NeutronCloud: Resource-Aware Distributed GNN Training in Fluctuating Cloud Environments

Mingyi Cao*
Northeastern University
Shenyang, China
2201763@
stu.neu.edu.cn

Chunyu Cao*
Northeastern University
Shenyang, China
caochunyu@
stumail.neu.edu.cn

Yanfeng Zhang
Northeastern University
Shenyang, China
zhangyf@
mail.neu.edu.cn

Zhenbo Fu
Northeastern University
Shenyang, China
fuzhenbo@
stumail.neu.edu.cn

Xin Ai
Northeastern University
Shenyang, China
aixin0@
stumail.neu.edu.cn

Qiange Wang
Northeastern University
Shenyang, China
wangqiange94@
gmail.com

Yu Gu
Northeastern University
Shenyang, China
guyu@
mail.neu.edu.cn

Ge Yu
Northeastern University
Shenyang, China
yuge@
mail.neu.edu.cn

ABSTRACT

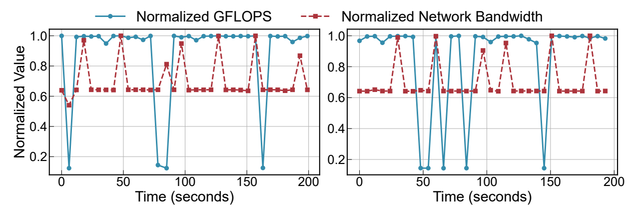
Graph Neural Networks (GNNs) are widely employed to learn representations from graph-structured data. To support large-scale graph training, researchers use distributed techniques, partitioning the graph across multiple computing nodes and performing parallel training by exchanging dependency vertex information via cross-node communication. However, existing GNN training systems operate on statically partitioned subgraphs, making them difficult to adapt to resource fluctuations. In practice, resource fluctuations in cloud environments often cause variability in compute and communication resources, posing challenges for aligning each worker's workload to its available resources during GNN training.

In this paper, we propose NeutronCloud, a system designed for efficient GNN training in cloud environments. First, we adopt a resource-aware workload adjustment strategy. It builds on hybrid dependency handling by obtaining dependency information through both local computation and remote communication. During training, it dynamically adjusts the ratio between locally computed and remotely fetched dependencies based on each worker's available resources, ensuring workload-resource alignment. Second, we employ a dependency-aware partial-reduce approach reusing historical vertex embeddings and skipping the stragglers during gradient aggregation to address extreme resource fluctuations that cause some workers to lag significantly behind others in the cluster. Experimental results on the resource-fluctuating environment demonstrate that NeutronCloud achieves 1.83×-4.43× speedup compared to state-of-the-art distributed GNN systems.

PVLDB Reference Format:
Mingyi Cao, Chunyu Cao, Yanfeng Zhang, Zhenbo Fu, Xin Ai, Qiange Wang, Yu Gu, Ge Yu. NeutronCloud: Resource-Aware Distributed GNN Training in Fluctuating Cloud Environments. PVLDB, 19(1): 56 - 69, 2025.
doi:10.14778/3772181.3772186

*Both authors contributed equally to this research.

Proceedings of the VLDB Endowment, Vol. 19, No. 1 ISSN 2150-8097.
doi:10.14778/3772181.3772186



(a) Worker 1 (b) Worker 2

Figure 1: Normalized computational throughput (FLOPS) and network bandwidth measured over 200 seconds on two Alibaba Cloud shared GPU instances (each with a shared NVIDIA A10 GPU and up to 10 Gbps bandwidth).

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Toaoc/NeutronCloud.

1 INTRODUCTION

Graph Neural Networks (GNNs) have emerged as a fundamental approach for graph-based tasks such as social network analysis [5, 11], link prediction [22, 31, 52], and recommendation systems [49]. The GNN models iteratively perform neighbor aggregation and representation updates for each vertex to capture complex topological information. Distributed training methods are adopted to partition graph data across multiple workers to handle large-scale graphs, running GNN models in parallel. Most existing distributed GNN training systems [3, 6, 10, 16, 24, 34, 37, 38, 42, 44, 51, 54, 57] assume a stable resource environment, where computational and network resources remain consistent during training. However, in reality, resource fluctuations are ubiquitous in real-world applications.

Resource fluctuations can be categorized into computational resource fluctuations and network resource fluctuations. In cloud environments, computational resources fluctuate due to contention on shared physical hosts [35]. For example, burstable instances (e.g., AWS T-series or Alibaba Cloud "shared-core" VMs) [1, 36] may decrease CPU/GPU performance when neighboring VMs experience high demand. Likewise, network performance varies due to congestion and bandwidth contention. Multi-tenant clusters often face communication bottlenecks caused by overloaded data center switches and cross-worker network contention [28, 47, 53]. To illustrate these fluctuations, we evaluate computational power (FLOPS, floating point operations per second) and network bandwidth over time in a two-worker cluster on Alibaba Cloud's shared GPU instances. As shown in Figure 1, FLOPS and network bandwidth drop by up to 90% and 35%, respectively.

Such fluctuations significantly impact the performance of distributed GNN training. When performing distributed GNN training in a resource-fluctuating environment, the initially assigned workload becomes mismatched with the fluctuating resources, causing computational and communication bottlenecks, and leading to the emergence of slower workers (stragglers). In addition, GNN data samples exhibit complex interdependencies, further exacerbating this issue. In distributed GNN training, graph partitioning results in local vertices needing to aggregate features from remote neighbors (called remote dependency vertices), and all worker parameters need to be synchronized at the end of each epoch. Thus, the delay caused by stragglers propagates throughout the entire cluster during training, slowing down the overall training performance.

We summarize that the key challenge in distributed GNN training under resource fluctuations is how to quickly adjust each worker's computational and communication workload to match the available resources. For Deep Neural Network (DNN) systems, there are no dependencies among input samples. They mitigate resource-load mismatches through real-time load migration to address resource fluctuations [14, 56]. However, for GNN systems, the data dependencies make workload migration difficult. Workload migration requires not only transferring vertex features but also adjusting and maintaining the dependencies between workers to ensure graph consistency. This leads to substantial additional overhead and reduces the efficiency of workload adjustment. The overhead of migration can even offset the benefits of workload adjustment.

In this paper, we propose NeutronCloud, a distributed GNN training system capable of resource-aware workload adjustment, addressing the above challenge through two critical strategies.

First, we propose a resource-aware workload adjustment strategy that adapts dependency handling based on real-time resource conditions. The fetching of remote dependencies takes up most of the time in the entire distributed GNN training process [2, 12, 32, 37, 45]. We propose a lightweight resource-aware workload adjustment strategy based on the hybrid dependency handling method. This strategy dynamically adjusts the processing of remote dependency vertices in the cluster to adjust computational and communication workloads. Specifically, we cache remote dependency vertices and their multi-hop neighbors, obtaining embeddings through local computation to address the reduction in communication resources. When computational resources are constrained, more remote dependency vertex embeddings are fetched via cross-worker communication. To enable flexible runtime adjustment, we trade additional storage for adaptability by pre-caching remote dependency vertices and their multi-hop neighbors during the preprocessing phase. This design eliminates the need for workload migration during adjustment.

Second, when some workers face extreme resource degradation (e.g., computation and communication resources decrease simultaneously), adjusting vertex dependencies processing is not enough to reduce the serious delay caused by the synchronization of the severe stragglers. To address this problem, we introduce a dependency-aware partial-reduce strategy, allowing local computation using cached historical embeddings when embeddings from severe stragglers cannot be received in time. Despite being slightly outdated, these embeddings still provide useful information for computation without stalling the process. During gradient aggregation, we synchronize gradients only from faster workers, skipping the severe stragglers. To ensure unbiased gradients and model convergence, we adopt a dependency-aware weighted gradient aggregation strategy and set a bound on severe stragglers that are skipped.

In summary, we make the following contributions.

- We propose a resource-aware workload adjustment strategy, which dynamically adjusts the number of remote dependencies for each worker by quantifying variations in computation and communication resources, ensuring a better match between resource and workload.
- We propose a dependency-aware partial-reduce approach to reduce synchronization overhead. By using history embeddings, we update only the faster worker's parameters while setting a bound to ensure convergence.
- We develop NeutronCloud, an efficient GNN training system for resource fluctuation environments. The experimental results show that NeutronCloud achieves 1.83× - 4.43× speedup compared to state-of-the-art GNN systems.

2 BACKGROUND

2.1 Resource Fluctuations

Resource fluctuations are common in real-world applications and are often caused by contention on shared physical hosts or network contention. These resource fluctuations frequently impact computing power and network bandwidth, significantly affecting the performance of distributed GNN training.

To analyze the impact of resource fluctuations on the performance of distributed GNN systems, we evaluate two GNN training systems, NeutronStar [45] and Sancus [32], on resource-fluctuating and resource-stable clusters using a two-layer GCN model. The cluster consists of four workers, each equipped with NVIDIA T4 GPUs, interconnected via a 10 Gbps network.

Motivated by existing methods for simulating resource fluctuations [25–27, 56], we emulate runtime fluctuations by injecting sleep commands into workers. Specifically, in each iteration, each worker incurs additional overhead with a 10% probability, equivalent to twice the average epoch runtime. To emulate sustained fluctuation patterns (as shown in Figure 1), each injected sleep command lasts for 5 consecutive epochs.

