NeutronStar: Distributed GNN Training with Hybrid Dependency Management

ABSTRACT

GNN's training needs to resolve issues of vertex dependencies, i.e., each vertex representation's update depends on its neighbors. Existing distributed GNN systems adopt either a dependencies-cached approach or a dependencies-communicated approach. Having made intensive experiments and analysis, we find that a decision to choose one or the other approach for the best performance is determined by a set of factors, including graph inputs, model configurations, and an underlying computing cluster environment. If various GNN trainings are supported solely by one approach, the performance results are often suboptimal. We study related factors for each GNN training before its execution to choose the best-fit approach accordingly. We propose a hybrid dependency-handling approach that adaptively takes the merits of the two approaches at runtime. Based on the hybrid approach, we further develop a distributed GNN training system called NeutronStar, which makes high performance GNN trainings in an automatic way. NeutronStar is also empowered by effective optimizations in CPU-GPU computation and data processing. Our experimental results on 16-node Aliyun cluster demonstrate that NeutronStar achieves 1.81X-14.25X speedup over existing GNN systems including DistDGL and ROC.

1 INTRODUCTION

Deep Neural Networks (DNNs) have been used for problem solving in different application areas, such as image classifications, semantic segmentations, and machine translations. Recently, research efforts have been made to extend DNN to graph data analytics [5, 17, 24, 26, 28, 41, 45, 47, 48], known as Graph Neural Networks (GNNs). The goal is to use GNN solve a wide variety of application problems that can be abstracted as graph models, including social networks, physical systems, biological networks, and knowledge graphs. Considering huge sizes of generated graphs from applications, we must conduct massive parallel and distributed computation to process GNNs.

In a DNN training, there are no dependencies among input samples. The training samples can be randomly partitioned into disjoint subsets for parallel training (known as data parallel) without information data exchanges among data samples. The partial gradients retrieved from backward computation on each subset are aggregated and used to update the parameters for next iteration's training. In contrast, GNN trainings exhibit complex interdependencies among data samples (i.e., graph vertices), leading to significantly different execution patterns from that of DNN trainings. In a GNN training, dependencies among vertex samples are determined by a given input graph. The **vertex dependencies** are defined as *the direct in-neighbors of a vertex*. Each vertex gathers and aggregates the feature vectors from its in-neighbours, and the aggregated vector is used to generate a new vertex representation based on a parameterized function (an NN model). The model parameters are updated through backward propagation. As it may require k graph operations to gather k-hop away neighbors' information, there could be k NN models to train. The graph operations and NN operations are alternatively executed, making the NN's forward/backward propagations mixed with graph operations. A key challenge for a distributed GNN system design is on how to effectively handle vertex dependencies during NN's forward and backward propagations.

A number of distributed GNN systems have been recently proposed to support large-scale GNN trainings, e.g., AliGraph [53], Euler [7], AGL [51], and DistDGL [52]. These distributed GNN systems are designed on top of existing distributed machine learning systems such as Tensorflow [1] and distributed PyTorch [23] for the benefits of highly optimized NN execution. They rely on a **Dependencies Cached** (DepCache) approach, working together with data parallel model for GNN's training. Specifically, for a k-layer GNN model, a subset of nodes (including node features and node labels) and their dependent k-hop neighbors' features are assigned to a worker in advance for NN operations. As depicted in Figure 1(b), the graph propagation and NN's forward/backward computation only occur within each worker. By the DepCache approach, no representation/gradient exchange occurs among workers, which matches the data parallel model. In this way, large-scale GNN training can be supported with the help of existing DNN systems. However, the DepCache approach may result in serious redundant computations causing degraded performance. As mentioned above, each node representation's update depends on the chained computation results of its k-hop neighbors. There could exist a big overlap between two nodes' k-hop neighbors so that the chained computations of different nodes are overly redundant, particularly for large graphs from various applications.

On the other hand, some other systems [14, 29] adopt a **Dependencies Communicated** (DepComm) approach. Instead of caching dependencies locally, these systems let each vertex gather its neighbors' representations from remote workers via communication. The computation results of upstream dependent vertices can be transferred to and shared by its downstream computations. As shown in Figure 1(c), the graph propagation (including both the forward and backward propagation) occurs across workers and leads to cross-worker communications. There is no redundant computation but at a cost of necessary communication. In an advanced distributed platform equipped with GPU accelerators, the communication overhead could seriously hurt performance as well.

Having made intensive experiments and analysis, we find that a decision to choose one or the other approach for the best performance is determined by a set of factors, including graph inputs, model configurations, and environment configurations. In a GNN training system based on a single approach, the execution performance is often suboptimal. Supported by high-end computing resources, GNN training with factors of rare dependencies among vertices and a wide hidden layer size would benefit from DepCache

because the redundant computation does not affect performance much. On the other hand, supported by a high network bandwidth communication facility, GNN training with factors of rich dependencies among vertices and a small hidden layer size would gain performance from DepComm because the performance is not much sensitive to the additional communication cost.

In this paper, we propose a hybrid dependency management approach as shown in Figure 1(d). Unlike existing distributed GNN systems [7, 14, 29, 38, 51–53] that adopt either DepCache or DepComm, NeuTronStar combines DepCache and DepComm to achieve the maximum performance. The key novelty of our system lies on that, with respect to a given graph and a given hardware environment, we estimate the DepCache cost and the DepComm cost for each dependent neighbor and choose the most cost-efficient processing strategy for different dependencies. Based on the hybrid dependency management approach, we propose NeutronStar, a distributed GNN training system with GPU-acceleration. NeutronStar implements a flexible automatic differentiation framework that supports automatic gradients computation according to the chain rule and cross-worker gradients backward propagation. Instead of manually implementing the operators for cross-worker forward/backward computation, we adopt a vertex-cut master-mirror framework to decouple the cross-worker graph operations from the in-worker NN operations, so that the hardware-optimized operator implementations from existing DNN systems can be employed for local NN operations. Currently, NeutronStar relies on the GPU-optimized operator implementations in PyTorch autograd library [32], and it has the flexibility to be extended to work with other hardware-optimized operators by using TensorFlow [1] or MindSpore [20]. In addition, NeutronStar implements chunk-based partitioning, ring-based task scheduling, communication and computation tasks overlapping, and lock-free parallel message queuing for efficient heterogeneous CPU-GPU processing and communication. High-level python APIs are also provided for ease of use.

We have made the following contributions in this paper.

- **Providing insights into the two existing approaches.** We conduct a comprehensive study on the performance merits and limits of the two GNN training approaches (DepCache and DepComm), and identify the key tradeoff between redundant computations and a large amount of communications.
- **Proposing a hybrid dependency management framework.** We quantify the cost of redundant computation and that of cross-worker communication, and propose a hybrid dependency management approach, aiming to maximize the performance of distributed GNN training.
- **Delivering a distributed GNN system with GPU accelerations.** We propose a series of optimizations for CPU-GPU heterogeneous computation and highly efficient communication. By integrating all these optimizations, we design and implement NeutronStar, which also provides high-level python-based APIs.

We evaluate NeutronStar on a 16-node Aliyun GPU cluster. The experimental results show that NeuTronStar outperforms the state-of-the-art GNN systems, i.e., 1.83X-14.25X speedups over DistDGL, 1.81X-5.29X speedups over ROC, 2.03X-15.02X speedups over DepCache with NeutronStar's codebase, and 1.51X-2.21X speedup overs DepComm with NeutronStar's codebase.

2 EXECUTION PATTERNS OF GNN

In this section, we focus on the execution patterns of GNN training. We discuss the two existing distributed GNN training approaches to provide insights into the merits and limits of each. Based on the experiments and analysis, we characterize factors-dependent performance of each approach. The three major factors are graph inputs, model configurations, and execution environments.

2.1 GNN Training

Algorithm 1 GNN Training Process for a Single Epoch

Input: graph structure for training $G(V, E)$, number of layers L, node features $\{\mathbf{h}_v^{(0)} \mid v \in V\}$, node labels $\{\mathbb{L}_v \mid v \in V_{\mathbb{L}}\}$, initial model parameters $\{W1^{(l)}, W2^{(l)}\}$ for each layer l

Output: updated parameters $\{W1^{(l)}, W2^{(l)}\}$ for each layer l

Forward:

1: **for** l = 1 to L **do**
2: **for** each $v \in V$ **do**
3: **for** each $e_{u,v} \in E$ **do**
4: $\mathbf{h}_{u,v}^{(l)} = \overrightarrow{\mathcal{F}_{W1^{(l)}}}\left(\mathbf{h}_u^{(l-1)}, \mathbf{h}_v^{(l-1)}, e_{u,v}\right)$
5: $\mathbf{h}_v^{(l)} = \overrightarrow{\mathcal{F}_{W2^{(l)}}}\left(\mathbf{h}_v^{(l-1)}, \{\mathbf{h}_{u,v}^{(l)} \mid e_{u,v} \in E\}\right)$
6: **for** each $v \in V_{\mathbb{L}}$ **do**
7: $\hat{\mathbb{L}_v} = \overrightarrow{\mathcal{P}}(\mathbf{h}_v^{(L)})$
8: $loss = loss\_func(\{\hat{\mathbb{L}_v} \mid v \in V_{\mathbb{L}}\}, \{\mathbb{L}_v \mid v \in V_{\mathbb{L}}\})$

Backward:

9: **for** each $v \in V_{\mathbb{L}}$ **do**
10: $\nabla\mathbf{h}_v^{(L)} = \overleftarrow{\mathcal{P}}(loss, \mathbf{h}_v^{(L)})$
11: **for** l = L to 1 **do**
12: **for** each $v \in V$ **do**
13: $\left\{\nabla\mathbf{h}_{u,v}^{(l)} \mid e_{u,v} \in E\right\} = \overleftarrow{\mathcal{F}_{W2^{(l)}}}\left(\nabla\mathbf{h}_v^{(l)}\right)$
14: **for** each $e_{u,v} \in E$ **do**
15: $\left\{\nabla\mathbf{h}_u^{(l-1)}, \nabla\mathbf{h}_v^{(l-1)}\right\} = \overleftarrow{\mathcal{F}_{W1^{(l)}}}\left(\nabla\mathbf{h}_{u,v}^{(l)}\right)$
16: update $W1^{(l)}$ based on $\nabla W1^{(l)}$ generated in Line 13 and update $W2^{(l)}$ based on $\nabla W2^{(l)}$ generated in Line 15

