolutional kernel on the graph. The graph Laplacian defined as $\boldsymbol{L} = \boldsymbol{I}_N - \boldsymbol{D}^{(-1/2)}\boldsymbol{A}\boldsymbol{D}^{(-1/2)}$), where $\boldsymbol{D} = diag\{d_1, d_2, ..., d_N\}$ 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]).

$$\begin{aligned}\boldsymbol{x} *_{\mathcal{G}} \mathcal{K} \approx & \sum_{k=0}^{K} \theta_k T_k(\hat{\boldsymbol{L}})\boldsymbol{x} \\ = & \theta_0 T_0(\hat{\boldsymbol{L}})\boldsymbol{x} + \theta_1 T_1(\hat{\boldsymbol{L}})\boldsymbol{x} + \cdots + \theta_K T_K(\hat{\boldsymbol{L}})\boldsymbol{x},\end{aligned} \tag{3}$$

where the Chebyshev polynomials are recursively defined by,

$$T_k(\hat{\boldsymbol{L}}) = 2\hat{\boldsymbol{L}}T_{k-1}(\hat{\boldsymbol{L}}) - T_{k-2}(\hat{\boldsymbol{L}}) \tag{4a}$$

$$with \quad T_0(\hat{\boldsymbol{L}}) = \boldsymbol{I}_N,\ T_1(\hat{\boldsymbol{L}}) = \hat{\boldsymbol{L}}, \tag{4b}$$

where $\hat{\boldsymbol{L}} = 2\boldsymbol{L}/\lambda_M - \boldsymbol{I}_N$, λ_M is the the largest eigenvalue of $\boldsymbol{L}$, and $\theta_0, \theta_1, \cdots, \theta_K$ are the Chebyshev coefficients, which parameterize the convolutional kernel $\mathcal{K}$. Substituting (4a) into (3), and considering λ_M as a learnable parameters, the truncated expansion can be rewritten as,

$$\sum_{k=0}^{K} \eta_k \boldsymbol{L}^k\boldsymbol{x} = \eta_0\boldsymbol{I}_N\boldsymbol{x} + \eta_1\boldsymbol{L}\boldsymbol{x} + \cdots + \eta_K\boldsymbol{L}^K\boldsymbol{x}, \tag{5}$$

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

$$\sum_{k=0}^{K} \eta_k \boldsymbol{L}^k\boldsymbol{x} = \sum_{k=0}^{K} \eta'_k T'_k(\boldsymbol{x}) \tag{6a}$$

$$with \quad T'_k(\boldsymbol{x}) = \boldsymbol{L}\eta'_{k-1}T'_{k-1}(\boldsymbol{x}),\ T'_0(x) = x \tag{6b}$$

$$and \quad \eta'_k = \eta_k/\eta_{k-1},\ \eta'_0 = \eta_0. \tag{6c}$$

and $\eta'_0, \eta'_1, \cdots, \eta'_K$ are also considered as a series of learnable parameters. We generalize it to high-dimensional signal $\boldsymbol{X} \in \mathbb{R}^{\mathrm{N}\times \mathrm{d_I}}$ as follows: each item in (6a) can be written as $\boldsymbol{H}_k = T'_k(\boldsymbol{X})\boldsymbol{W}_k$. Recalling (6b), the convolutional operation on a graph with a high-dimensional signal can be simplified as K-order polynomials and defined as,

$$\boldsymbol{x} *_{\mathcal{G}} \mathcal{K} \approx \sum_{k=0}^{K} \boldsymbol{H}_k,\ with\ \boldsymbol{H}_k = \boldsymbol{L}\boldsymbol{H}_{k-1}\boldsymbol{W}_k, \boldsymbol{H}_0 = \boldsymbol{X}\boldsymbol{W}_0, \tag{7}$$

where $\boldsymbol{H}_k \in \mathbb{R}^{\mathrm{N}\times \mathrm{d}}$, and $\boldsymbol{W}_k \in \mathbb{R}^{\mathrm{d}\times \mathrm{d}}$, which are also learnable and parameterize the convolutional kernel. Based on the matrix multiplication rule, the *i-th* line $\boldsymbol{h}_i^k$ in $\boldsymbol{H}_k$ can be written as,

$$\boldsymbol{h}_i^k = \sum_{j=1}^{N} \boldsymbol{L}(i,j)(\boldsymbol{h}_j^{k-1}\boldsymbol{W}_k). \tag{8}$$