As shown in Figure 2, the per-epoch runtime in a resource-fluctuating environment is 3.5× slower than that in a resource-stable environment. Distributed GNN training needs to synchronize data across all workers in each epoch. Resource fluctuations on any worker can impact the efficiency of the entire cluster. This indicates that resource fluctuations can significantly degrade the efficiency of distributed GNN training.

2.2 GNN Training

A graph neural network processes a graph as input, where each vertex and edge is associated with high-dimensional features. Through

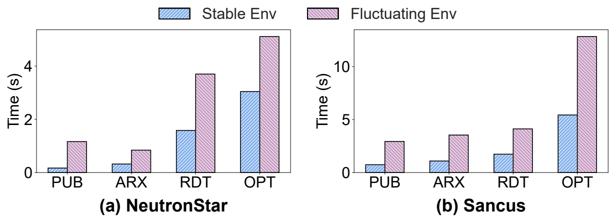


Figure 2: Per-epoch runtime comparison of NeutronStar [45] and Sancus [32] under stable environment (Stable Env) and resource-fluctuating environment (Fluctuating Env) on four datasets PUB (Pubmed), ARX (Ogbn-Arxiv), RDT (Reddit), and OPT (Ogbn-Products).

multiple layers, GNNs iteratively propagate and aggregate information from neighboring vertices, updating their representations to capture the graph structure information.

In the l-th layer, the aggregation result a_v^l of vertex v is obtained by collecting the embedding h_u^{l-1} of its neighboring vertex u in the $(l-1)$-th layer, as shown in the following formula:

$$a_v^l = Aggregate(h_u^{l-1}, u \in N(v)) \tag{1}$$

Subsequently, the update function uses the result a_v^l and h_v^{l-1}, which is the embedding of the vertex v in the $(l-1)$-th layer, to compute the representation h_v^l of the vertex v in the l-th layer, as shown in the following formula.

$$h_v^l = Update(a_v^l, h_v^{l-1}) \tag{2}$$

The final-layer vertex embeddings are fed into downstream tasks, where a task-specific loss is computed against ground truth labels. This loss triggers backward propagation to calculate gradients via automatic differentiation. These gradients drive parameter updates through gradient-based optimizers like SGD or its adaptive variants (e.g., Adam). The standard SGD update rule is:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t) \tag{3}$$

where the η denotes the learning rate controlling the step size, the $\nabla_\theta \mathcal{L}(\theta_t)$ denotes the gradient of loss function w.r.t parameters.

2.3 Distributed GNN Training Approach

Distributed GNN training partitions the input graph across multiple workers. Dependencies arise when vertices need to aggregate features from remote neighbors. The critical aspect of distributed GNN systems is efficiently handling remote dependencies.

We categorize existing dependency handling strategies into three approaches. The **Dependencies Communicated** (DepComm) approach requires each vertex to gather its neighbors' representations from remote workers via cross-worker communication [17, 38], as shown in Figure 3 (a). This method reduces storage consumption but leads to significant communication overhead. The **Dependencies Cached** (DepCache) approach caches the features of multi-hop neighbors of remote dependency vertices on the local worker in advance for multi-layer computing [57], as shown in Figure 3 (b). This eliminates inter-worker communication but results in significant redundant computation and storage overhead.

As shown in Figure 3(c), the hybrid dependency-resolution scheme assigns a subset of remote dependency vertices to DepCache (e.g., vertex 0 on worker 1) and the remainder to DepComm (e.g., vertex 1 on worker 1). The approach can balance the use of computational and communication resources. Based on this hybrid design, we

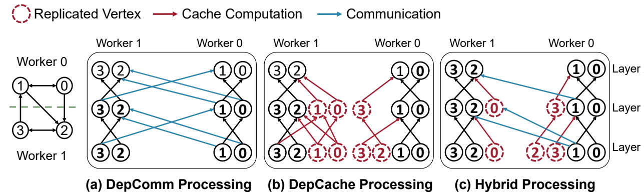


Figure 3: Hybrid dependency handling method of distributed GNN training.

Algorithm 1 DepComm-DepCache Hybrid Training for L-layer GCN

Input: $G = (V, E)$, L, $\{\mathbf{h}_v^{(0)}\}_{v \in V}$, $\{L_v\}_{v \in V_L}$; partition $\{V_1, \ldots, V_m\}$; params $\{W^{(\ell)}\}_{\ell=1}^{L}$
Output: updated $\{W^{(\ell)}\}_{\ell=1}^{L}$
1: For worker i: $E_i \leftarrow \{(u, v) \in E \mid v \in V_i\}$; $N_{\text{loc}}(v) = N(v) \cap V_i$, $N_{\text{rem}}(v) = N(v) \setminus V_i$
2: **for each worker** $i = 1, \ldots, m$ **in parallel do**
3: **for** $\ell = 1$ to L **do** ▷ Forward
4: **for** each $v \in V_i$ **do**
5: $S_{\text{loc}} \leftarrow \{\mathbf{h}_u^{(\ell-1)} : u \in N_{\text{loc}}(v)\}$
6: $F_\ell(v) \leftarrow \{u \in N_{\text{rem}}(v) : \text{DepComm}(u, v, \ell)\}$
7: $C_\ell(v) \leftarrow N_{\text{rem}}(v) \setminus F_\ell(v)$
8: $S_{\text{cache}} \leftarrow \{\text{CacheGet}(u, \ell-1) : u \in C_\ell(v)\}$
9: PullByOwner $\{\mathbf{h}_u^{(\ell-1)} : u \in F_\ell(v)\}$ from owner(u)
10: $S_{\text{comm}} \leftarrow \{\mathbf{h}_u^{(\ell-1)} : u \in F_\ell(v)\}$
11: $S \leftarrow S_{\text{loc}} \cup S_{\text{cache}} \cup S_{\text{comm}}$
12: $a_v \leftarrow \text{Aggregate}(S)$
13: $\mathbf{h}_v^{(\ell)} \leftarrow \text{Update}(a_v, \mathbf{h}_v^{(\ell-1)}; W^{(\ell)})$
14: $\hat{L}_v \leftarrow P(\mathbf{h}_v^{(L)})$ for $v \in V_{L,i}$; $loss_i \leftarrow \text{Loss}(\{\hat{L}_v\}, \{L_v\})$; $loss \leftarrow \text{AllReduceSum}(loss_i)$
15: Backprop to obtain $\{\nabla \mathbf{h}_v^{(\ell-1)}\}$ and $\{\nabla W^{(\ell)}\}$ for $v \in V_i$
16: **for** $\ell = L$ **down to** 1 **do** ▷ Backward: send grads only for DepComm neighbors
17: **for** each $v \in V_i$ **do**
18: **for** each $u \in F_\ell(v)$ **do**
19: Send $\nabla \mathbf{h}_u^{(\ell-1)}$ to owner(u)
20: **for** $\ell = 1$ to L **do** ▷ Sync & update
21: $\nabla W^{(\ell)} \leftarrow \text{AllReduceSum}(\nabla W^{(\ell)})$
22: $W^{(\ell)} \leftarrow \text{OptStep}(W^{(\ell)}, \nabla W^{(\ell)})$

propose a dynamic adjustment mechanism that adaptively adjusts the ratio between DepCache and DepComm during training. This enables each worker to adapt its computation and communication workload to fluctuating resources.

Algorithm 1 outlines the hybrid dependency handling. In the forward pass (Lines 5–13), the worker collects local neighbors, partitions remote neighbors into DepComm and DepCache, computes the DepCache branch locally, fetches DepComm embeddings via communication, and then aggregates and updates. The per-worker loss is computed and reduced across workers (Line 14). In the backward pass (Lines 15–20), local gradients are computed on each worker, and only gradients for DepComm remote neighbors are sent to their owners, while DepCache neighbors are handled locally. Finally (Lines 21–22), gradients are All-Reduced and parameters are updated.

Therefore, the exchange of vertex embeddings and gradients at each layer, combined with the synchronization of parameter updates, creates layer-wise communication barriers, introducing significant synchronization overhead to distributed GNN training.

2.4 Existing GNN Systems in Fluctuating Environments

Most existing distributed GNN systems are built on the assumption of stable resource availability. Systems such as DGL [43], AliGraph [57], MGG [46], ROC [17], and DGCL [2] rely on static, predefined task allocation and communication–computation pipelines, with optimizations for computation, memory, or communication applied at initialization. These designs assume static resource conditions and therefore cannot adapt to runtime resource fluctuations. As a result, execution plans quickly become suboptimal, and delays caused by stragglers can cascade across synchronization barriers, leading to resource underutilization.

Some recent systems attempt to mitigate the impact of stragglers caused by resource fluctuations. PipeGCN [41] adopts pipelined execution with historical embeddings to overlap computation and communication, while Sancus [32] introduces historical embeddings to reduce communication with stragglers. These approaches can partially hide the latency introduced by stragglers. However, they still rely on synchronous parameter updates (e.g., all-reduce), which are bottlenecked by the slowest worker in each iteration and limit adaptability under resource fluctuations.

While NeutronStar [45] also adopts a hybrid dependency handling strategy, it lacks runtime resource awareness and cannot adjust dependency handling based on dynamic execution conditions. Moreover, its hybrid dependency handling method requires evaluating the benefit of each dependency and performing global sorting, which incurs high overhead and makes it unsuitable for online adjustment.