Algorithm 1 shows the GNN training process for a single epoch. Given a graph $G(V, E)$ where the feature of each vertex $v \in V$ is denoted as $\mathbf{h}_v^{(0)}$, the training of GNN model first launches the **forward propagation** process to compute a representation for each vertex v by stacking multiple graph propagation layers. To obtain each vertex v' representation at layer l, i.e., $\mathbf{h}_v^{(l)}$, it requires to process v's in-neighbors through an edge-associated computation and a vertex-associated computation as shown in Line 2-5. The **edge-associated** function $\overrightarrow{\mathcal{F}_{W1^{(l)}}}$ is a parameterized function (with parameter matrix $W1^{(l)}$) which is executed on each of v's incoming edges. It takes the source u/destination v's previous layer representations $\mathbf{h}_u^{(l-1)}$/$\mathbf{h}_v^{(l-1)}$ and the edge properties $e_{u,v}$ as input to generate the edge-associated intermediate tensor $\mathbf{h}_{u,v}^{(l)}$ (Line 4). The **vertex-associated** function $\overrightarrow{\mathcal{F}_{W2^{(l)}}}$ is another parameterized function (with parameter matrix $W2^{(l)}$), which takes $\mathbf{h}_v^{(l-1)}$ and all v's incoming edge-associated tensors $\{\mathbf{h}_{u,v}^{(l)} | e_{u,v} \in E\}$ as input and outputs a new vertex representation $\mathbf{h}_v^{(l)}$ for next layer's computation (Line 5). Figure 1 (a) shows an example of the forward

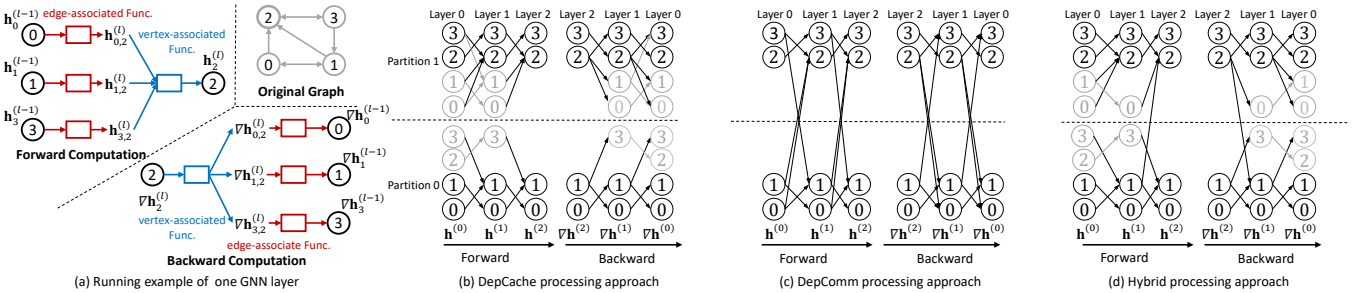

Figure 1: An example of DepCache, DepComm, **and** Hybrid **processing. (a) The original input graph and the forward and backward computation (1-layer) on vertex 2. (b) A 2-layer GCN with two workers of** DepCache. **(c) A 2-layer GCN with two workers of** DepComm. **(d) A 2-layer GCN with two workers of** Hybrid. **In (a), a red arrow/box indicates an edge-associated function, and a blue arrow/box indicates a vertex-associated function. In (b), (c), and (d), a black circle/arrow indicates locally assigned vertices/edges. A gray circle/arrow indicates cached vertices/edges.**

computation. In each layer, a vertex gathers the representations from its one-hop neighbors and applies the parameterized function on vertex or edge or both to produce the new representation. The vertex representation of the last layer $\mathbf{h}_v^{(L)}$ captures the structural information for all neighbors within L hops of v, and is used as the input for down-stream prediction task $\overrightarrow{\mathcal{P}}$ (Line 6-7). The set of predicted labels $\{\hat{\mathbb{L}_v}|v \in V_{\mathbb{L}}\}$ and the set of ground-truth node labels $\{\mathbb{L}_v|v \in V_{\mathbb{L}}\}$, where $V_{\mathbb{L}}$ is the set of vertices with node labels, are used to calculate the loss value (Line 8), which will be used in the following backward propagation.

The **backward propagation** works by computing the gradient of the loss function with respect to each weight by the chain rule, computing the gradient one layer at a time, iterating backward from the last prediction task to the first graph propagation layer. In the vertex-centric context, each vertex v first computes the gradient of the L-th layer's representation $\nabla\mathbf{h}_v^{(L)}$ in $\overleftarrow{\mathcal{P}}(\cdot)$ for the prediction network as shown in Line 9-10. Then the gradients are propagated back layer-by-layer mixed with the graph operations. The **backward** vertex-associated function $\overleftarrow{\mathcal{F}_{W2^{(l)}}}$ is executed on each vertex v, taking the gradient of vertex representation $\nabla\mathbf{h}_v^{(l)}$ as input to generate 1) the gradients of edge tensors $\{\nabla\mathbf{h}_{u,v}^{(l)}|e_{u,v} \in E\}$ (Line 13) and 2) the partial gradient of layer-specific parameter matrix $\nabla W2^{(l)}$. The **backward** edge-associated function $\overleftarrow{\mathcal{F}_{W1^{(l)}}}$ takes each edge tensor's gradient $\nabla\mathbf{h}_{u,v}^{(l)}$ as input and outputs partial gradients $\nabla\mathbf{h}_u^{(l-1)}$, $\nabla\mathbf{h}_v^{(l-1)}$, which are aggregated on each vertex for next layer's backward computation (Line 15). It also outputs the partial gradient of parameter matrix $\nabla W1^{(l)}$. As shown in Figure 1 (a), in the backward computation, a vertex needs to calculate the gradient of the model parameters and propagate the gradient back to its in-neighbors for the backward computation of previous layer. In this way, the partial gradient (of both parameters and representations) to be calculated by each vertex depends on its multi-hop outgoing neighbors, which is known as vertex dependencies.

2.2 Distributed GNN Training Approaches

GNN training needs to handle vertex dependencies. In a single machine-based training system, the complete graph data and model parameters can be accessed locally. However, in a distributed environment, such data are partitioned and distributed among workers, which may result in remote dependencies. As retrieving representations of dependent neighbors remotely can lead to cross-worker forward/backward propagations, a major concern of designing a distributed GNN system is on how to effectively handle dependencies at runtime. There are two main training approaches adopted by existing distributed GNN systems.

Algorithm 2 DepCache Distributed Training Approach

Input: $G(V, E)$, L, $\{\mathbf{h}_v^{(0)}|v \in V\}$, $\{\mathbb{L}_v|v \in V_{\mathbb{L}}\}$, initial model parameters $\{W1^{(l)}, W2^{(l)}\}$ for each layer l
Output: updated parameters $\{W1^{(l)}, W2^{(l)}\}$ for each layer l
1: partition V into m disjoint subsets $\{V_1, \ldots, V_m\}$ and assign V_i, $E_i = \{e_{u,v} \mid v \in V_i, e_{u,v} \in E\}$, and $\{\mathbb{L}_v \mid v \in V_i \cap V_{\mathbb{L}}\}$ to each worker i
Worker $i = 1, 2, \ldots, m$ **do in parallel**
2: $V_i^L = V_i$, $E_i^L = E_i$,
3: **for** $l = L$ to 1 **do** //retrieve dependencies in a BFS manner
4: fetch $V_i^{l-1} = \{u \mid v \in V_i^l \cup V_i, e_{u,v} \in E\}$ and $E_i^{l-1} = \{e_{u,v} \mid v \in V_i^{l-1}, e_{u,v} \in E\}$ from global storage to local
5: fetch $\{\mathbf{h}_v^0 \mid v \in V_i^0\}$ from global storage to local
6: execute Algorithm 1 layer by layer on each V_i^l and E_i^l
Synchronize between workers
7: update $\{W1^{(l)}, W2^{(l)}\}$ for each layer l

Dependencies Cached (DepCache). In DepCache, the graph data are stored in a distributed graph storage, and each worker takes a subset of vertex samples for training. The key idea of DepCache is to let each worker make the dependent neighbors readily prepared at local before training starts. As a k-layer GNN, DepCache needs to retrieve not only a vertex's direct in-neighbors but also all its $\{2, \ldots, k\}$-hop in-neighbors being cached locally. With the k-hop dependencies, DepCache performs normal forward/backward propagation layer-by-layer within a worker without any communication, so that existing data parallel training systems (e.g., TensorFlow) can be employed [1]. The details of DepCache are shown in Algorithm 2. The vertices, edges and their labels are first evenly partitioned into m disjoint subsets $\{V_1, V_2, \ldots, V_m\}$, where m is the number of workers (Line 1). On each worker, we then retrieve each subset's L-hop dependencies in a BFS-manner and fetch them to local (Line 3-4) along with its L-hop in-neighbors' initial features $\{\mathbf{h}_v^{(0)}|v \in V_i^0\}$ (Line 5). Note that, V_i^l denotes V_i's $(L-l)$-hop in-neighbors combined with V_i itself. Figure 1(b) provides an illustrative visual example. On each worker, with the initial

features of V_i's L-hop in-neighbors $\{V_i^{L-1}, \ldots, V_i^0\}$, L-hop in-edges $\{E_i^L, \ldots, E_i^1\}$, and V_i's ground-truth labels $\{\mathbb{L}_v | v \in V_i \cap V_\mathbb{L}\}$, we run the forward/backward propagation to obtain the partial gradients of L layer-specific weight matrices (Line 6). With these partial gradients obtained from multiple workers, these layer-specific weight matrices are synchronously updated in a central manner [1, 23] (Line 7). It is noticeable that, as known that a vertex or an incoming edge can be depended by multiple vertices, it could be cached by multiple vertices and replicated multiple times.

Algorithm 3 DepComm Distributed Training Approach

Input: $G(V, E)$, L, $\{\mathbf{h}_v^0 \mid v \in V\}$, $\{\mathbb{L}_v \mid v \in V_\mathbb{L}\}$, initial model parameters $\{W1^{(l)}, W2^{(l)}\}$ for each layer l
Output: updated parameters $\{W1^{(l)}, W2^{(l)}\}$ for each layer l

1: partition V into m disjoint subsets $\{V_1, \ldots, V_m\}$ and assign V_i, $E_i = \{e_{u,v} \mid v \in V_i, e_{u,v} \in E\}$, $\{\mathbf{h}_v^{(0)} \mid v \in V_i\}$, and $\{\mathbb{L}_v \mid v \in V_i \cap V_\mathbb{L}\}$ to each worker i
Worker $i = 1, 2, \ldots, m$ **do in parallel**
2: **for** l = 1 to L **synchronously do**
3: **for** each $v \in V_i$ **do**
4: **for** each $e_{u,v} \in E_i$ and $u \notin V_i$ **do**
5: fetch $\mathbf{h}_u^{(l-1)}$ from remote worker
6: *execute Line 2-5 of Algorithm 1*
7: *execute Line 6-10 of Algorithm 1*
8: **for** $l = L$ to 1 **synchronously do**
9: *execute Line 12-15 of Algorithm 1*
10: **for** each $v \in V_i$ **do**
11: **for** each $e_{u,v} \in E_i$ and $u \notin V_i$ **do**
12: send $\nabla\mathbf{h}_u^{(l)}$ to remote worker
Synchronize between workers
13: update $\{W1^{(l)}, W2^{(l)}\}$ for each layer l

Dependencies Communicated (DepComm). In contrast, the basic idea of DepComm is to communicate the data of remote vertex dependencies between workers as needed. Algorithm 3 depicts the details of DepComm. It basically follows Algorithm 1. Each worker takes in charge of a subset of vertices. During the layer-by-layer forward propagation, as it needs the dependent neighbor tensors that do not reside locally, it fetches them from remote peer workers (Line 5) and executes the edge-associated and vertex-associated computations as depicted in Algorithm 1. During the layer-by-layer backward propagation, as it generates the partial gradients of its in-neighbor u's tensor $\nabla\mathbf{h}_u^{(l)}$, it sends the gradients back to the worker where u resides (Line 12).