Thus, we can translate convolutional operation into K rounds of propagation and aggregation on graphs. Specifically, at round k, the projection function is $\boldsymbol{n}_i^k = \boldsymbol{h}_i^{k-1}\boldsymbol{W}_k$, the propagation function is $\boldsymbol{m}_{j\to i}^k = \boldsymbol{L}(i,j)\boldsymbol{n}_i^k$, the aggregation function is $\boldsymbol{h}_i^k = \sum_{j\in N(i)} \boldsymbol{m}_{j\to i}^k$. In other words, $\boldsymbol{h}_j^{k-1}$ (or $\boldsymbol{h}_i^k$) is the *j-th* line (or *i-th* line) of $\boldsymbol{H}_k$(or $\boldsymbol{H}_{k-1}$) and also the output embedding of v_i (or the input embedding of v_j). Each node propagates its message $\boldsymbol{n}_i^k$ 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,

$$\boldsymbol{n}_i^k = f_k(\boldsymbol{h}_i^{k-1}|\boldsymbol{W}_k). \tag{9}$$

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

$$\boldsymbol{h}_i^k = g_k(A(i,1)\boldsymbol{n}_1^k, A(i,2)\boldsymbol{n}_2^k, \cdots, A(i,N)\boldsymbol{n}_N^k|\boldsymbol{\mu}_k), \tag{10}$$

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_j^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,

$$L = l(\boldsymbol{h}_0^K, \boldsymbol{h}_1^K, \cdots, \boldsymbol{h}_N^K). \tag{11}$$

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

$$\frac{\partial L}{\partial \boldsymbol{n}_i^k} = \frac{\partial L}{\partial \boldsymbol{h}_1^k}a_{1,i}\frac{\partial \boldsymbol{h}_1^k}{\partial \boldsymbol{n}_i^k} + \frac{\partial L}{\partial \boldsymbol{h}_2^k}a_{2,i}\frac{\partial \boldsymbol{h}_2^k}{\partial \boldsymbol{n}_i^k} + \cdots + \frac{\partial L}{\partial \boldsymbol{h}_N^k}a_{N,i}\frac{\partial \boldsymbol{h}_N^k}{\partial \boldsymbol{n}_i^k}. \tag{12}$$

Thus, (12) can be rewritten as,

$$\frac{\partial L}{\partial \boldsymbol{n}_i^k} = \sum_{j\in N_{in}(i)} a_{j,i}\frac{\partial L}{\partial \boldsymbol{h}_j^k}\frac{\partial \boldsymbol{h}_j^k}{\partial \boldsymbol{n}_i^k} \tag{13}$$

So the backward of a conjunction stage also can be calculated by a message passing, where each node (i) broadcasts the current gradient $\partial L/\partial \boldsymbol{h}_j^k$ to its neighbors along edges $e_{j,i}$; (ii) calculates the differential $\partial \boldsymbol{h}_j^k/\partial \boldsymbol{n}_i^k$ on each edge $e_{j,i}$, multiplies the received gradient and edge weight $a_{j,i}$, and sends the results (vectors) to the destination node v_i; (iii) sums up the received derivative vectors, and obtains the gradient $\partial L/\partial \boldsymbol{n}_i^k$. 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 passes' procedure of "projection-propagation-aggregation", where the projection function (Line 6) can be considered as the implementation of an individual stage, while the propagation function (Line 7) and aggregation function for the conjunction stage. Aggregation function adopts the combination of (1a) and (1b). If the gradients of the k-th layer node embeddings $\partial L/\partial \boldsymbol{h}_i^k$ are given, the gradients of $(k-1)$-th layer node embeddings and the corresponding parameters are computed as,

$$\frac{\partial L}{\partial \boldsymbol{\mu}_k^{(2)}} = \sum_{i=1}^{N} \frac{\partial L}{\partial \boldsymbol{h}_i^k} \frac{\partial \boldsymbol{h}_i^k}{\partial \boldsymbol{\mu}_k^{(2)}}, \tag{14}$$

$$\frac{\partial L}{\partial \boldsymbol{\mu}_k^{(1)}} = \sum_{i=1}^{N} \frac{\partial L}{\partial \boldsymbol{h}_i^k} \frac{\partial \boldsymbol{h}_i^k}{\partial \boldsymbol{M}_i^k} \frac{\partial \boldsymbol{M}_i^k}{\partial \boldsymbol{\mu}_k^{(1)}}, \tag{15}$$