3 SYSTEM OVERVIEW

We introduce NeutronCloud, a distributed GNN system capable of handling resource fluctuations through two critical strategies. First, NeutronCloud provides a resource-aware workload adjustment strategy that adjusts the computational and communication workloads of each worker to match the real-time resource conditions. Second, NeutronCloud introduces a dependency-aware partial-reduce approach to reduce synchronization overhead between fast and slow workers.

Resource-aware Workload Adjustment Strategy. NeutronCloud employs a lightweight resource-aware workload adjustment algorithm based on the hybrid dependency handling method. The strategy consists of two phases. In the preprocessing phase, remote dependency vertices and their multi-hop neighbors are cached in local CPU memory. Additionally, the costs of different handling methods for remote dependency vertices are computed, thereby determining per-vertex adjustment priorities. In the online adjustment phase, the algorithm continuously records the computation and communication time of each worker per epoch and quantifies resource fluctuations. Based on the adjustment priorities obtained in the preprocessing phase, a binary search is performed to determine the handling method for remote dependency vertices in the next epoch, thereby speeding up the adjustment process. Moreover, no workload migration is required during the adjustment process.

Dependency-aware Partial-reduce Strategy. We propose a dependency-aware partial-reduce method to address the significant

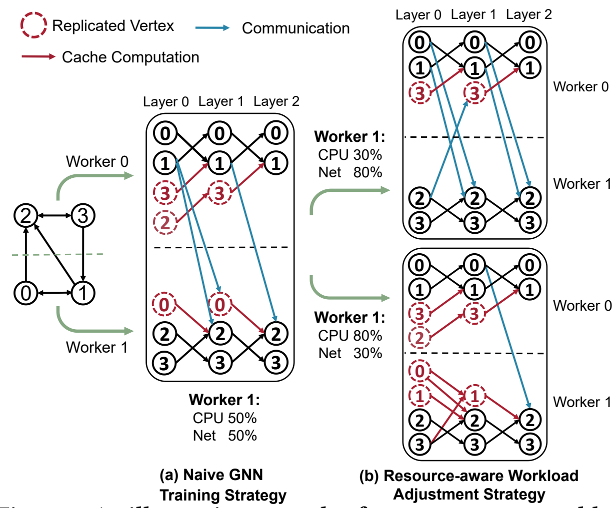


Figure 4: An illustrative example of resource-aware workload adjustment.

synchronization time caused by severe stragglers. This method consists of partial computation and partial update. Partial computation is applied to graph propagation (including both forward and backward propagation), when computing remote dependency vertices handled by the DepComm approach, if the severe stragglers fail to provide timely embedding updates, we use the locally cached historical embeddings of dependency vertices to continue forward propagation. Similarly, historical gradients are used for backward propagation. Additionally, a lightweight controller monitors embedding and gradient staleness, ensuring they are used only if their divergence from the latest embeddings is below a certain threshold.

Partial update limits gradient aggregation to faster workers and skips the stragglers. A runtime controller monitors worker progress to ensure that each update involves the majority of workers. To maintain fairness and convergence, we ensure that all workers participate in parameter updates within a bounded number of epochs, and cached embeddings are periodically refreshed. Finally, we apply dependency-aware weighted parameter synchronization to ensure unbiased aggregation.

4 RESOURCE-AWARE WORKLOAD ADJUSTMENT

In this section, we first analyze the impact of resource fluctuation on GNN training. Then, we define an optimization objective that minimizes the runtime of each worker under resource-fluctuating environments by adjusting how dependencies are handled. Finally, we propose a lightweight algorithm to approximate this objective during training.

4.1 Impact Study of Resource Fluctuations

To design our workload adjustment strategy, we analyze how resource fluctuations affect each worker's total runtime, including local subgraph computation, DepCache computation, and DepComm communication. Given the processing times of the two dependency-handling approaches, we formulate an objective to minimize the total runtime by adjusting their proportions.

Training Time under Resource Fluctuations. We decompose each worker's training time under resource fluctuations into three components: $T'_{\text{local},i}$ for local subgraph computation, $T'_{\text{cache},i}$ for DepCache computation, and $T'_{\text{comm},i}$ for DepComm communication. This decomposition is expressed as:

$$T'_i = T'_{\text{local},i} + T'_{\text{cache},i} + T'_{\text{comm},i} \tag{4}$$

Impact of Resource Fluctuations on DepCache. DepCache improves training efficiency by locally caching the features of multi-hop dependency neighbors to reduce communication. For each vertex, this results in a layer-wise recursive structure: at layer l, the embedding computation depends on its in-neighbors from layer $l-1$, and so on. As shown in Figure 4(a), computing the embedding of Vertex 3 at Layer 1 in Worker 0 requires the Layer-0 embedding of Vertex 2.

Under resource fluctuations, the cost of processing cached dependencies becomes sensitive to system performance. In particular, contention on compute and memory resources affects vertex and edge operations, which is reflected in the runtime-aware per-dimensional costs T'_v and T'_e of processing each vertex and edge. Let $V^{(l)}_{\text{cache},i}$ and $E^{(l)}_{\text{cache},i}$ denote the cached vertices and edges at layer l on worker i. Since multi-hop dependencies are cached across layers, the overall DepCache cost for worker i is computed by accumulating costs across multiple layers:

$$T'_{\text{cache},i} = \sum_{l=1}^{L-1} \left(|V^{(l)}_{\text{cache},i}| \cdot T'_v + |E^{(l)}_{\text{cache},i}| \cdot T'_e \right) \cdot d^{(l)} \tag{5}$$

where $d^{(l)}$ is the feature dimension of layer l.

Impact of Resource Fluctuations on Local Subgraph Computation. Similar to DepCache computation, the local subgraph computation time under resource fluctuations is also composed of the multi-layer vertex and edge processing costs, as shown in the following equation:

$$T'_{\text{local},i} = \sum_{l=1}^{L} \left(|V^{(l)}_{\text{local},i}| \cdot T'_v + |E^{(l)}_{\text{local},i}| \cdot T'_e \right) \cdot d^{(l)} \tag{6}$$

where $|V^{(l)}_{\text{local},i}|$ and $|E^{(l)}_{\text{local},i}|$ represent the number of vertices and edges in the local partition of worker i.

Impact of Resource Fluctuations on DepComm. The DepComm cost arises from cross-partition dependency vertices when a vertex must fetch feature vectors from remote neighbors of other workers. As shown in Figure 4(a), computing the embedding of Vertex 2 on Worker 1 of Layer 1 requires accessing the embedding of its remote neighbor Vertex 1 located on Worker 0 of Layer 0. Such communication recurs across layers, as each GNN layer aggregates information from the remote neighbors. Under bandwidth fluctuations, the per-dimension communication cost between Worker i and Worker j is denoted by $T'_{c,ij}$. The total communication cost is given by:

$$T'_{\text{comm},i} = \sum_{l=1}^{L} \left(d^{(l-1)} \cdot \sum_{\substack{j=0 \\ j \neq i}}^{m-1} T'_{c,ij} \cdot |V^{(l-1)}_{\text{comm},ij}| \right) \tag{7}$$

where $|V^{(l-1)}_{\text{comm},ij}|$ denotes the number of dependency vertices transferred from worker j to worker i at layer $l-1$, and m denotes the total number of workers.

Optimization Objective. To support decentralized optimization in resource-fluctuating environments, each worker independently minimizes its own runtime-aware training time T'_i, which reflects the current execution delay under resource fluctuations. This is achieved by dynamically adjusting the assignment of dependency vertices between DepCache and DepComm, based on locally available resource. Moreover, the additional memory consumption for DepCache must respect the worker's memory constraint C_i. The optimization problem for worker i is formulated as:

$$\min \quad T'_i = T'_{\text{comm},i} + T'_{\text{cache},i} + T'_{\text{local},i} \tag{8}$$

$$V_{\text{cache},i} \cup V_{\text{comm},i} = V_{\text{depend},i} \tag{9}$$

$$V_{\text{cache},i} \cap V_{\text{comm},i} = \emptyset \tag{10}$$

$$\sum_{v \in V_{\text{cache},i}} s(v) \leq C_i \tag{11}$$

where $V_{\text{depend},i}$ denotes the complete set of dependency vertices required by worker i, and $s(v)$ denotes the storage cost of vertex v.

The storage cost $s(v)$ of a dependency vertex v is determined by the number of neighbor vertices and edges it introduces across layers when cached locally. Formally, we define:

$$s(v) = \sum_{l=1}^{L-1} \left(|V^{(l)}_i(v)| + |E^{(l)}_i(v)| \right) \cdot d^{(l)} \tag{12}$$

where $|V^{(l)}_i(v)|$ and $|E^{(l)}_i(v)|$ denote the number of vertices and edges introduced by v at layer l.

4.2 Lightweight Resource-aware Workload Adjustment

NP-hardness. We show that the dependency adjustment problem in our formulation is NP-hard by a polynomial-time reduction from the Cardinality-Constrained Knapsack Problem (CCKP). Given any CCKP instance, we map each item to a dependency; the item size maps to the storage cost, and the item profit maps to the reduction in runtime variation. The capacity constraints are preserved. Under this mapping, selecting items in CCKP to maximize total profit is equivalent to selecting dependency handling method to minimize runtime variation subject to the memory budget. Since this reduction is in polynomial time and CCKP is NP-hard, our problem is also NP-hard.