Comparison of the Two Approaches. The major difference between the two approaches is the way to handle massive vertex dependencies in distributed forward and backward propagation. We show an example in Figure 1. For a 2-layer GCN model, Figure 1(b) and Figure 1(c) illustrate the forward and backward computations of DepCache and DepComm, respectively. By caching the multi-hop dependencies locally in DepCache, the training process can easily adapt to existing distributed DNN training systems, e.g., TensorFlow [1]. But there exist redundant computations on the cached vertices and edges in multiple workers. In contrast, based on a vertex-centric graph processing framework, DepComm can eliminate the redundant computations, but at the cost of communication. Therefore, the tradeoff between redundant computation cost from DepCache and communication overhead from DepComm must be an important consideration in the system design.

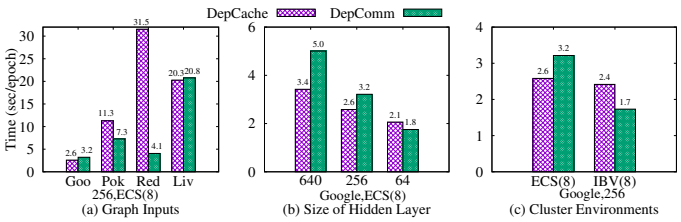

Figure 2: Performance comparison between DepComm and DepCache. (a) with different graph inputs. (b) with different hidden layer settings. (c) with different cluster environments. The results are reported in per-epoch execution time.

Existing GNN Systems Review. A number of distributed GNN systems (such as AliGraph [53], Euler [7], AGL [51], P3 [10], and DistDGL [52]) adopt DepCache approach. To reduce the redundant computation overhead, these systems all leverage a sampling approach that only samples a portion of dependent neighbors and edges for training. The sampling approach is always combined with mini-batch gradient descent training. With sacrifice of accuracy [14, 37], these systems make it possible to process massive graphs in production. In addition, DistDGL [52] and P3 [10] also provide GPU support to accelerate training. On the other hand, a set of newly emerged GNN systems (including DistGNN [29], ROC [14], DGCL [52], and Dorylus [37]) adopt DepComm approach. They do not sample and directly perform full graph computation with higher accuracy guarantee. ROC [14], DGCL [3], and Dorylus [37]) further support GPU-accelerated training. They mainly focus on the optimizations on cross-worker communications and host-GPU communications. We summarize the features of these systems in Table 1. We also list the classical single machine GNN systems, e.g., PyG [9], DGL [43], and NeuGraph [27].

2.3 Performance of the Two Approaches

We show the performance divergence between DepCache and DepComm in this section. For a fair comparison, we implement the vanilla versions of DepCache and DepComm without using advanced optimizations. We conduct the experiments on two clusters, (1) an 8-node Aliyun ECS GPU cluster (abbr. ECS) and (2) a private GPU cluster with 8 compute nodes (abbr. IBV). The detailed configurations can be found in Section 5. In the following, we test DepCache and DepComm by varying three factors, including the graph inputs, the hidden layer sizes, and the clusters.

Graph Inputs. Since the DepCache approach has more redundant computation overhead but less communication overhead than the DepComm approach, the performance of the two approaches for various graphs is hard to predict. In Figure 2(a), we use the DepCache engine and DepComm engine to test four graph inputs Google (a hyperlink graph of web), Pokec (a social network), Reddit (a post-to-post graph), and LiveJournal (a social network) on the ECS with a 2-layer GCN [17]. The hidden layer size is set to 256. For Google and LiveJournal graphs, DepCache outperforms DepComm by 1.23X and 1.03X. In contrast, DepComm outperforms DepCache by 1.54X and 7.76X on Pokec and Reddit graphs.

Table 1: Summarization of GNN systems

Systems	System Support		Training Execution Strategy			Pros	Cons
	Distributed	GPU	Dep. Process	Gradient Descent	Training Data		
PyG [9]	✗	✓	-	mini/full-batch	complete/sampling		
DGL [43]	✗	✓	-	mini/full-batch	complete/sampling	• high accuracy [14];	• hard to scale [52];
NeuGraph [27]	✗	✓	-	full-batch	complete		
AliGraph [53]	✓	✗	DepCache	mini-batch	sampling		
Euler [7]	✓	✗	DepCache	mini-batch	sampling	• parallel friendly [52];	• redundant
AGL [51]	✓	✗	DepCache	mini-batch	sampling	• compatible with	computation [14];
DistDGL [52]	✓	✓	DepCache	mini-batch	sampling	existing DNN systems [52];	
P3 [10]	✓	✓	DepCache	mini-batch	sampling		
DistGNN [29]	✓	✗	DepComm	full-batch	complete		
ROC [14]	✓	✓	DepComm	full-batch	complete	• no redundant	• heavy
DGCL [3]	✓	✓	DepComm	full-batch	complete	computation [37];	communication [22];
Dorylus [37]	✓	✓	DepComm	mini-batch	complete	• high accuracy [14];	
NeutronStar	✓	✓	Hybrid	full-batch	complete		

Model Configurations. Even on the same graph input, different model configurations introduce different amounts of redundant computation and communication overhead, giving different choices for a selection of either DepCache or DepComm. We run a 2-layer GCN for the Google graph on the ECS. Figure 2(b) shows the performance results of the two approaches with three hidden layer settings (i.e., a kind of model configuration). DepCache outperforms DepComm by 1.43X with hidden layer size 640 and 1.23X with 256, while DepComm outperforms DepCache by 1.16X with hidden layer size 64.

Cluster Environments. Another sensitive factor affecting performance is related to key physical parameters of a cluster environment. The communication bandwidth and computing power of clusters have impact on the performance of the two approaches. We run a 2-layer GCN on the Google dataset with hidden layer setting to 256. The experiments are launched on the ECS as well as on the IBV. Figure 2(c) shows the performance results on the two clusters. We can observe that DepCache outperforms DepComm by 1.23X on the ECS cluster. While on the IBV cluster, benefiting from the significant improvement of network bandwidth, DepComm outperforms DepCache by 1.41X.

Discussion on Benefit. In short, DepCache is best suitable for GNN training with rare dependencies and with a wide hidden layer size on a cluster with high-end computation resources because the redundant computation cost is lower than the communication cost. On the other hand, DepComm is best suitable for GNN training with rich dependencies and with a small hidden layer size on a cluster with high network bandwidth because the additional communication cost is lower than the redundant computation cost. In next section, we will introduce our hybrid approach by taking the merits of both existing approaches to gain the optimal performance.

3 HYBRID DEPENDENCY MANAGEMENT

In this section, we first provide a cost model to formalize the redundant computation cost from DepCache and the communication cost from DepComm. Based on this cost model, we propose a hybrid approach that employs the advantages of both DepCache and DepComm to minimize the overall cost.

As described in Algorithm 2 and Algorithm 3, in both of DepCache and DepComm, the vertices and their associated in-edges are partitioned into m disjoint subsets, i.e., $\{V_1, \ldots, V_m\}$ and $\{E_1, \ldots, E_m\}$,

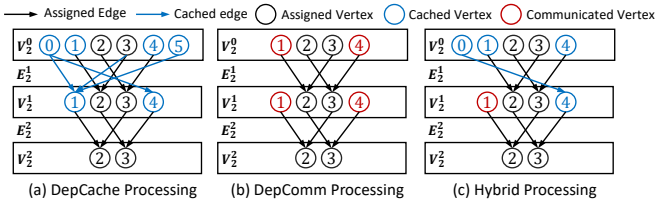

Figure 3: Dependencies handling of the three approaches.

where $E_i = \{e_{u,v} \mid v \in V_i, e_{u,v} \in E\}$. Each vertices subset V_i and their in-edges E_i are assigned to a worker i for parallel processing. The core difference between DepComm and DepCache is the way to handle V_i's dependencies. As shown in Figure 3, the computation of vertices' l-layer representations $\{\mathbf{h}_v^l \mid v \in V_i^l\}$ depends on their in-edges' l-layer representations $\{\mathbf{h}_{u,v}^l \mid e_{u,v} \in E_i^l\}$, whose computation further depends on their source nodes' $(l-1)$-layer representations $\{\mathbf{h}_u^{(l-1)} \mid u \in V_i^{l-1}\}$. We analyze the cost of acquiring each dependent neighbor in DepCache and DepComm, and based on which, we provide the cost model of Hybrid approach.

Cost of Cached Dependency. In DepCache as shown in Figure 3(a), V_i's L-hop dependent neighbors $\{V_i^{L-1}, \ldots, V_i^1, V_i^0\}$ and L-hop dependent edges $\{E_i^L, \ldots, E_i^1\}$ are cached locally. As a result, all the layer-by-layer forward computations and backward computations are occurred locally. The dependent vertex/edge might be cached by multiple workers. Each time it is replicated for caching, a redundant vertex/edge computation occurs. Furthermore, each layer has its own dependencies. For an l-layer computation, the redundant computation cost of its dependent in-neighbor u's representation $\mathbf{h}_u^{(l-1)}$ is determined by the dependent neighbors subtree rooted at u from layer $l-1$ to layer 0. For example in Figure 3(a), the computation on edge (1, 2) at layer 2 depends on node 1's layer-1 representation, then the subtree rooted at node 1 should be cached. Let $V_i^k(u)$ be u's in-neighbors in V_i^k (e.g., node 0, 3, and 5 when $k = 0$) and let $E_i^k(u)$ be u's in-edges in E_i^k (e.g., edge (0,1), (3,1), and (5,1) when $k = 1$). With respect to an l-layer computation, the redundant computation cost resulted from its dependent neighbor u can be formalized as

$$t_r^l(u) = \sum_{k=1}^{l-1} \left(\left| V_i^k(u) \setminus V_i \right| \cdot T_v + \left| E_i^k(u) \setminus E_i \right| \cdot T_e \right) \cdot d^{(k)}, \quad (1)$$

where $d^{(k)}$ is the dimension of the k-th layer, and T_v and T_e represent the per-dimensional cost of vertex tensor computation and edge tensor computation, respectively. The local set of vertices and edges (V_i and E_i) are excluded from cost accounting since they will not result in redundant computation.