$$\frac{\partial L}{\partial \boldsymbol{m}_{j\to i}^k} = \frac{\partial L}{\partial \boldsymbol{h}_i^k} \frac{\partial \boldsymbol{h}_i^k}{\partial \boldsymbol{M}_i^k} \frac{\partial \boldsymbol{M}_i^k}{\partial \boldsymbol{m}_{j\to i}^k}, \tag{16}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_k} = \sum_{i=1}^{N} \sum_{j\in N_O(i)} \frac{\partial L}{\partial \boldsymbol{m}_{j\to i}^k} \frac{\partial \boldsymbol{m}_{j\to i}^k}{\partial \theta_k}, \tag{17}$$

$$\begin{aligned} \frac{\partial L}{\partial \boldsymbol{n}_i^k} = & \sum_{j\in N_O(i)} \frac{\partial L}{\partial \boldsymbol{m}_{j\to i}^k} \frac{\partial \boldsymbol{m}_{j\to i}^k}{\partial \boldsymbol{n}_i^k} + \\ & \sum_{j\in N_I(i)} \frac{\partial L}{\partial \boldsymbol{m}_{i\to j}^k} \frac{\partial \boldsymbol{m}_{i\to j}^k}{\partial \boldsymbol{n}_i^k} + \frac{\partial L}{\partial \boldsymbol{h}_i^k} \frac{\partial Apy_k}{\partial \boldsymbol{n}_i^k}, \end{aligned} \tag{18}$$

$$\frac{\partial L}{\partial \boldsymbol{W}_k} = \sum_{i=1}^{N} \frac{\partial L}{\partial \boldsymbol{n}_i^k} \frac{\partial \boldsymbol{n}_i^k}{\partial \boldsymbol{W}_k}, \tag{19}$$

$$\frac{\partial L}{\partial \boldsymbol{h}_i^{k-1}} = \frac{\partial L}{\partial \boldsymbol{n}_i^k} \frac{\partial \boldsymbol{n}_i^k}{\partial \boldsymbol{h}_i^{k-1}}. \tag{20}$$

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.

D PERFORMANCE ASSESSMENT

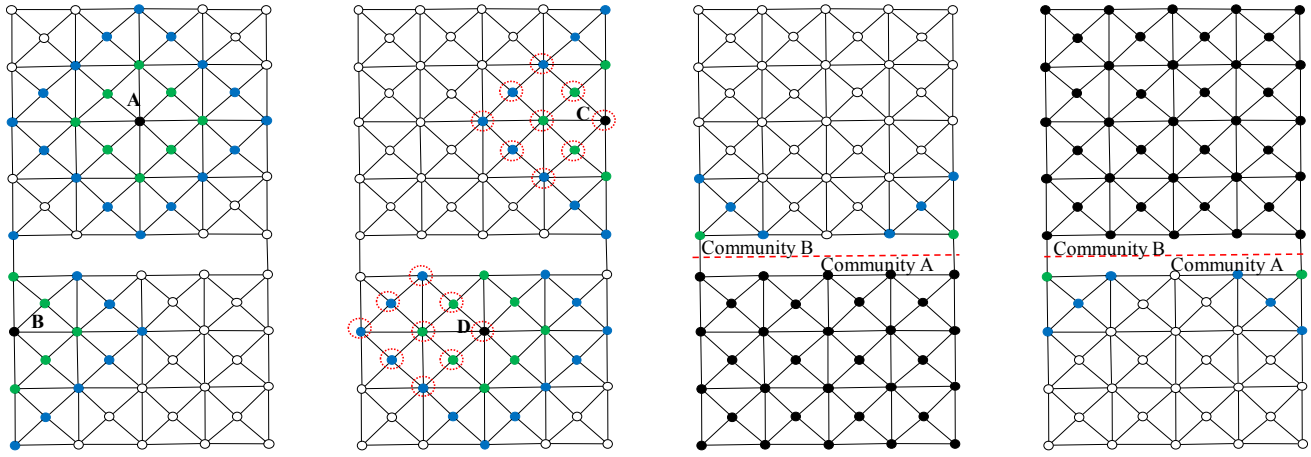
D.1 Scalability Assessment

In this section, we show the detailed scalability of DistDGL [112, 113] by means of training four GCN models of a different number of layers on the Reddit dataset. Same with the tests in the main text, the number of layers ranges from 2 to 5, and node sampling is disabled for a fair comparison. Table A3 shows the runtime per mini-batch in the function of the number of trainers. From the table, the runtime increases as the number of trainers grows, indicating that DistDGL does not scale at all concerning number of trainers. Further, except for the 2-layered model, all other models encountered socket errors when the number of trainers is large. Specifically, the 3-layered model failed in 128 trainers, while both the 4-layered and 5-layered models started to fail from 64 trainers.