Lightweight Adaptive Adjustment Algorithm. We propose a workload adjustment algorithm to address this NP-hard problem. The algorithm dynamically adjusts the handling of the dependencies to align the computation and communication load of each worker with its available resources. As shown in Figure 4, when a worker faces limited CPU resources, more dependencies are retrieved using the DepComm approach (upper part of Figure 4(b)). Conversely, when a worker suffers from limited communication bandwidth, the DepCache approach is preferred (lower part of Figure 4(b)). The algorithm initially sets all dependency vertices to use DepComm, and then selects the vertices with the highest benefit to convert them to DepCache. The algorithm consists of two stages: preprocessing and online adjustment. The preprocessing stage is executed before training starts to cache essential data and prioritize dependency vertices through sorting by their computation

Algorithm 2 Preprocessing Phase

Input: vertex subset V_i, edge subset E_i; set of remote dependency vertices D; number of remote workers m; T_v, T_e.

Output: $\{\mathcal{V}_{i,j}\}_{j=1}^{m}$: remote dependency vertices owned by worker j, sorted by $T_{\text{cache},i,j}(v)$;
$\{\mathcal{N}_{i,j}\}_{j=1}^{m}$: arrays aligned with $\mathcal{V}_{i,j}$. For each v, $\mathcal{N}_{i,j}(v)$.V = # multi-hop neighbor vertices, $\mathcal{N}_{i,j}(v)$.E = # multi-hop neighbor edges, $\mathcal{N}_{i,j}(v)$.S = per-vertex storage cost;
$\{\mathcal{S}_{i,j}\}_{j=1}^{m}$: prefix-sum arrays of storage cost over $\mathcal{V}_{i,j}$, i.e., $S_{i,j}[k] = \sum_{t=1}^{k} \mathcal{N}_{i,j}(\mathcal{V}_{i,j}[t])$.S.

1: **for** $j = 1$ to m **do**
2: $\quad \mathcal{V}_{ij} \leftarrow \{\, v \in D, v \in V_j \,\}$
3: $\quad$ **for** each $v \in \mathcal{V}_j$ **do**
4: $\quad\quad T_{\text{cache},ij}(v) \leftarrow \mathcal{N}_{ij}(v).V \cdot T_v \; + \; \mathcal{N}_{ij}(v).E \cdot T_e$
5: $\quad$ SORT $(\mathcal{V}_{ij})$ by $T_{\text{cache},j}(v)$ *(ascending)*
6: $\quad$ REINDEX $\mathcal{N}_{i,j}$ accordingly
7: $\quad \mathcal{S}_{ij}[1] \leftarrow \mathcal{N}_{ij}(\mathcal{V}_{ij}[1]).S$
8: $\quad$ **for** $k = 2$ to $|\mathcal{V}_{ij}|$ **do**
9: $\quad\quad \mathcal{S}_{ij}[k] \leftarrow \mathcal{S}_{ij}[k-1] + \mathcal{N}_{ij}(\mathcal{V}_{ij}[k]).S$
10: **return** $\{\mathcal{V}_{i,j}\}_{j=1}^{m}, \{\mathcal{N}_{i,j}\}_{j=1}^{m}, \{\mathcal{S}_{i,j}\}_{j=1}^{m}$

overhead. The online adjustment stage dynamically adjusts dependency handling approaches based on real-time resource feedback. By offloading expensive operations to preprocessing, the online adjustment remains lightweight and efficient during training.

Preprocessing Phase. Each worker i precomputes two ordered arrays, $\mathcal{N}_i$ and $\mathcal{S}_i$. Array $\mathcal{N}_i$ groups remote dependency vertices by their owner j into $\mathcal{V}_{i,j}$ and, for each $v \in \mathcal{V}_{i,j}$, stores the counts of multi-hop neighbor vertices/edges and the per-vertex storage cost. Array $\mathcal{S}_i$ contains prefix-sum arrays $S_{i,j}$ of the storage costs for each $\mathcal{V}_{i,j}$. We group by owner because vertices from the same remote worker share the same per-feature-dimension communication time $T_{c,i,j}$ (Algorithm 2, Lines 1–2). Using the measured per-feature-dimension processing costs T_v and T_e, we compute $T_{\text{cache},i,j}(v)$ for $v \in \mathcal{V}_{i,j}$ (Line 4), sort $\mathcal{V}_{i,j}$ accordingly and reorder $\mathcal{N}_{i,j}$ (Lines 5–6), then build $S_{i,j}$ (Lines 7–9). These precomputations enable fast lookups during online adjustment. Although T_e and T_v may fluctuate, they typically scale proportionally; hence the relative order of $T_{\text{cache},i,j}(v)$ is empirically stable across epochs. This is because compute-bound (e.g., GEMM, related to T_v) and memory-bound (e.g., SpMM, related to T_e) operations degrade proportionally under GPU contention. We validate this proportional degradation through experiments, with detailed results available in the technical report [4] due to the page limit. Therefore, these arrays only need to be generated once before training, significantly reducing the adjustment overhead.

Online Adjustment Phase. During training, worker i adapts the handling of remote dependencies to resource fluctuations while keeping the *cached* set within budget C_i (Algorithm 3). For each owner j, since $T'_{\text{comm},i,j}$ is constant over $v \in \mathcal{V}_{i,j}$ and $\mathcal{V}_{i,j}$ is sorted by $T'_{\text{cache},i,j}(v)$, we locate the positive-benefit region $\gamma_{i,j}(v) = T'_{\text{comm},i,j} - T'_{\text{cache},i,j}(v) > 0$ by binary search (Lines 2–6). From the prefix-sum array $S_{i,j}[\cdot]$ we obtain its storage $s_{i,j} = S_{i,j}[k_j]$ (Line 7) and the total demand $S_i = \sum_{j=1}^{m} s_{i,j}$ (Line 9). If $S_i \le C_i$, convert all positive-benefit vertices to DepCache and leave the rest DepComm (Lines 12–13). Otherwise allocate budget proportionally,

Algorithm 3 Online Adjustment Phase

Input: $\{\mathcal{V}_{i,j}\}, \{\mathcal{N}_{i,j}\}, \{S_{i,j}\}$; updated costs $T'_v, T'_e, T'_{c,i,j}$; total storage constraint C_i; number of remote workers m.

Output: $\{\mathcal{V}_{i,j}^{\text{cache}}\}_{j=1}^{m}, \{\mathcal{V}_{i,j}^{\text{comm}}\}_{j=1}^{m}$.

1: **Initialize:** $\mathcal{V}_{i,j}^{\text{cache}} \leftarrow \emptyset,\ \mathcal{V}_{i,j}^{\text{comm}} \leftarrow \emptyset$ for $j = 1, \ldots, m$
2: **for** $j = 1$ to m **in parallel do**
3: $\quad$ **for** each $v \in \mathcal{V}_{i,j}$ **do**
4: $\quad\quad T'_{\text{cache},i,j}(v) \leftarrow \mathcal{N}_{i,j}(v).\text{V} \cdot T'_v \; + \; \mathcal{N}_{i,j}(v).\text{E} \cdot T'_e$
5: $\quad\quad \gamma_{i,j}(v) \leftarrow T'_{c,i,j} \; - \; T'_{\text{cache},i,j}(v)$
6: $\quad$ **BinarySearch** in $\mathcal{V}_{i,j}$ for largest index k_j s.t. $\gamma_{i,j}(v) > 0$
7: $\quad s_{i,j} \leftarrow S_{i,j}[k_j]$
8: $\quad$ // k_j is the last vertex with positive benefit.
9: $S_i \leftarrow \sum_{j=1}^{m} s_{i,j}$
10: **if** $S_i \le C_i$ **then**
11: $\quad$ **for** $j = 1$ to m **in parallel do**
12: $\quad\quad \mathcal{V}_{i,j}^{\text{cache}} \leftarrow$ first k_j items of $\mathcal{V}_{i,j}$
13: $\quad\quad \mathcal{V}_{i,j}^{\text{comm}} \leftarrow \mathcal{V}_{i,j} \setminus \mathcal{V}_{i,j}^{\text{cache}}$
14: **else**
15: $\quad$ **for** $j = 1$ to m **do**
16: $\quad\quad C_{i,j} \leftarrow (s_{i,j}/S_i) \cdot C_i$
17: $\quad$ **for** $j = 1$ to m **in parallel do**
18: $\quad\quad k'_j \leftarrow \max\{\, k \le k_j \mid S_{i,j}[k] \le C_{i,j} \,\}$
19: $\quad\quad \mathcal{V}_{i,j}^{\text{cache}} \leftarrow$ first k'_j items of $\mathcal{V}_{i,j}$
20: $\quad\quad \mathcal{V}_{i,j}^{\text{comm}} \leftarrow \mathcal{V}_{i,j} \setminus \mathcal{V}_{i,j}^{\text{cache}}$
21: **return** $\{\mathcal{V}_{i,j}^{\text{cache}}\}_{j=1}^{m},\ \{\mathcal{V}_{i,j}^{\text{comm}}\}_{j=1}^{m}$

$C_{i,j} = (s_{i,j}/S_i)\, C_i$ (Lines 15–16), and for each j take the largest prefix of $\mathcal{V}_{i,j}$ that fits $C_{i,j}$ using $S_{i,j}$ (Lines 18–20). This prioritizes high-benefit conversions under the memory constraint.