Cost of Communicated Dependency. With the DepComm approach, each layer's dependent neighbor representations might be fetched from remote workers, which results in communication cost. As shown in Figure 3(b), the computation of node 2's layer-2 representation $\mathbf{h}_2^{(2)}$ needs node 1's layer-1 representation $\mathbf{h}_1^{(1)}$. With respect to an l-layer computation, the communication cost resulted from its dependent neighbor u can be formalized as

$$t_c^l(u) = T_c \cdot d^{(l-1)}, \tag{2}$$

where $d^{(l-1)}$ is the dimension of tensor $h_u^{(l-1)}$, and T_c represents the per-dimension communication overhead. The forward/backward computation is executed layer-by-layer, so the communication between workers is occurred multiple times.

A Cost Model of Hybrid Approach. Based on the redundant computation cost and communication cost of each dependent neighbor, Hybrid processing handles dependencies with discrimination, choosing a subset of dependencies for DepCache processing and another subset for DepComm processing. Let $\mathcal{D}_i^l$ be the remote dependent neighbors of layer l's vertices V_i^l, i.e., $\mathcal{D}_i^l = V_i^{l-1} \setminus V_i$. The remote dependencies of each layer l are divided into two disjoint subsets $\mathcal{R}_i^l$ and C_i^l where $\mathcal{R}_i^l$ is the DepCache set, C_i^l is the DepComm set, and $\mathcal{R}_i^l \cup C_i^l = \mathcal{D}_i$ and $\mathcal{R}_i^l \cap C_i^l = \emptyset$. Notice that, maintaining the cached dependencies and the intermediate results in DepCache also needs additional storage overhead. We should take the memory limit into consideration for designing the cost model. Therefore, the per-epoch cost for an L-layer GNN's training can be formalized as

$$\begin{aligned} \mathcal{T}_i =& \mu \cdot \sum_{l=1}^{L} \sum_{u \in \mathcal{R}_i^l} t_r^l(u) + \sum_{l=1}^{L} \sum_{u \in C_i^l} t_c^l(u), \\ \textbf{s.t.,} \quad & size\left(\{\mathcal{R}_i^1, \ldots, \mathcal{R}_i^L\}\right) \le \mathcal{S} \end{aligned} \tag{3}$$

where $0 < \mu \le 1$ is a factor to trim the redundant computation cost since the multi-hop dependencies might be overlapped with each other and the overlapped part should be only counted once. $\mathcal{S}$ is the memory limit and $size(\cdot)$ means the memory space for maintaining the multi-hop dependencies. To achieve the minimum $\mathcal{T}_i$, we need to determine the partitioning of dependencies for each layer, i.e., $\{\mathcal{R}_i^1, C_i^1\}, \ldots, \{\mathcal{R}_i^L, C_i^L\}$. The optimal problem is NP-hard as it can be reduced to a classical NP-hard problem, i.e., 0-1 integer linear planning problem [12]. Next, we propose a heuristic to find the partitioning of dependencies.

Partitioning of Dependencies. We propose a greedy-based heuristic to partition dependencies for DepCache and DepComm as shown in Algorithm 4. Firstly, we probe the environment-specific factors T_v, T_e, and T_c by executing a test training on a small graph (Line 1). A set V_{rep} is initialized to maintain the already cached multi-hop dependencies for all layers (Line 2), which is used to avoid repeated counting for the redundant computation cost. We resolve the partition of dependencies layer-by-layer from layer 1 to layer L. In each

Algorithm 4 Partition of Dependencies for Hybrid Processing

Input: vertices subset V_i, edges subset E_i, their remote dependent neighbors $\{\mathcal{D}_i^L, \ldots, \mathcal{D}_i^1\}$, and memory constraint $\mathcal{S}$
Output: partitions of dependencies $\{\mathcal{R}_i^1, C_i^1\}, \ldots, \{\mathcal{R}_i^L, C_i^L\}$
1: Probe T_v, T_e, and T_c on a small test graph
2: $V_{rep} \leftarrow \{\emptyset, \ldots, \emptyset\}$, $\{\mathcal{R}_i^1, \ldots, \mathcal{R}_i^L\} \leftarrow \{\emptyset, \ldots, \emptyset\}$
3: **for** $l = 1$ to L **do**
4: initialize a priority queue $\mathcal{Q}_{rc}^l$
5: **for** each u in $\mathcal{D}_i^l$ **do**
6: measure $t_r^l(u)$, $t_c^l(u)$ by excluding V_{rep}
7: push $\langle u, t_r^l(u)\rangle$ to $\mathcal{Q}_{rc}^l$
8: **while** $\mathcal{Q}_{rc}^l \neq \emptyset$ **do**
9: pop $\langle u, t_r^l(u)\rangle$ from $\mathcal{Q}_{rc}^l$ with minimal $t_r^l(u)$
10: re-measure $t_r^l(u)$ by excluding V_{rep}
11: **if** $t_r^l(u) < t_c^l(u)$ **then**
12: $\mathcal{R}_i^l \leftarrow \mathcal{R}_i^l \cup u$
13: $V_{rep} \leftarrow V_{rep} \cup \{V_i^{l-1}(u), \ldots, V_i^0(u)\}$
14: **if** $size(\{\mathcal{R}_i^1, \ldots, \mathcal{R}_i^L\}) > \mathcal{S}$ **then**
15: exclude u from $\mathcal{R}_i^l$, $C_i^l = \mathcal{D}_i^l \setminus \mathcal{R}_i^l$, and return
16: $C_i^l = \mathcal{D}_i^l \setminus \mathcal{R}_i^l$

layer, we first estimate each dependent neighbor u's redundant computation cost $t_r^l(u)$ and communication cost $t_c^l(u)$, where the already cached dependencies V_{rep} are excluded for measurement (Line 6). Since the communication cost $t_c^l(u)$ is consistent for each dependent neighbor u, we only need to determine the partition of dependencies based on their caching cost $t_r^l(u)$. We tend to pick the dependent neighbor that is more cache-efficient (with small $t_r^l(u)$) for caching and use a greedy-based method to realize the selection process. We rely on a priority queue $\mathcal{Q}_{rc}^l$ to maintain $\langle u, t_r^l(u)\rangle$ pairs where smaller $t_r^l(u)$ is with higher priority (Line 7). We then pop from the priority queue to retrieve the dependent neighbor u with the minimal $t_r^l(u)$ (Line 9) and re-measure its redundant computation cost by excluding the already cached multi-hop dependencies V_{rep} to avoid recounting (Line 10). If the redundant computation cost $t_r^l(u)$ is less than the communication cost $t_c^l(u)$, we add u to the cached subset $\mathcal{R}_i^l$ (Line 12) and add u's multi-hop dependencies from different layers $\{V_i^{l-1}(u), \ldots, V_i^0(u)\}$ to the replication set V_{rep} (Line 13). After determining the cached set $\mathcal{R}_i^l$ of each layer, we can infer the communicated set C_i^l of each layer by excluding $\mathcal{R}_i^l$ from $\mathcal{D}_i^l$ (Line 16). If the storage size for multiple layer's cached dependencies $size(\{\mathcal{R}_i^1, \ldots, \mathcal{R}_i^L\})$ exceeds the memory limit $\mathcal{S}$, the algorithm will terminate immediately (Line 14-15).

Graph Partitioning. We decouple the dependency partitioning and graph partitioning into separate steps, and focus on the dependency partitioning in this paper. We use chunk-based algorithm for graph partitioning as it is well-studied and widely-adopted [13, 14, 54]. Additionally, dependency partitioning is orthogonal to graph partitioning, and different graph partitioning methods [4, 14, 15, 35, 40, 50, 54] can work with our dependency partitioning method to improve performance of GNN systems.

Convergence Speed. The prior studies [14, 22, 37] have researched the convergence behavior of DepComm and DepCache processing. Relying on mini-batch training and sampling techniques to reduce

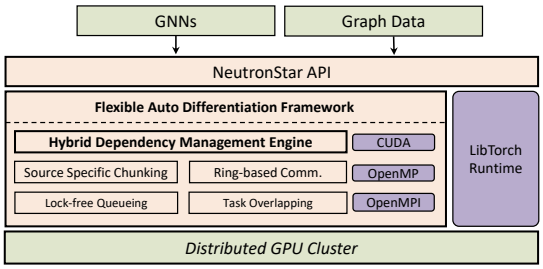

Figure 4: NeutronStar system overview.

the redundant computation, DepCache processing usually has a lower training accuracy and a slower convergence speed, because only a subset of neighbors are participated in the training. In contrast, DepComm processing can achieve a higher accuracy and a faster convergence speed due to its full-batch training and full-neighbor aggregation. Our hybrid approach can keep the high accuracy and fast convergence speed, because it retains the full-batch training and full-neighbor aggregation as DepComm processing.

4 THE NEUTRONSTAR

We design and implement NeutronStar, a distributed GPU-accelerated GNN training system, to support the hybrid dependency management. NeutronStar has also decoupled the graph operations, NN functions, and dependency management to support automated and distributed gradient backward propagation for general GNNs, and then the existing DNN auto-grad libraries, e.g., PyTorch, can be applied (Section 4.1). The design and implementation of DepComm, DepCache, and Hybrid processing are discussed in Section 4.2. The optimizations, including a source-specific subgraph chunking method and a set of computation and communication optimizations to support large scale GNN training on GPUs are described in Section 4.3. Figure 4 presents the overall structure of NeutronStar.

4.1 Flexible Auto Differentiation Framework

To implement the hybrid dependency management, NeutronStar must first implement DepCache and DepComm processing and after that it is possible to switch DepCache and DepComm in vertices. The existing distributed GNN training systems can directly use the lower-level DNN libraries to implement DepCache because all dependencies have already cached locally. While for DepComm, as the cross-worker backward propagation cannot be implemented by directly calling the functions of DNN libraries, those GNN training systems have to provide a set of operators for cross-worker computation and communication. Manually implementing those operators requires high human efforts and is also tedious and error-prone. Moreover, due to the lack of hardware-specific optimizations implemented in the DNN libraries, e.g., PyTorch on GPU [32], Tensorflow on TPU [1], and MindSpore on Ascend AI chip [20], the existing distributed GNN training systems are usually inefficient in performance or not general on hardware configurations.

To address this issue, we propose a flexible and automatic differentiation framework in NeutronStar to achieve the automated backward propagation for GNN training and benefit from the hardware-optimized operators in DNN autograd libraries [1, 20, 32]. The key idea is to decouple the dependency management from the in-worker graph operations and neural network (NN) functions.

```
GCNconv(nn.module):
   def __init__(self, in_f, out_f):
       self.W=nn.Linear(in_f, out_f)
# compute and apply the attention
   def edge_udf(edge):
       return edge.src*edge.w
# udf vertex update function
   def VertexForward(vertex, agg_msg):
       x=self.W(agg_msg)
       return nn.ReLU(x)
# forward function
   def forward(graph, f_dst):
       f_src= GetFromDepNbr(graph, f_dst)
       Edge = ScatterToEdge(graph, f_src, f_dst)
       Msg = EdgeForward(Edge, edge_udf)
       Agg_msg = GatherByDst(Msg, agg='sum')
       return VertexForward(f_dst, Agg_msg)
```

Figure 5: GCN [17] implementation with NeutronStar APIs.