We would like to express that we have excluded Dorylus [80] and ROC [42] from this comparison for the following reasons. Firstly, Dorylus is based on AWS Lambda and cannot run in our Kubernetes cluster. Secondly, Dorylus performs graph operations in distributed graph servers and conducts neural network operations by Lambda threads. However, it still needs to load all neighbors of given target nodes into Lambda threads. This is the same as DistDGL. Thirdly, Dorylus runs slower than DGL (non-sampling) on Reddit as shown in its paper. Finally, ROC requires each graph server to load the entire graph into memory and thus does not match the objective of our tests for fully distributed graphs.

D.2 Parameter Tuning

As mentioned in the main text, we launch only one trainer in each machine and tune the number of threads assigned to the trainer and the distributed server. In this test, in each machine, the number of threads assigned to the server is calculated as $64-p$, where p denotes the number of threads used by the trainer. Figure A2 illustrates the performance changes along with the tuning of value p for each GCN model. According to our tuning, the best performance of DistDGL is gained by setting $p = 44$ for the 2-layered model, $p = 48$ for the 3-layered model, $p = 36$ for the 4-layered model, and $p = 58$ for the 5-layered model.



(a) Nodes *A* and *B* form a batch (b) Nodes *C* and *D* form a batch (c) Community *A* forms a batch (d) Community *B* forms a batch

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; • Stable training convergence.	• The highest cost in one step.
Mini-batch	• Friendly for parallel computing; • Easy to implement on modern DL frameworks.	• Redundant calculation among batches; • Exponential complexity with depth; • Power law graph challenge.
Cluster-batch	• Advantages of mini-batch but with less redundant calculation.	• Limited support for graphs without obvious community structures; • Instable learning speed and imbalanced batch size.

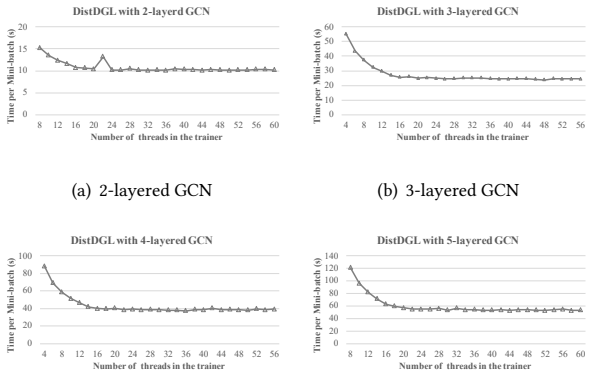
Table A2: Accuracy comparison with GAT model.

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



(a) 2-layered GCN (b) 3-layered GCN

(c) 4-layered GCN (d) 5-layered GCN

Figure A2: Parameter tuning of DistDGL for 4 GCN models on Reddit.

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

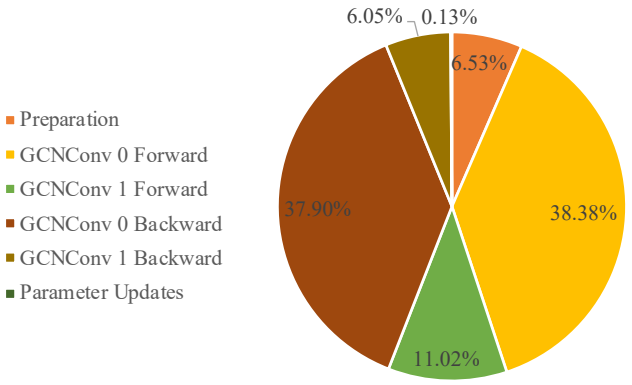


Figure A3: Ablation study of GraphTheta on the runtime percentage of each stage of 2-layered GCN on ogbn-papers100M.

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.