Complexity Analysis. We fuse benefit computation into the binary search to avoid extra passes. Each adjustment performs at most $(m-1)$ binary searches (one per remote owner $j \neq i$) over sorted lists $\mathcal{V}_{i,j}$. Let $N_{\text{dep}} = \sum_{j \neq i} |\mathcal{V}_{i,j}|$. The online time is $\sum_{j \neq i} O(\log |\mathcal{V}_{i,j}|) = O\big(m \log(N_{\text{dep}}/m)\big)$ in the balanced case, and $O(m \log N_{\text{dep}})$ in general. The preprocessing is a one-time step before training that sorts and builds prefix sums over N_{dep} items, costing $O(N_{\text{dep}} \log N_{\text{dep}})$.

Approximation guarantee. Our method achieves a $\frac{1}{2}$-approximation guarantee as it is equivalent to that of the default greedy algorithm of CCKP, which ranks items by their benefit-to-cost ratio $\gamma(v)/s(v)$ and selects the largest feasible prefix, achieving a worst-case $\frac{1}{2}$-approximation [19]. Our algorithm adopts a simplified strategy that ranks vertices by $\gamma(v)$ and selects the Top-K vertices under the memory budget. In our design, ranking by $\gamma(v)$ is effectively equivalent to ranking by $\gamma(v)/s(v)$, as vertices with higher $\gamma(v)$ typically induce lower neighborhood memory costs $s(v)$. This allows us to omit $s(v)$ from the ranking criterion.

5 DEPENDENCY-AWARE PARTIAL-REDUCE

5.1 Overall Workflow

Motivation. The previously proposed resource-aware workload adjustment strategy generally mitigates the impact of resource fluctuations. However, under extreme resource degradation (e.g., simultaneous drops in compute capacity and network bandwidth),

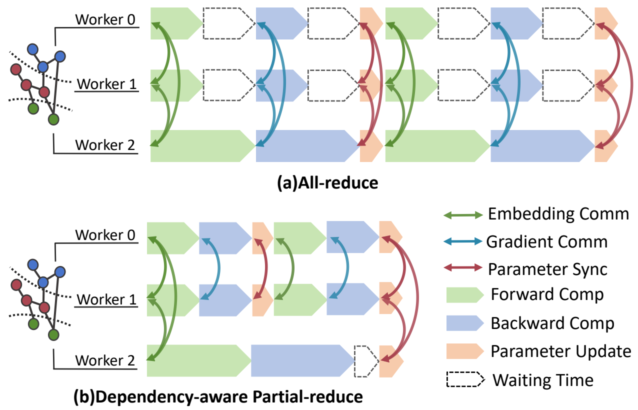


Figure 5: Partial-reduce for GNN. "Comm" indicates communication, "Comp" indicates computation, and "Sync" indicates synchronization.

some workers may lag significantly behind the rest of the cluster. When such degradation persists, workload adjustment alone may be insufficient to mitigate **severe stragglers**.

Challenges of Partial-reduce in GNNs. In distributed DNNs, partial-reduce (a variant of all-reduce) reduces synchronous overhead caused by severe stragglers by synchronizing gradients among only a subset of faster workers in each epoch. However, due to complex dependencies between training samples (vertices), partial-reduce faces the following challenges in distributed GNN training.

First, partial-reduce only works on the parameter synchronization stage. Yet workers must exchange embeddings and gradients of dependency vertices at every layer, creating implicit synchronization points. Thus, each epoch includes additional layer-wise synchronizations in addition to parameter synchronization.

Second, the number of workers involved in parameter updates may vary across iterations in partial-reduce, leading to unstable gradient scaling and slower convergence [8, 23, 33].In distributed DNN training, this is typically handled by weighted gradient averaging, with each worker's contribution proportional to its number of local training samples [59]. However, the number of training vertices for different workers is different, and the aggregation operation collects non-training vertices embeddings, which will also generate gradients for these non-training vertices during backward computation, as shown in Figure 6(a). Simply performing a weighted averaging of parameters across workers based on the number of training vertices may result in significant convergence bias.

A Two-stage Solution: Partial Computation and Partial Update. We propose a dependency-aware partial-reduce method for GNN, consisting of partial computation and partial update, as shown in Figure 5. Partial computation reuses historical vertex embeddings and gradients for forward/backward computation. Figure 5(b) illustrates that during the backward computation of the first epoch, Worker 0 and Worker 1 proceed without waiting for the gradients from the straggling Worker 2. Partial update collects the gradients from each worker based on corresponding weight values, which are computed from the number of training vertices participating in each layer. Specifically, a Breadth-First Search(BFS) traversal is initiated from the training vertices to collect the number

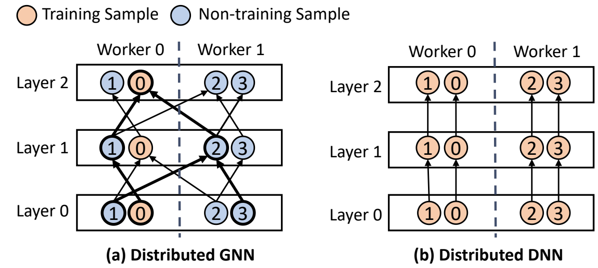


Figure 6: BFS-based partial update.

of neighbor vertices per worker at each layer. These counts are used to compute the normalization weights for gradient aggregation.

5.2 Partial Computation for Forward/Backward

We adopt a partial-computation mechanism in which workers use locally cached historical embeddings instead of waiting for remote embeddings when communication delays occur. In addition, we introduce a layer-wise timing constraint to limit the maximum runtime of each layer.

Concretely, for each GNN layer l, we estimate a baseline *runtime* $T_0^{(l)}$ from a short warm-up run of a few epochs and keep it fixed throughout training. We then set an adaptive slack $\Delta T_{\max}^{(l)} = \alpha T_0^{(l)}$, with $\alpha = 0.5$ by default, which serves as the maximum tolerable *waiting time* (elapsed runtime minus pure compute time) for remote vertex embeddings. Whenever a worker's elapsed time for layer l at an iteration exceeds $T_0^{(l)} + \Delta T_{\max}^{(l)}$, it stops waiting and instead uses *cached historical embeddings* for the unresolved vertices.

To ensure embedding version consistency, we design a dynamic staleness control mechanism to track the staleness of embeddings on each worker. The controller dynamically controls the maximum allowable delay for remote embedding synchronization. Specifically, when a worker's cached embeddings fall behind by more than K iterations, the system forces the worker to wait for the latest remote embeddings. This approach ensures that the embeddings used in the computation remain up-to-date, preventing approximation errors due to outdated embeddings.

5.3 Partial Update for Parameter Synchronization

Workflow of Partial Update. Partial update employs a dual mechanism of dynamic grouping and bounded staleness constraints, as shown in Figure 5(b). The lightweight controller monitors the per-iteration runtime (aggregated across all layers). Let T_0 denote the iteration-level baseline runtime (e.g., the sum of $T_0^{(l)}$ over layers) and let ΔT be the corresponding slack (e.g., αT_0). When the current iteration's elapsed time reaches $T_0 + \Delta T$, if the number of ready workers is $\geq M/2+1$ (M is the total number of workers), it triggers a weighted parameter synchronization within the temporary group of ready workers (e.g., workers 0–1 synchronize while bypassing worker 2). Otherwise, the controller continues waiting until the threshold is met.

To prevent parameter drift when some workers have not participated for a long time, all workers are required to join parameter synchronization every K iterations. The controller checks participation every K iterations; if any worker is absent, the next synchronization round is performed across all workers in the cluster.

BFS-based Gradient Weighting Strategy. To address convergence degradation caused by varying worker participation across iterations and the uneven distribution of training vertices in partial update, we propose a BFS-based gradient weighting strategy. The key idea is to normalize each worker's gradient contribution according to the number of training-related vertices.

Let $n_k^{(l)}$ denote the number of vertices contributing to gradients at layer l on worker k. These vertices are determined through a backward BFS traversal starting from the training vertices. Each worker computes its local normalized gradient as:

$$g_k^{(l)} = \frac{1}{n_k^{(l)}} \sum_{v \in \mathcal{V}_k^{(l)}} \nabla \mathcal{L}_v, \tag{13}$$

where $\mathcal{V}_k^{(l)}$ is the set of training-related vertices on worker k at layer l, and $\nabla \mathcal{L}_v$ is the gradient of the loss with respect to vertex v. The parameter update rule then becomes:

$$\theta_{t+1}^{(l)} = \theta_t^{(l)} - \eta \cdot \frac{\sum_{k \in \mathcal{S}_t} n_k^{(l)} \cdot g_k^{(l)}}{\sum_{k \in \mathcal{S}_t} n_k^{(l)}}, \tag{14}$$

where η is the global learning rate and $\mathcal{S}_t$ is the set of workers active at iteration t. This dependency-aware normalization adjusts each worker's contribution by the number of vertices involved in the backward computation (training vertices and their dependencies), stabilizes gradient scaling across iterations, and improves the convergence rate under dynamic worker participation.