GNN Programming Model and APIs. NeutronStar decouples the forward and backward execution flow into multiple-pairs of graph operations and NN functions. Each pair of operations represents one stage in the forward and backward computation. The detailed forward execution flow are listed in the following.

- **`GetFromDepNbr`** is a dependency management operation defined on a set of vertices. In each layer, it fetches the vertex data (feature and representation) from their dependent neighbors remotely in DepComm or locally in DepCache.
- **`ScatterToEdge`** is an edge message generating operation that scatters the source and destination representations to edges for the `EdgeForward` computation.
- **`EdgeForward`** is a parameterized function defined on each edge to generate an output message by combining the edge representation with the representations of source and destination.
- **`GatherByDst`** is a neighbor aggregating function to compute the neighborhood representation by aggregating incoming messages with commutative and associative operators, e.g., `min`, `max`, `sum`.
- **`VertexForward`** is a parameterized function defined on each vertex to generate new vertex representation by applying zero or several NN models on aggregated neighborhood representations.

Similar to the forward computation, NeutronStar decouples the backward computation flow into five operations.

- **`VertexBackward`** calculates the gradient for parameters and neighbor representations, according to the **`VertexForward`** function and the obtained vertex gradient from the succeeding layer.
- **`ScatterBackToEdge`** is a graph operation that scatters the source and destination gradients to edges for the backward computation.
- **`EdgeBackward`** calculates the gradient for both parameters and vertex representations, according to the **`EdgeForward`** function and the input gradients from **`ScatterBackToEdge`**.
- **`GatherBySrc`** gathers and aggregates the partial gradients from all outgoing edges to get the partial gradient of each vertex.
- **`PostToDepNbr`** manages the gradient of dependent neighbors in the backward computation flow. By using the dependencies in `GetFromDepNbr`, it sends the vertex gradients back to its origins remotely in DepComm or locally in DepCache.

Notice that these five functions are automatically generated according to the input forward computation program. NeutronStar will

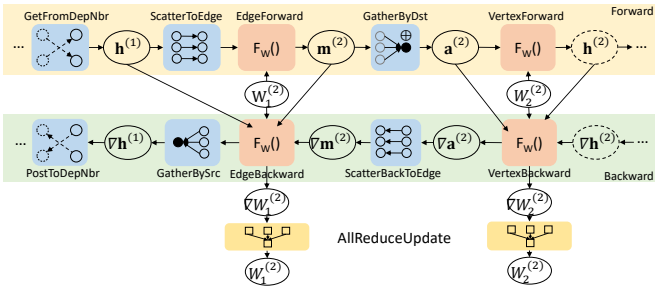

Figure 6: NeutronStar 's forward and backward execution flow for a single layer. The execution flow involves multiple vertex/edge/parameter tensors (circle), graph operations (blue box), NN operations (orange box), and all-reduce update operations (yellow box).

automatically connect and execute these functions in the backward computation without user's manual specification.

NeutronStar decouples the graph operations, i.e., `ScatterToEdge` and `GatherByDst`, and the NN functions, i.e., `EdgeForward` and `VertexForward`. Users do not need to deal with the graph operations and can focus on implementing different types of GNNs. Figure 5 shows a GCN implementation with NeutronStar. Using our high-level Python APIs, the user only needs to define the computation for each GNN layer in `EdgeForward` and `VertexForward`. The computation will take the input feature or representation from the previous layer automatically. After `GetFromDepNbr` and `ScatterToEdge`, each edge will first computes its representation based on the representation of source and the given edge weight. Next, the generated message are gathered and aggregated through `sum` operation in `GatherByDst`. The generated neighborhood representation are then fed to a fully connected layer, and passed through a non-linear ReLU activation in `VertexForward`, which generates the representation used by the next layer or the downstream tasks.

Figure 6 shows the execution flow of a general-proposed GNN model in a single layer. In forwarding, `GetFromDepNbr` fetches dependent neighbors' representations $\mathbf{h}^{(1)}$ from local cache in DepCache or remote peers in DepComm. Then, `ScatterToEdge` scatters $\mathbf{h}^{(1)}$ to edges for the edge-specific parameterized operation `EdgeForward`. The output edge tensors $\mathbf{m}^{(2)}$ are grouped by their destination and aggregated by `GatherByDst`, followed by applying `VertexForward` on the group-by result tensors $\mathbf{a}^{(2)}$. It will outputs vertex tensors $\mathbf{h}^{(2)}$ for next layer's forward computation.

NeutronStar also decouples the distributed dependency management and in-worker GNN propagation. With the `GetFromDepNbr` and `PostToDepNbr` operations, NeutronStar hides the details of dependency processing and allows the GNN propagation of each layer to be running like in a single machine. Users can build their GNN layers with the existing DNN libraries to achieve automated and efficient intra-layer gradient backward propagation on different accelerators. NeutronStar will automatically connect the backward computation flow of all layers through the dependency management operation to achieve automated cross-layer-autograd. Users no longer need to consider the complex backward propagation.

Automated Gradient Backward Propagation. The existing DNN autograd libraries need to reserve all intermediate results in the forward stage to support automated backward computation. To achieve

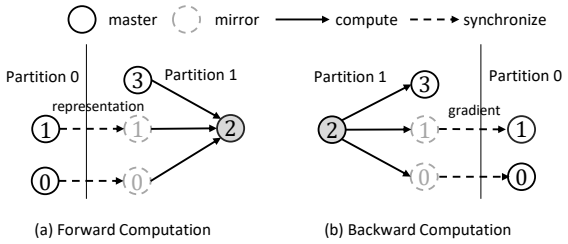

Figure 7: An example of master-mirror framework with synchronize-compute mode (forward computation) and compute-synchronize mode (backward computation).

the cross-worker auto differentiation in GNN training, we use different approaches in forward propagation and backward propagation. In the forward stage, all required dependencies from remote workers or local cache are prepared before each layer's computation; while in the backward stage, we first perform backward computation locally with the dependencies and its generated intermediate results in the forward stage, and then we pass the gradients of the dependencies back to their assigned workers for the previous layer computation. That is, we adopt a **synchronize-compute** mode for forward computation and switch to a **compute-synchronize** mode for backward computation.

As shown in Figure 6, in the backward phase, with previously obtained gradient tensors $\nabla\mathbf{h}^{(2)}$, the `VertexBackward` operation generates the vertex gradient tensors $\nabla\mathbf{a}^{(2)}$. The `ScatterBackToEdge` operation propagates $\nabla\mathbf{a}^{(2)}$ back to edges to produce the edge tensor gradients $\nabla\mathbf{m}^{(2)}$ that are used in the `EdgeBackward` function. In the `GatherBySrc` operation, the output partial vertex tensor gradients are gathered by source vertex to generate the vertex tensor gradients $\nabla\mathbf{h}^{(1)}$, and will be finally sent back to its assigned worker through the `PostToDepNbr` operation.

Parameter Update. We optimize the update of shared parameters. In the backward computation, `VertexBackward` and `EdgeBackward` will output the parameter matrix's partial gradients $\nabla W_2^{(2)}$ and $\nabla W_1^{(2)}$. The partial gradients from multiple workers will be aggregated synchronously to update parameters $W_2^{(2)}$ and $W_1^{(2)}$ in `AllReduceUpdate`. The updated parameters will be used in the next epoch. This All-Reduce update model is orthogonal to and can be replaced by the Parameter-Server model.

4.2 Hybrid Dependency Management Engine

As mentioned above, the forward-backward computation switching requires the Hybrid engine to support dependency management in two directions, i.e., `GetFromDepNbr` and `PostToDepNbr`. This requires different implementations of DepCache and DepComm. In the implementation of DepCache, as the cached neighbors usually have a large sparsity, NeutronStar manages the cached vertex data (feature, representation, and gradients) with a hash indexed and compressed 1-D Tensor.

In the implementation of DepComm, as NeutronStar adopts vertex-cut graph partitioning and relies on a master-mirror design to achieve efficient bidirectional cross-worker dependencies communication, the vertices with their in-edges are split into multiple subsets, each of which is assigned to one worker. The assigned

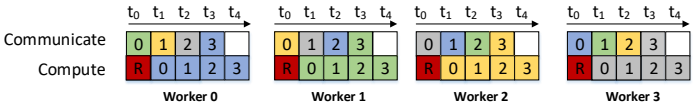

Figure 8: Ring-based task scheduling and pipelining (forward computation). Each box is a data chunk. The communicate task sends each output chunk to different destination workers at different time slots. The data chunks destined to the same worker are with the same color. The compute task tends to receive a remote data chunk and process it at different time slots. The data chunk with label 'R' will be processed with DepCache**.**

vertices will be computed as masters in the worker node, and a master node has multiple mirror nodes corresponding to its outgoing edges that locate in other workers. As shown in Figure 7, in a forward phase, each mirror (node 0 or 1) synchronizes the required vertex representation from its remote master, and performs edge and vertex computation to update the master (node 2) representation in the local worker. In a backward phase, the vertex and edge backward computations are first launched to calculate the gradients of node 2's out-neighbors (nodes 0, 1, and 3) in the local worker. Then, each mirror sends its gradient to its remote master node, where the gradients from multiple mirror nodes are aggregated.

4.3 Graph Chunking and Task Scheduling

Each worker takes charge of a subset of vertices along with their in-edges for parallel processing. Considering the massive graphs with high-dimensional node features, we do not assume all the required data can fit into GPU device memory. In each worker, the in-edges of the assigned vertices for DepComm processing are partitioned into chunks according to their source nodes. We organize each chunk of edges with CSC/CSR format [42, 54] (CSC for forward computation and CSR for backward computation) and make a chunk with all the edges whose source vertices reside on the same remote worker. By means of such chunk-based design, NeutronStar only needs to load a chunk of representation and edges from one worker at a time, which can significantly reduce the amount of data to be loaded into the GPU. Other merits are related to computing and communication efficiency. As depicted in Figure 7, with the master-mirror-based communication design, the representations (or gradients) corresponding to a source-specific chunk are from (or to) the same worker and can be packed to communicate during the forward/backward computation. Each time a worker receives a data chunk (representations or gradients), it loads the received chunk and the corresponding edges into GPU for the `EdgeForward` computation. To further utilize such a property, NeutronStar adopts several task optimizations to further hide the communication latency and overlap the computation and communication.

Ring-based Communication. To fully utilize the network bandwidth, NeutronStar organizes the worker nodes in a ring and schedules the communication tasks in a balanced cyclic manner. Each worker sends its calculated vertex representations data to remote workers and organizes its output data into m output chunks according to their destination workers, where m is the number of workers. In NeutronStar, worker i sends its j-th output chunk to worker $(i + j + 1)\%m$, i.e., the j-th output chunk from different workers will be arrived at the destination worker at different time slots, hoping that any two workers do not send messages to the same destination worker simultaneously. Based on this rule, the compute tasks receive chunks from other workers at different time slots, so as to avoid network congestion and improve communication performance. Figure 8 illustrates an example of ring-based task scheduling on a four-node cluster during forward computation. The scheduling for backward computation is similar. The difference is that, instead of communicate-compute in forward computation, we schedule the compute tasks first to calculate gradients and then schedule the communicate tasks to send gradients to remote workers.

Overlapping Communication with Computation. With the chunk-based task partitioning, we can further pipeline the execution by overlapping the communication and compute tasks. For example as shown in Figure 8, on worker 1, when communication task sends its chunk 0 generated from previous layer's computation, the compute task with DepCache is scheduled to utilize the idle GPU resource. When communication task sends its chunk 1, the compute task with DepComm is scheduled to process the received message chunk 0 from worker 3. In this way, both GPU computation resources and network bandwidth can be well utilized.

Lock-free Parallel Message Enqueuing. A common issue in multi-thread message passing system is on its efficient implementation of concurrent queue. In NeutronStar, multiple threads are invoked to execute a task. The generated messages from multiple threads are enqueued and compacted before being sent. To deal with a write conflict problem, existing graph computation systems use mutex locks [8, 54]. Locks create bottlenecks causing high latency in a system if many parallel threads are involved. For GNN training, the communicated messages have a relatively regular pattern, which can be used to improve the message passing performance. As discussed in Section 4.3, in each layer's training, there is a task sending messages to a specific worker. We can create a fixed-size message-sending buffer and pre-define the message locations for different destination nodes, where we construct a write position index (array) by parsing the destination node ids. When writing messages with multiple threads, we let each thread write message to the position according to the message's destination node id, so the write conflict problem is completely avoided.

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setup

Environments. Our experiments are conducted on Aliyun ECS-cluster with 16 GPU nodes. Each node (ecs.gn6i-c16g1.4xlarge instance) is equiped with 16 vCPUs, 62GB DRAM, and 1 NVIDIA Tesla T4 GPU, running Ubuntu 18.04 LTS OS. The network bandwidth is 6 Gbps/s. IBV-cluster is an 8-node private GPU cluster used for the comparison in Section 2. Each node contains one Intel Xeon E5-2680 CPU, 128GB DRAM, and one NVIDIA Tesla V100 GPU running Ubuntu 18.04. The nodes are connected with 100Gb/s EDR Infiniband. Libraries CUDA 10.0, OpenMPI-3.0.2, PyTorch v1.5 backend [33], and cuDNN 7.0 are used in both clusters.

Datasets and GNN Algorithms. Major parameters of graph datasets that are used in our experiments are presented in Table 2: Google [21] is a web graph. Pokec [36], LiveJournal [2], Com-Orkut [49] and Twitter [19] are four social networks. Reddit [11] is a post-to-post graph. Wiki-Link [18] records user interactions

Table 2: Dataset description

Dataset	|V|	|E|	ftr. dim	#𝕃	avg. deg	hid. dim
Google	0.87M	5.1M	512	16	5.86	256
Pokec	1.6M	30M	512	16	18.75	256
LiveJournal	4.8M	68M	320	16	14.12	160
Reddit	0.23M	114M	602	41	487	256
Orkut	3.1M	117M	320	20	38.1	160
Wiki-link	12M	378M	256	16	31.12	128
Twitter	42M	1.5B	52	16	70.5	32
Cora	2.7K	5.4K	1433	7	-	128
Citeseer	3.3K	4.7K	3307	6	-	128
Pubmed	20K	44K	500	3	-	128

on Wikipedia. In addition, Cora, Citeseer and Pubmed [34], three citation networks that need smaller memory spaces to store are used for the single-node evaluation. The average degrees, the node feature dimensions, and the number of labels of the graphs are also given in Table 2. For graphs without vertex features, we use randomly generated features. We use three popular GNN models, GCN [17], GIN [48], and GAT [41]. All of them are in a 2-layer structure. The hidden layer dimensions of datasets are listed in Table 2.

The Systems for Comparisons. In our performance evaluation, we compare NeutronStar with the typical DepCache-based system DistDGL [52] and the typical DepComm-based system ROC [14]. We use the default configurations for these systems. DistDGL relies on data sampling to reduce redundant computation cost [52], which is set to execute a (10, 25) neighborhood sampling for the training. In such a configuration, DistDGL picks a maximum of 10 neighbors for the first hop of a node, and then a maximum of 25 neighbors for each of those 10. For the single-node evaluation, we compare NeutronStar with the state-of-the-art systems DGL v0.5 [6], PyTorch Geometric [9], and the single-node configured ROC [14]. To evaluate the performance of our Hybrid approach, we also implement DepCache and DepComm in NeutronStar. Unless explicitly stated, the results are reported in per-epoch runtime, i.e., the time to conduct a forward and backward pass for all vertices in the graph. Shorter per-epoch time implies better time-to-accuracy performance. All the per-epoch runtime results are measured by averaging results of 100 epochs.

5.2 Performance Analysis of NeutronStar

We first evaluate Hybrid processing with DepComm and DepCache processing in NeutronStar to reveal where the performance enhancement of NeutronStar comes from.

Performance Gain Analysis. NeutronStar provides a hybrid dependency management method and implements several efficient optimizations. We quantify the gain from the hybrid dependency management and that from the optimizations separately. We first report the performance of raw DepCache, DepComm and Hybrid processing, and then integrate the system optimizations one-by-one to Hybrid processing, including the ring-based communication, the lock-free message queuing, and the communication/computation task overlapping as discussed in Section 4.

Figure 9 shows the normalized speedup of the raw Hybrid and DepComm processing over the raw DepCache processing. Hybrid

Table 3: Analysis of cost and benefit of Hybrid processing

Engine	Runtime of 100 epochs (s)						
	Goo	Pok	Liv	Red	Ork	Wik	Twi
DepCache	236.6	1327.1	1712.2	2866.7	4024.9	25828.1	28931.2
DepComm	311.4	730.9	1412.2	327.5	1509.1	4005.8	4728.1
Hybrid	141.5	294.4	591.9	162.6	652.6	1914.3	2258.8
Preprocessing	+1.7	+4.8	+8.4	+4.5	+16.9	+39.9	+58.3

can achieve 1.63X-10.34X and 1.24X-1.68X speedups over DepCache and DepComm, respectively. On Google, in which case DepCache has better performance than DepComm, Hybrid can achieve 1.32X speedup over DepComm and has nearly same performance as DepCache. The varied performance enhancements of Hybrid over DepCache and DepComm are related to the vertex distribution in the cluster after the graph partitioning is applied. For graphs with smaller average degrees, e.g, Google, DepCache has better performance than DepComm, because the overhead of redundant computation in DepCache is not that large. For graphs with larger average degrees, e.g., Reddit, DepComm is better than DepCache. In such cases, the overhead of redundant computation in DepCache is determined by the average degree of the partition, while DepComm is not sensitive to it because DepComm aggregates data from all vertices in the partition and communicates to dependent workers once. This is the idea in Hybrid (Section 3): if a vertex has a large number of dependent neighbors, we prefer the communication-based approach DepComm, and if a vertex has a small number of dependent neighbors, we prefer the cache-based approach DepCache.

As shown in the figure, the ring-based communication, noted as "R", can bring Hybrid an on average 1.10X-1.15X speedups over the non-optimized Hybrid. The lock-free message queuing, noted as "L", can further bring an 1.08X-1.12X speedup over the Hybrid only with the ring-based communication optimization. Finally, the communication/computation overlapping, noted as "P", can get additional 1.19X-1.41X speedups. In summary, NeutronStar that uses the Hybrid processing with all these optimizations can achieve 1.67X-16.75X, 2.09X-2.44X and 1.46X-1.77X speedups over the systems with the raw DepCache, DepComm, and Hybrid processing, respectively.

Overhead of Hybrid Dependency Partitioning. As the dependency management method has the cost to partition the input graph after graph partitioning, we evaluate the overhead and show in Table 3, where the execution time of Hybrid dependency partitioning is denoted as "Preprocessing" and we compare it to the execution time of running GCN for 100 epochs with DepComm, DepCache, and Hybrid, respectively. We observe that the hybrid dependency partitioning brings up to 3% overhead into the Hybrid processing, while significantly improves its performance over the DepComm and DepCache processing. The low overhead of hybrid dependency partitioning comes from two folds. First, the dependency partitioning algorithm uses a heuristic design and is executed in parallel to evaluate the cost of DepComm and DepCache for vertices (shown in Line(5-7) of Algorithm 4). Second, as GNN training follows the same pattern in different epochs, the dependency partitioning algorithm only needs to be executed once.

Varying DepCache-DepComm **Ratios.** NeutronStar employs the hybrid dependency management and can automatically determine the dependencies to be cached or to be communicated aiming to

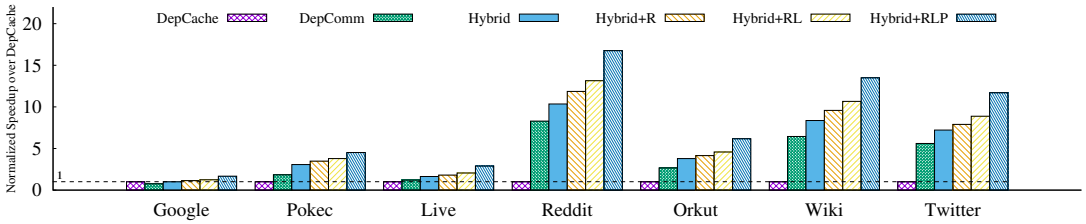

Figure 9: Performance analysis of NeutronStar on GCN, where R for the ring based communication, L for the lock-free message queuing, and P for the task-pipelining optimization.

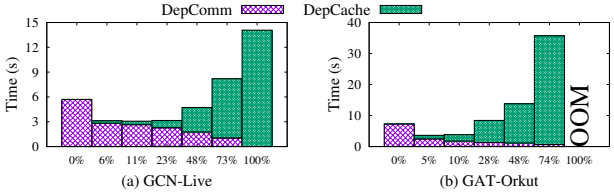

Figure 10: Runtime results when varying the ratios between cached dependencies and communicated dependencies.

achieve the optimal performance. To investigate how the performance changes under different ratios between cached dependencies and communicated dependencies, we disable the probing process and manually tune the factors $\{T_v, T_e, T_c\}$ to let the system prefer caching or communicating dependencies with various degrees. We run GCN on the LiveJournal graph and run GAT on the Orkut graph by varying the ratios between cached dependencies and communicated dependencies. Figure 10 shows the runtime decomposition results, where x-axis is the proportion of cached dependencies. Each runtime bar consists of the time proportion for processing the communicated dependencies and that for processing the cached dependencies. We can observe that neither communicating all dependencies nor caching all dependencies will reach the optimal performance. Caching all dependencies can even result in an out-of-memory error when running GAT on Orkut. The optimal performance is achieved when mixing DepCache and DepComm. Furthermore, our greedy algorithm (see Algorithm 4) chooses the most cache-efficient dependencies for DepCache and the most communicate-efficient dependencies for DepComm, which helps improving the performance of hybrid processing.