Bias Elimination under Partial-reduce. We demonstrate that the proposed vertex-weighted gradient aggregation yields an unbiased estimate of the global gradient. We take expectations on both sides and assume the vertex-level gradients are unbiased, i.e., $\mathbb{E}[g_k^{(l)}] = \nabla \mathcal{L}(\theta_t)$, we have:

$$\begin{aligned} \mathbb{E}\left[\frac{\sum_{k \in \mathcal{S}_t} n_k^{(l)} \cdot g_k^{(l)}}{\sum_{k \in \mathcal{S}_t} n_k^{(l)}}\right] &= \frac{\sum_{k \in \mathcal{S}_t} n_k^{(l)} \cdot \mathbb{E}[g_k^{(l)}]}{\sum_{k \in \mathcal{S}_t} n_k^{(l)}} \\ &= \frac{\sum_{k \in \mathcal{S}_t} n_k^{(l)} \cdot \nabla \mathcal{L}(\theta_t)}{\sum_{k \in \mathcal{S}_t} n_k^{(l)}} = \nabla \mathcal{L}(\theta_t). \end{aligned} \tag{15}$$

In contrast, traditional GNN training directly sums local gradients across active workers without normalization, which leads to biased gradient estimates when only a subset of training vertices is covered at each iteration. The expected aggregated gradient becomes:

$$\mathbb{E}\left[\sum_{k \in \mathcal{S}_t} g_k^{(l)}\right] = \frac{N_t^{(l)}}{N^{(l)}} \nabla \mathcal{L}(\theta_t), \tag{16}$$

where $N^{(l)} = \sum_{k=1}^{K} n_k^{(l)}$ is the total number of training vertices at layer l, and $N_t^{(l)} = \sum_{k \in \mathcal{S}_t} n_k^{(l)}$ is the number of vertices contributing to gradients at layer l among all workers involved in gradient aggregation. The factor $N_t^{(l)}/N^{(l)}$ reflects incomplete training signal coverage and causes systematic underestimation of the true gradient.

Table 1: Dataset description.

Dataset	|V|	|E|	#𝔽	#𝕃	#ℍ
Yelp (YP)	716,874	13,954,819	300	100	128
Reddit (RDT)	232,965	114,615,892	602	41	128
Ogbn-products (OPT)	2,449,029	61,859,140	100	47	64
Amazon (AMZ)	1,598,960	132,169,734	200	107	128

Therefore, by incorporating dependency-aware normalization, our method eliminates this bias and ensures unbiased gradient estimation under resource fluctuation and imbalanced participation. The convergence of the method is formally established, and the detailed proof is provided in the technical report [4] due to space constraints.

6 EXPERIMENTS

6.1 Experimental Setup

Environments. Our experiments are conducted on Aliyun ECS cluster with 16 GPU nodes. Each node has 16 vCPUs, 155GB DRAM, and 1 NVIDIA Tesla T4 GPU, running Ubuntu 20.04 LTS OS. The network bandwidth is 10 Gbps.

Datasets and GNN Algorithms. Table 1 lists the four graph datasets used in our evaluation: Reddit [13] is based on user interactions in a social network. Ogbn-products [15] originate from similar relationships between products in an e-commerce platform. The Yelp [50] dataset is constructed from user reviews of local businesses. The Amazon [6] dataset is derived from product co-purchasing graphs, with nodes representing products and edges capturing frequently co-reviewed or co-purchased items. The vertex feature dimensions, the number of labels of datasets, and hidden layer dimensions are listed in Table 1. We use four popular GNN models, including Graph Convolutional Network (GCN) [21], Graph Attention Network (GAT) [39], GraphSAGE [13], and TAGCN [9] to evaluate the performance, all of them are in a 2-layer structure.

The Systems for Comparisons. In our experiments, we compare NeutronCloud with two types of systems: mini-batch systems and full-graph systems. For the mini-batch system, we choose DistDGL [55] as the baseline. DistDGL reduces computational and memory overhead through sampling. In our experiment, DistDGL samples up to 10 first-hop neighbors per seed node and up to 15 second-hop neighbors per first-hop neighbor. For the full-graph system, we compare NeutronCloud with NeutronStar [45] and Sancus [32].NeutronStar adopts a hybrid dependency-handling approach designed to balance computation and communication workload, enabling high-performance GNN training. Sancus uses historical embeddings to reduce cross-worker communication. In NeutronCloud, we follow the graph partitioning strategy used in NeutronStar, adopting a chunk-based [58] approach that divides the vertex ID space into contiguous ranges. Vertex features and labels are colocated with their corresponding vertices, while edges are assigned to partitions based on their destination vertex. By default, we set the staleness bound $K = 3$ in both accuracy and runtime performance evaluations, meaning that each worker is allowed to compute with cached embeddings and delay gradient synchronization for up to 3 epochs. All experimental results are

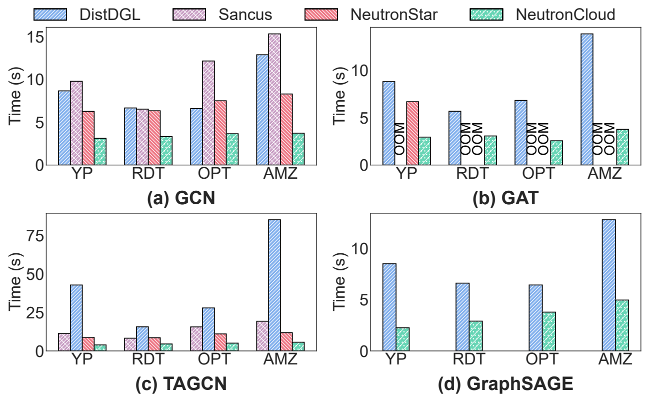


Figure 7: Per-epoch runtime of different systems on different models. "OOM" indicates the out-of-memory error.

presented in terms of the runtime per epoch, which refers to the time for forward and backward propagations, of all vertices in the graph. The results are obtained by averaging over 100 epochs. A shorter runtime per epoch indicates that the model takes less time to achieve the same accuracy.

Heterogeneity Simulation. Inspired by existing heterogeneous training methods [25–27, 56], we simulate a resource-fluctuating environment in the experiment. Specifically, for each worker, we independently add a certain probability of sleep time in each epoch to reflect resource dynamic. Specifically, each worker has a 10% probability of adding 5 seconds of sleep time within an epoch, which is partially added to both forward and backward propagations.

6.2 Overall Comparison

We compare the performance of NeutronCloud by running GCN, GAT, GraphSAGE, and TAGCN on a 16-node cluster. Recording the average runtime and the average increased training time caused by resource fluctuations (called **fluctuation-induced delay** time) per epoch. The experimental results are summarized in Figure 7 and Figure 8 respectively.

Per-epoch Time Comparison. Compared to NeutronStar, DistDGL, and Sancus, NeutronCloud demonstrates superior performance across all datasets. For the average runtime per epoch, it also achieves speedups of up to 2.10×, 2.97× and 4.15× (Figure 7).

Sancus periodically performs a full broadcast of the local embeddings of each worker to all workers, usually every few iterations, regardless of whether other partitions actually need them. This design introduces substantial redundant communication and lengthens waiting times, further degrading end-to-end performance. Thus bandwidth fluctuations exacerbate delays and harm training efficiency. NeutronStar adopts a pipeline parallelism strategy. In each worker, the workload is partitioned into multiple chunks, and chunk-level scheduling is applied to overlap the communication and compute tasks. It exhibits better adaptability and performance compared to Sancus and DistDGL in most datasets under resource-fluctuating environment.

The static workload allocation method used by DistDGL, NeutronStar and Sancus cannot match fluctuating available resources, leading to mismatches between workload and available resources. The advantages of NeutronCloud stem from two main factors: (1) the resource-aware workload adjustment strategy enables dynamic changes in workload to match the continuously changing resources;

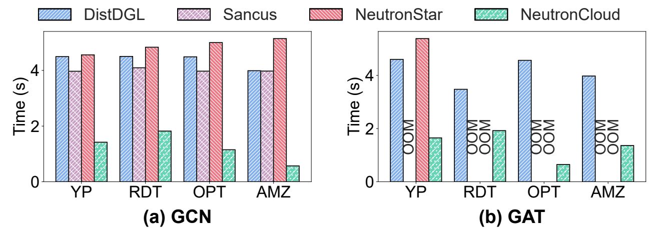


Figure 8: Fluctuation-induced delay time of different systems on GCN and GAT models. "OOM" indicates the out-of-memory error.

(2) the proposed dependency-aware partial-reduce method, which effectively alleviates the negative impact of extreme resource fluctuations on training efficiency.

Fluctuation-induced Delay Time Comparison. The fluctuation-induced delay time can directly reflect the impact of resource variability on distributed GNN training. For the average fluctuation-induced delay time per epoch, compared to DistDGL, NeutronStar, and Sancus, NeutronCloud achieves speedups of 3.89×, 4.81× and 3.87×, respectively, as shown in Figure 8. Sancus reduces communication overhead by reusing historical embeddings, making it less affected by communication resource fluctuations compared to DistDGL and NeutronStar.

6.3 Preprocessing Overhead Analysis

We compare the preprocessing time, including prioritizing dependent vertices through sorting and counting the number of training vertices, to the execution time of running GCN for 100 epochs. Table 2 shows the results. The preprocessing phase adds an average overhead of 3.91% to NeutronCloud. This overhead gradually decreases as the graph size increases (e.g., 5.66% on Yelp vs. 2.84% on Amazon). And this is typically amortized during training, while the techniques applied in this phase contribute to an average speedup of 2.08× for NeutronCloud.