5.3 Overall Performance

Comparison with Distributed Systems. We compare the overall performance of NeutronStar with ROC [14], DistDGL [52], and DepCache and DepComm implementation in the NeutronStar's code base. Because ROC degrades significantly when scaling to more than 4 nodes, we report its best results on 4 nodes instead of 16 nodes. As ROC does not support GAT due to the lack of edge computation and DistDGL does not provide GIN's distributed implementation, we only report their available execution time. We enable all optimizations in DepComm processing as well as in Hybrid processing in the following experiments.

Figure 11 shows the per-epoch execution time of three GNN models with different GNN systems on multiple datasets. ROC and DepCache report the "out-of-memory" error for several cases, while NeutronStar (in both DepComm and Hybrid processing modes) and DistDGL can complete running in all cases. NeutronStar can achieve on average 1.81X-5.29X speedups over ROC. We also observe that our DepComm is also faster than the DepComm-based ROC. This means that the performance improvement over ROC is not only from our hybrid dependency management approach but also shows the effectiveness of system optimizations, such as the overlapping of communication and computation tasks and the parallel lock-free message queuing method. Our experiments indicate that the ROC worker does not differentiate the output messages with various destinations and send the whole messages block to all workers, where the remote workers pick the necessary dependencies from the block. This incurs significant communication cost. Due to the source-specific chunking, NeutronStar can greatly reduce the communication cost. On the other hand, NeutronStar achieves 1.83X-14.25X speedup over DistDGL. Though the sampling and mini-batch training are used to reduce the computation cost (with sacrifice of accuracy). The reason for the worst performance of DistDGL is as follows. Its sampling process needs additional computing and communication costs to access graph data in the distributed storage. In addition, compared to DepCache and optimized DepComm, NeutronStar achieves 2.03X-15.02X and 1.19X-1.69X speedup, respectively.

Table 4: Comparison with shared-memory-based systems

System	Per-epoch Runtime (s)			
	Goo	**Pok**	**Red**	**Ork**
DGL-CPU	4.59	19.95	45.80	50.57
PyG-CPU	251.850	OOM	OOM	OOM
NeutronStar-CPU (1 node)	8.93	20.64	12.63	47.05
NeutronStar-GPU (16 nodes)	1.41	2.94	1.62	6.53

Comparison with Shared-Memory-Based Systems. We compare the distributed NeutronStar with the shared-memory-based systems DGL-CPU, PyG-CPU, and NeutronStar-CPU (uses CPU-based PyTorch as its NN backend) in Table 4. We run the GCN model on four medium-size graphs that can fit into the memory of a single machine and show the per-epoch execution time. The PyG-CPU reports the Out-Of-Memory error on three large graphs, because it uses the matrix, instead of the compressed matrix, to store the graph. As shown in the table, NeutronStar on 16 GPUs can achieve the best performance.

5.4 GPU/CPU/Network Utilization

We evaluate the utilizations of GPU, CPU, and network resources during the training of GCN on Orkut, for DistDGL, ROC, and DepCache, DepComm, and Hybrid of NeutronStar. Figure 12 shows the results in a 20-second time window. The GPU and CPU utilization is recorded every 100 millseconds and averaged in an 1-second

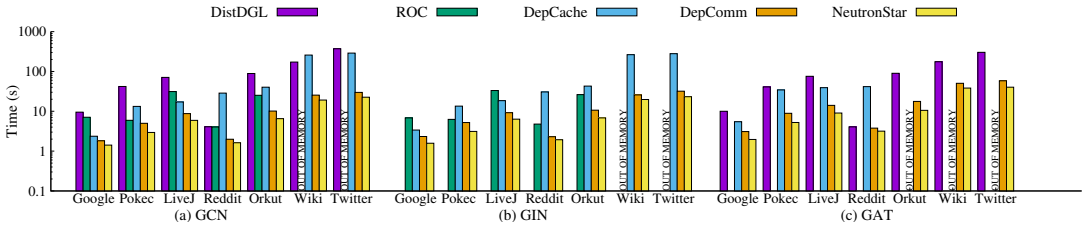

Figure 11: Overall performance comparison.

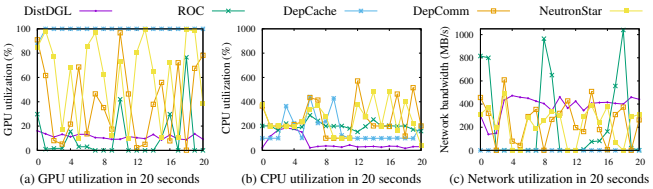

Figure 12: GPU utilization, CPU utilization, and network utilization comparison (GCN on Orkut).

interval. The network utilization is measured by counting the number of bytes received per second. Figure 12(a) shows that DepCache achieves full load of GPU (99.4% on average) due to redundant computation, and DepComm (39.9% on average) and NeutronStar (60.5% on average) achieves higher GPU utilization than ROC (10.2% on average) due to the computation/communication overlapping technique. DistDGL has quite low GPU utilization, i.e., 11.30% on average. This is because that DistDGL has a sampling step to fetch remote dependent samples, which can be the bottleneck to limit the GPU utilization. Figure 12(b) shows that ROC (211% on average), DepComm (270% on average) and NeutronStar (268% on average) have higher CPU utilization than that of DepCache (177% on average) and DistDGL (63.7% on average). This is because that CPU is often used for communications and DepCache does not need to communicate dependencies between workers, and DistDGL's sampling process is the bottleneck that is bounded by the I/O throughput of the storage. Additionally, DepComm and NeutronStar have higher CPU utilization over ROC due to the lock-free parallel message queuing. Figure 12(c) shows that DepCache consumes much less network bandwidth than others and DepComm and NeutronStar can smooth the bandwidth curve than ROC. This can be attributed to the design of ring-based communication and the overlapping of communication and computation tasks. DistDGL uses the most bandwidth as it relies on the sample-based mini-batch training that needs to fetch the dependency samples continuously.

5.5 Scaling Performance

We evaluate the scalability of NeutronStar in DepCache, DepComm, and Hybrid with DistDGL and ROC. Figure 13 shows the execution time of GCN training on Pokec, Reddit, Orkut and Wiki with different cluster sizes. Due to the GPU memory limitation, some graphs can not fit into 1 or 2 machines and we report the execution results from the minimum number of cluster sizes.

We observe that the execution time of DistDGL and NeutronStar in DepComm and Hybrid are reduced with more nodes. However, ROC exhibits poor scalability. That may relate to the implementation of communication in ROC, where each worker needs to pull the entire partitioning data from a remote worker, even if only a small subset of data is required. In contrast, NeutronStar employs the source-specific chunking method to improve the communication performance and gets almost linear speedup. As the cluster size increases, NeutronStar in Hybrid achieves 2.0X speedup on Pokec (from 2 to 16 nodes), 6.40X speedup on Reddit (from 1 to 16 nodes), 2.52X speedup on Orkut (from 4 to 16 nodes) and 1.61X speedup on Wiki (from 8 to 16 nodes). Because each worker of DepCache caches all dependencies, the redundant computation does not decrease with more nodes. It hence exhibits poor scalability.

5.6 Accuracy Comparisons

Figure 14 reports the accuracy for Hybrid, DepComm, DepCache, and DepCache-Sampling on Reddit. DepCache-Sampling uses DGL's sampling on one node, and others use NeutronStar with 16 nodes. We also enable all optimizations for Hybrid and DepComm for fair comparison. After running enough epochs to fully converge, DepComm, DepCache and Hybrid can reach an accuracy of **95.22%**, **94.12%** and **94.86%**, respectively. DepCache-Sample can reach its highest accuracy of **93.92%**. Therefore, we set **93.92%** as the target accuracy to make the training finish in reasonable time. As shown in Figure 14, Hybrid can reach the target accuracy faster than all others. Although Hybrid requires more epochs to reach the **93.92%** accuracy than DepComm, Hybrid is still 1.20X faster than DepComm as it needs less execution time in each epoch. Hybrid is 1.96X faster than DepCache-Sampling. Benefiting from the mini-batch training, DepCache-Sampling requires least training epochs. But it is still slower than Hybrid due to the long per-epoch runtime and fails to achieve higher accuracy due to the sampling. As shown in the figure, DepCache can not reach the target accuracy until 3669.4 seconds later, leading to the lowest convergence speed.

5.7 With Graph Partitioning Algorithms

We evaluate the effectiveness of our hybrid dependency management with different graph partitioning algorithms: chunk-based method [54], Metis [16], and Fennel [40]. Figure 15 presents the performance of NeutronStar in DepComm and Hybrid with these three partitioning methods on Reddit, Orkut, and Wiki. We enable all optimizations for both DepComm and Hybrid for a fair comparison. We observe that Hybrid achieves 1.21-1.48X, 1.12-1.23X, and 1.17-1.32X speedups over DepComm with chunk-based, Metis, and Fennel graph partitioning, respectively. This demonstrates that the dependency management is orthogonal to the graph partitioning and a distributed GNN training system can get benefits from both of Hybrid dependency management and graph partitioning.

5.8 Performance Comparison on a single GPU

We also compare NeutronStar with the single-machine GNN systems DGL [52] and PyG [9] as well as ROC by running two GNN

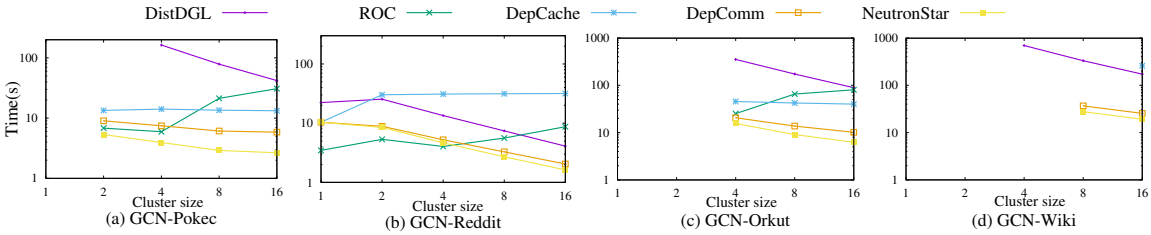

Figure 13: Scaling performance when varying cluster size from 1 to 16.