Table 2: Preprocessing vs. training time breakdown (seconds and portion of total) for 100 epochs. Baseline denotes the training time before applying our approach.

Dataset	Training(Baseline)	Preprocessing	Training	Total
Yelp	675.00	18.74 / 5.66%	312.00 / 94.34%	330.74
Reddit	683.00	16.62 / 4.77%	332.00 / 95.23%	348.62
Products	750.00	8.94 / 2.39%	365.00 / 97.61%	373.94
Amazon	879.00	10.86 / 2.84%	372.00 / 97.16%	382.86

6.4 Scalability Analysis

Performance with varying cluster sizes. In this experiment, we compare NeutronCloud with other systems when training GCN on four datasets across varying cluster sizes. Figure 9 reports per-epoch runtime across cluster sizes, where NeutronCloud consistently outperforms other systems. Specifically, compared to DistDGL, NeutronStar, and Sancus, NeutronCloud achieves average speedup ratios of 2.53×-2.65× when scaling from 4 to 16 workers.

As the number of workers increases, the overall training time of distributed GNNs is expected to decrease. However, for full-graph systems such as Sancus and NeutronStar, the training time not only fails to reduce but instead increases due to resource fluctuations. In contrast, NeutronCloud experiences a slight reduction in training time because dynamic workload allocation methods

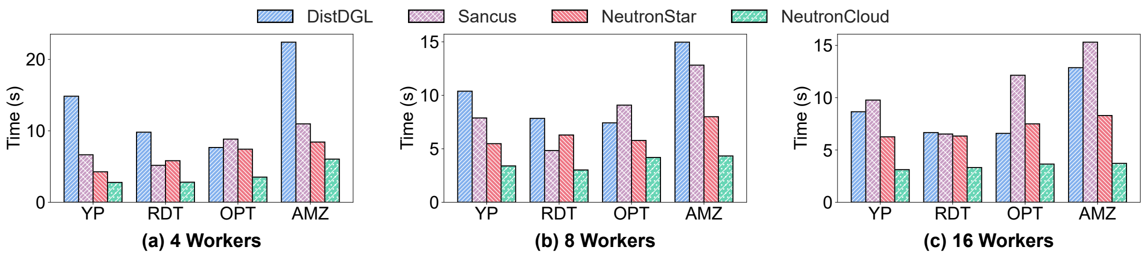


Figure 9: Per-epoch runtime of different systems with different cluster sizes on different datasets.

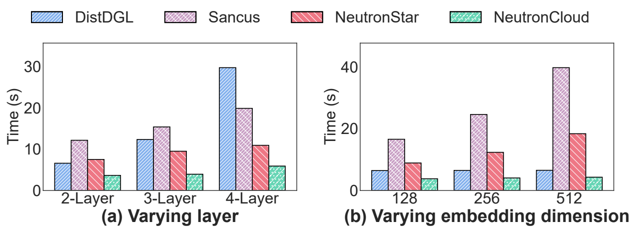


Figure 10: Per-epoch runtime of different systems on Ogbn-Products with varying model configurations.

can better match fluctuating available resources. Sancus exhibits poor scalability, possibly due to reliance on serial global broadcasting and broad, repeated embedding transfers. In large-scale clusters, growing workers amplify communication overhead and network bottlenecks, which limits Sancus's scalability. In contrast, NeutronCloud employs a dependency-aware partial-reduce method that better handles severe stragglers under resource fluctuations in large-scale clusters.

Performance with varying model layers. In this experiment, we compare NeutronCloud with baselines when training GCN with different model layers over Ogbn-products in a 16-node cluster. For the 2, 3, and 4-layer models, the DistDGL sampling strategies were set to (25,10), (25,15,10), and (25,20,15,10), respectively. The results are shown in Figure 10(a) We observe that the performance advantage of NeutronCloud over other baselines gradually increases with the model depths. For the 2-layer model, NeutronCloud achieves an average speedup of 2.40×. For the 3-layer and 4-layer models, the speedups were 3.15× and 3.42×, respectively.

Performance with varying embedding dimensions. In this experiment, we compare NeutronCloud with baselines when training GCN with different embedding dimensions on a 16-node cluster. As shown in Figure 10(b), NeutronCloud consistently outperforms baselines, and its advantage increases with the embedding size. Specifically, NeutronCloud achieves average speedups of 2.81×, 3.57×, and 5.02× over all baselines for 128-, 256-, and 512-dimensional embeddings, respectively. Larger embedding dimensions increase communication volume; NeutronCloud reduces this overhead by (i) limiting embedding exchange via partial-reduce and (ii) shifting dependency communication to local computation through resource-aware adjustment.

6.5 Performance Tolerance to Parameter Staleness (K)

We study GNN training's tolerance to parameter staleness. In our setting, historical embeddings are reused for computation, and only

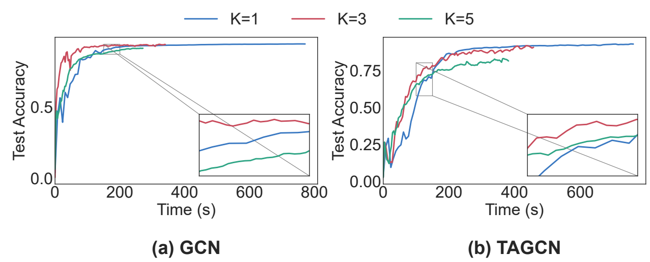


Figure 11: Accuracy for different parameters K.

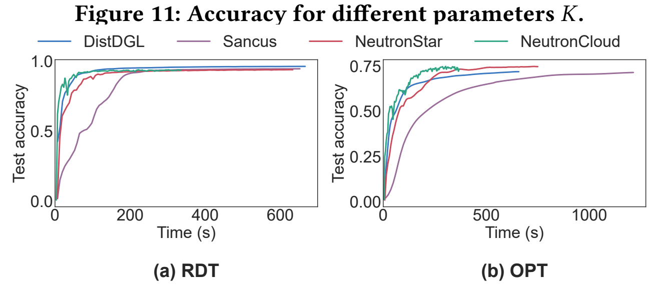


Figure 12: Time-to-accuracy.

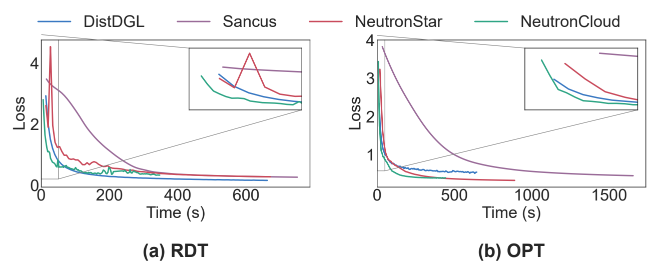


Figure 13: Time-to-loss.

a subset of gradients participates in parameter synchronization. We evaluate the convergence performance of the systems on a cluster with 16 workers and show the accuracy curves for the GNN models (GCN, TAGCN) on NeutronCloud, NeutronStar, DistDGL, and Sancus under different parameters K, as shown in Figure 11. Results indicate that increasing K reduces average per-epoch time, with a modest decrease in accuracy; when the value of parameter K is 3, the model achieves a relatively ideal balance between accuracy and training overhead.

6.6 Convergence Analysis

We evaluate the convergence of NeutronCloud, NeutronStar, DistDGL, and Sancus on a 16-worker cluster using a GCN model for node classification, and show the accuracy curves on two datasets in Figure 12.

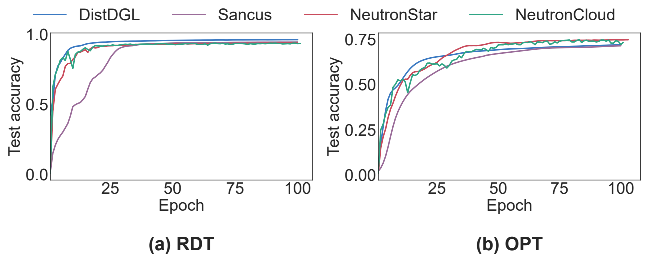


Figure 14: Epoch-to-accuracy.

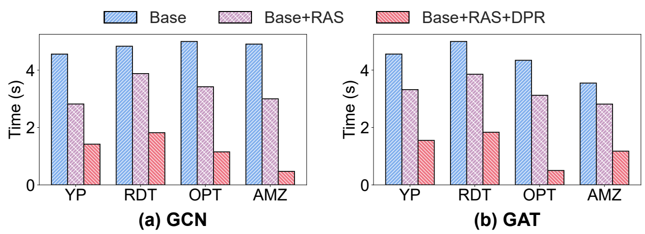


Figure 15: Performance gain analysis. "RAS" indicates the resource-aware workload adjustment strategy, "DPR" indicates the dependency-aware partial-reduce strategy.

The results show that in the early stages, NeutronCloud experiences accuracy fluctuations caused by severe stragglers. After 100 epochs, the test accuracy stabilizes, and NeutronCloud achieves the same test accuracy compared to other systems while reaching the target accuracy faster than all other systems. Similarly, Figure 13 presents the training loss over time. At convergence, NeutronCloud attains a training loss comparable to that of the baselines.