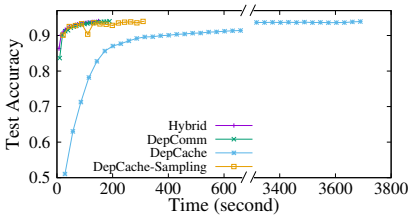

Figure 14: Accuracy comparisons between Hybrid, DepComm, DepCache, **and** DepCache-**sampling with GCN on the Reddit dataset. Each dot indicates five training epochs for** Hybrid **and** DepComm, **and one training epoch for** DepCache **and** DepCache-**sampling.**

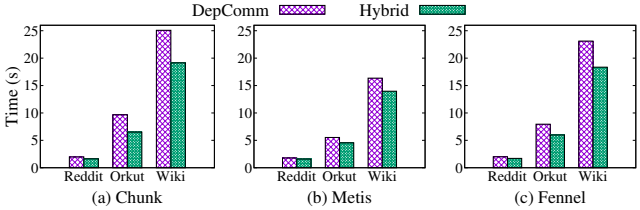

Figure 15: Performance comparison of optimized DepComm **and** Hybrid **under different graph partition algorithms.**

models on four small graphs cora, citeseer, pubmed, and Google. Table 5 shows the runtime results. We can see that NeutronStar (NTS) achieves comparable performance with DGL and PyG for both GCN and GAT and achieves 1.96-5.18X speedup over ROC on GCN model. Due to the lack of edge-centric NN computation support, ROC does not support GAT model. DGL and PyG report out-of-memory error when processing the Google graph. While NeutronStar can process much larger graphs by caching intermediate result in host memory. Moreover, benefiting from the chunk-based message computation, NeutronStar does not need to maintain the large edge tensors completely in device memory.

Table 5: Comparison with single-GPU-based system

Sys.	Runtime of GCN (ms)				Runtime of GAT (ms)			
	Cor	**Cit**	**Pub**	**Goo**	**Cor**	**Cit**	**Pub**	**Goo**
ROC	11.11	22.05	19.45	2298.3	-	-	-	-
DGL	8.15	9.63	8.4	OOM	13.6	15.56	13.45	OOM
PyG	**3.53**	**7.69**	4.55	OOM	**8.97**	**15.17**	9.96	OOM
NTS	4.45	8.7	**3.75**	**1167.7**	9.91	16.45	**9.24**	**2634.5**

6 RELATED WORK

Dependencies Cached GNN Systems: The DepCache approach is preferred by big companies. It is usually adopted with a sampling process which samples a subset of dependencies for mini-batch training. Alibaba's Aligraph [53] and Euler [7], Ant group's AGL [51], and Amazon's DistDGL [52] all adopt this mechanism. However, the sample-based strategy has been proved to have lower accuracy than full-batch training [14, 29]. P3 [10] is recently proposed to completely avoid communicating the data-intensive node features over the network by combining intra-layer model parallelism and data parallelism and further reduces communication using a simple dependencies caching mechanism.

Dependencies Communicated GNN Systems: DepComm relies on full-batch training and is endowed with high accuracy. Besides ROC [14], a series of DepComm-based systems are proposed recently. DistGNN [29] studies the aggregation operation in GNN training and proposes a hierarchical aggregation method to improve efficiency. DGCL [3] analyses the speeds of heterogeneous communication among devices and proposes an automatic routing algorithm to improve the communication efficiency. Dorylus [37] takes advantage of serverless computing to increase scalability at a low cost. Its key idea is computation separation. Tasks that operate over graph structure data are executed by CPU instances, while those processing tensor data are executed by Lambdas. The authors of [39] improve the efficiency of training through 1.5D, 2D, or 3D graph partition. To our best knowledge, NeutronStar is the first that combines DepCache and DepComm for the merits of both.

Single Machine GNN Systems: There exist a number of GNN systems that run on single machine. PyG [9] and DGL [43] are two typical GNN systems that provide message passing primitives. NeuGraph [27] defines a new, flexible SAGA-NN model to express GNNs. It fuses graph-related optimizations into deep learning frameworks and supports training on multiple GPUs. G3 [25] introduces a GPU-based PGPS (Parallel Graph Processing System) to train GNN. GNNAdvisor [44] implements a novel and highly-efficient 2D workload management tailored for GNN computation to improve GPU utilization. Marius [31] scale GNN training to large graphs on a single machine by fully utilizing the external storage. In [30], the authors make GPU threads directly access sparse features in host memory through zero-copy accesses without much CPU's help. Seastar [46] identifies the abundant operator fusion opportunities of the computational graphs in GNN training.

7 CONCLUSION

We have designed and implemented NeutronStar, which supports DepCache-DepComm hybrid dependency management, flexible auto differentiation framework, and rich optimizations for CPU-GPU heterogeneous computation and efficient communication and provides high-level python-based APIs. Compared with the state-of-the-art GNN frameworks DistDGL and ROC, NeutronStar achieves a performance speedup 1.8X-14.3X and proved to have better scalability and more balanced computation-communication resource utilization.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283, 2016.

[2] L. Backstrom, D. P. Huttenlocher, J. M. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, pages 44–54, 2006.

[3] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu. DGCL: an efficient communication library for distributed GNN training. In A. Barbalace, P. Bhatotia, L. Alvisi, and C. Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 130–144. ACM, 2021.

[4] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, pages 1:1–1:15, 2015.

[5] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 3837–3845, 2016.

[6] Deep graph library:towards efficient and scalable deep learning on graphs, 2020. https://www.dgl.ai/.

[7] Euler, 2019. https://github.com/alibaba/euler/wiki/System-Introduction.

[8] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, Z. Zheng, B. Zhang, Y. Cao, and C. Tian. Parallelizing sequential graph computations. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 495–510, 2017.

[9] M. Fey and J. E. Lenssen. Fast graph representation learning with pytorch geometric. *CoRR*, abs/1903.02428, 2019.

[10] S. Gandhi and A. P. Iyer. P3: distributed deep graph learning at scale. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 551–568, 2021.

[11] W. L. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 1024–1034, 2017.

[12] L. Hu, L. Zou, and Y. Liu. Accelerating triangle counting on GPU. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 736–748, 2021.

[13] Z. Jia, Y. Kwon, G. M. Shipman, P. S. McCormick, M. Erez, and A. Aiken. A distributed multi-gpu system for fast graph processing. *Proc. VLDB Endow.*, 11(3):297–310, 2017.

[14] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, 2020.

[15] G. Karypis and V. Kumar. Parallel multilevel graph partitioning. In *Proceedings of IPPS '96, The 10th International Parallel Processing Symposium, April 15-19, 1996, Honolulu, Hawaii, USA*, pages 314–319, 1996.

[16] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[17] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[18] J. Kunegis. KONECT: the koblenz network collection. In *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*, pages 1343–1350, 2013.

[19] H. Kwak, C. Lee, H. Park, and S. B. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 591–600, 2010.

[20] C. Lei. *Deep Learning and Practice with MindSpore.* Cognitive Intelligence and Robotics. Springer, 2021.

[21] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Math.*, 6(1):29–123, 2009.

[22] H. Li, Y. Liu, Y. Li, B. Huang, P. Zhang, G. Zhang, X. Zeng, K. Deng, W. Chen, and C. He. Graphtheta: A distributed graph neural network learning system with flexible training strategy. *CoRR*, abs/2104.10569, 2021.

[23] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, 2020.

[24] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel. Gated graph sequence neural networks. In Y. Bengio and Y. LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[25] H. Liu, S. Lu, X. Chen, and B. He. G3: when graph neural networks meet parallel graph processing systems on gpus. *Proc. VLDB Endow.*, 13(12):2813–2816, 2020.

[26] Q. Liu, M. Nickel, and D. Kiela. Hyperbolic graph neural networks. *CoRR*, abs/1910.12892, 2019.

[27] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai. Neugraph: Parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 443–458, 2019.

[28] D. Marcheggiani and I. Titov. Encoding sentences with graph convolutional networks for semantic role labeling. In M. Palmer, R. Hwa, and S. Riedel, editors, *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pages 1506–1515. Association for Computational Linguistics, 2017.

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

[30] S. Min, K. Wu, S. Huang, M. Hidayetoglu, J. Xiong, E. Ebrahimi, D. Chen, and W. W. Hwu. Large graph convolutional network training with gpu-oriented data communication architecture. *Proc. VLDB Endow.*, 14(11):2087–2100, 2021.

[31] J. Mohoney, R. Waleffe, H. Xu, T. Rekatsinas, and S. Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 533–549, 2021.

[32] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[33] Tensors and dynamic neural networks in python with strong gpu acceleration, 2020. https://pytorch.org/.

[34] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Gallagher, and T. Eliassi-Rad. Collective classification in network data. *AI Mag.*, 29(3):93–106, 2008.

[35] G. M. Slota, S. Rajamanickam, K. D. Devine, and K. Madduri. Partitioning trillion-edge graphs in minutes. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*, pages 646–655, 2017.

[36] L. Takac and M. Zabovsky. Data analysis in public social networks. *International Scientific Conference & International Workshop Present Day Trends of Innovations, May 28-29*, 2012.

[37] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim, and G. H. Xu. Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 495–514, 2021.

[38] A. Tripathy, K. A. Yelick, and A. Buluç. Reducing communication in graph neural network training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 70, 2020.

[39] A. Tripathy, K. A. Yelick, and A. Buluç. Reducing communication in graph neural network training. *CoRR*, abs/2005.03300, 2020.

[40] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. FENNEL: streaming graph partitioning for massive scale graphs. In *Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014, New York, NY, USA, February 24-28, 2014*, pages 333–342, 2014.

[41] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[42] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang. Sep-graph: finding shortest execution paths for graph processing under a hybrid framework on GPU. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, pages 38–52, 2019.

[43] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR*, abs/1909.01315, 2019.

[44] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding. Gnnadvisor: An adaptive and efficient runtime system for GNN acceleration on gpus. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 515–531, 2021.

[45] S. Wu, W. Zhang, F. Sun, and B. Cui. Graph neural networks in recommender systems: A survey. *CoRR*, abs/2011.02260, 2020.

[46] Y. Wu, K. Ma, Z. Cai, T. Jin, B. Li, C. Zheng, J. Cheng, and F. Yu. Seastar: vertex-centric programming for graph neural networks. In A. Barbalace, P. Bhatotia, L. Alvisi, and C. Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 359–375. ACM, 2021.

[47] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *CoRR*, abs/1901.00596, 2019.

[48] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[49] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowl. Inf. Syst.*, 42(1):181–213, 2015.

[50] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li. Graph edge partitioning via neighborhood heuristic. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*, pages 605–614, 2017.

[51] D. Zhang, X. Huang, Z. Liu, J. Zhou, Z. Hu, X. Song, Z. Ge, L. Wang, Z. Zhang, and Y. Qi. AGL: A scalable system for industrial-purpose graph machine learning. *Proc. VLDB Endow.*, 13(12):3125–3137, 2020.

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

[53] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou. Aligraph: A comprehensive graph neural network platform. *PVLDB*, 12(12):2094–2105, 2019.

[54] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 301–316, 2016.