However, Sancus exhibits a long convergence time, which may be due to its cross-worker communication. It does not immediately transmit vertex embeddings after each update but instead transmits them at fixed epoch intervals. NeutronCloud employs a bounded control mechanism in partial computation, limiting the staleness of historical embeddings to at most K iterations, ensuring that deviation does not accumulate indefinitely. Additionally, periodic All-Reduce in partial update is performed, ensuring that all workers complete parameter synchronization within K epochs, preventing the model from degrading due to severe stragglers. As a result, NeutronCloud achieves higher accuracy.

Figure 14 shows the accuracy curves of the GCN model on NeutronStar, NeutronCloud, DistDGL, and Sancus across training epochs. We observe that under the same number of epochs, NeutronCloud achieves slightly lower model accuracy compared to other systems, but it eventually achieves convergence accuracy comparable to that of other systems. This is because it uses historical (stale) embeddings for forward computation and performs parameter synchronization only among a subset of workers in each iteration. However, since NeutronCloud executes each epoch faster, the system still achieves faster convergence in terms of overall training time.

6.7 Performance Gain Analysis

Training efficiency analysis. To validate the effectiveness of NeutronCloud's key designs, we conduct experiments on GNN models

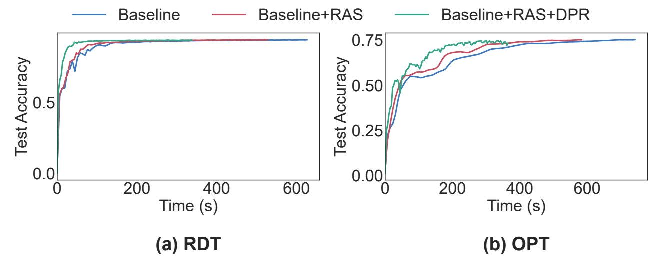


Figure 16: Test accuracy over time for the RDT and OPT datasets under three settings: Baseline, Baseline+RAS, and Baseline+RAS+DPR.

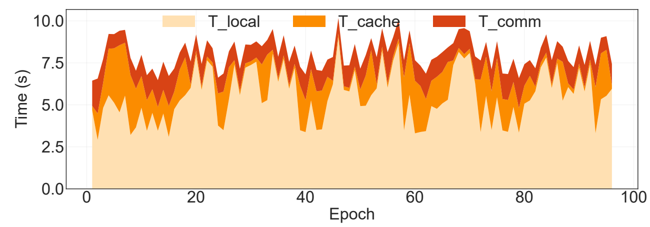


Figure 17: The distribution of overhead associated with T_local, T_cache, and T_comm during a 100-epoch training.

(GCN, GAT) across four datasets, evaluating the impact of resource-aware workload adjustment strategy (RAS) and dependency-aware partial-reduce (DPR) on system efficiency. To ensure a fair comparison, we start with a foundational framework established on the NeutronCloud codebase and gradually integrate the two optimization methods.

Figure 15 shows the average fluctuation-induced delay time per epoch. Compared to the `Baseline`, the `Baseline+RAS` achieves an average speedup of 1.40×. Figure 17 shows the per-epoch distribution of T_{local}, T_{cache}, and T_{comm} during a 100-epoch training of GCN on the Amazon dataset under `Baseline+RAS`. And the T_{local} includes the injected sleep time used to simulate resource fluctuations. By dynamically rebalancing the workload between DepCache and DepComm, NeutronCloud effectively mitigates the performance degradation introduced by these fluctuations.

Compared to the `Baseline+RAS`, the `Baseline+RAS+DPR` achieves an average speedup of 2.64×. The dependency-aware partial-reduce strategy includes two levels of mechanisms, reducing the impact of severe stragglers on overall training progress from both layer-wise synchronization and parameter synchronization dimensions under extreme conditions. It significantly reduces the prolonged synchronization overhead caused by severe stragglers.

Accuracy evaluation and analysis. To validate the effectiveness of NeutronCloud's designs, we evaluate the convergence performance of the `Baseline` and two optimization variants. Figure 16 shows the accuracy curves of the GCN model on the Reddit and Ogbn-products datasets. After 100 epochs, the test accuracy stabilizes, and both `Baseline+RAS+DPR` and `Baseline+RAS` achieve accuracy comparable to that of the `Baseline`.

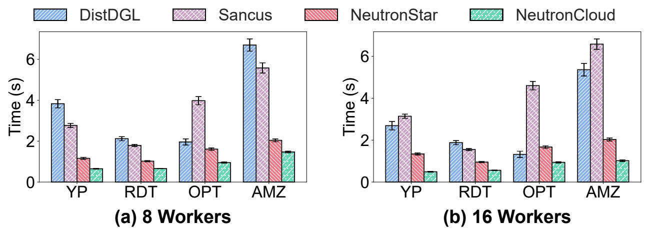


Figure 18: Per-epoch runtime comparison under real cloud heterogeneous setting.

6.8 Performance on Real Cloud GPUs

To evaluate our method in a real heterogeneous environment, we further conduct experiments on commercial cloud platforms that support GPU sharing. With the improvement of single-GPU performance, public GPU cloud providers have started offering virtual GPU containers, such as Vultr[40] and Alibaba Cloud[7], allowing multiple users to share the same physical GPU resources. By leveraging GPU virtualization technologies such as NVIDIA MIG[29] or SR-IOV[30], these platforms allocate independent computing instances to each user. These cloud providers typically allow users to request GPU resources on demand, renting computing power at a finer granularity (e.g., 1/2, 1/4, or even 1/20 of a single GPU). This approach improves the overall utilization of GPU resources while effectively reducing user costs.

Specifically, we launch 16 GPU instances on Alibaba GPU cloud provider to validate the adaptability of our method, each instance provisions a virtual GPU container with 1/3 of an NVIDIA A10 GPU and up to 20 Gbps network bandwidth. Since the computational performance of shared GPUs is affected by cluster scheduling policies, we observed significant dynamic heterogeneity among these instances. We run the GCN model on four datasets to compare the execution speed of NeutronCloud with other systems. To account for the variability of real cloud environments and ensure statistical reproducibility, we repeat each experiment multiple times across three independent periods and report the average per-epoch runtime. Figure 18 presents the experimental results with error bars that indicate the standard deviation between numerous experiments. The result demonstrates that NeutronCloud achieves stable performance across repeated trials, confirming the reproducibility of our results. Compared to DistDGL, NeutronStar, and Sancus, NeutronCloud achieves average speedup ratios of 3.90×, 1.83×, and 4.43×, respectively.

7 RELATED WORK

Existing Distributed GNN Systems. Several distributed GNN training systems have been proposed to address the scalability challenge of graph learning. Sancus [32] reduces synchronization overhead by using historical embeddings, but adopts a broadcast-based communication scheme where each worker sequentially sends all local embeddings to all others, regardless of actual demand. This incurs redundant communication and prolonged synchronization delays. NeutronStar [45] performs full-graph training with pipeline parallelism and chunk-level scheduling to accelerate training by overlapping communication and computation. However, it cannot adapt to environments with fluctuating resources due to the parameter synchronization approach of All-Reduce and static load distribution. DistDGL [55] adopts mini-batch training and reduces overhead through neighborhood sampling. However, its reliance on static METIS [18] partitioning can cause load imbalance under resource fluctuations, leading to unstable performance in resource-dynamic environments.

The Partial-Reduce Strategy for Distributed DNNs. Deploying deep learning workloads in heterogeneous environments (e.g., public clouds) exacerbates communication latency and straggler effects. Synchronous All-Reduce, originally designed for homogeneous clusters, becomes a performance bottleneck under such heterogeneity. To mitigate stragglers, recent work relaxes strict synchronization: *partial-reduce* [26] skips delayed workers during gradient aggregation; *DPAR* [20] adapts synchronization to node capacity; and *RNA* [48] allows nodes to update at their own pace. These relaxations reduce computation and communication overhead and improve throughput for distributed DNNs. While effective for distributed DNNs, these techniques cannot be directly applied to GNNs due to inter-sample dependencies between workers.

Distributed DNNs in Resource-fluctuating Environments. In multi-tenant cloud environments and shared clusters, compute and network capacity fluctuate over time, making fixed task assignment and strict synchronization ineffective. SDPipe [27] introduces a semidecentralized framework that couples heterogeneity-aware scheduling with dynamic gradient synchronization, adapting computation and communication to instantaneous worker capacity. This reduces straggler-induced stalls and synchronization latency and improves pipeline utilization for model-parallel DNNs. However, the design does not carry over to GNNs: models are small and trained in a data-parallel regime, and the dominant cost is dependency-driven communication (cross-partition neighbor-feature retrieval and consistency), not gradient synchronization. Consequently, SDPipe yields limited benefit for GNNs unless combined with mechanisms that reduce dependency communication.

8 CONCLUSION

We present NeutronCloud, a system designed for efficient GNN training in cloud environments with dynamic and fluctuating resources. Its performance and adaptability are enabled by two key components: (1) a resource-aware workload adjustment strategy that dynamically matches computational and communication workloads to real-time resource conditions, and (2) a dependency-aware partial-reduce strategy that reuses historical vertex embeddings and skips stragglers during gradient aggregation to improve training efficiency. Compared with existing distributed GNN systems such as DistDGL and Sancus, NeutronCloud achieves end-to-end training speedups ranging from 1.83× to 4.43× in the real cloud environments.

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China (2023YFB4503601), the National Natural Science Foundation of China (62461146205, U2241212, 62572108), the Distinguished Youth Foundation of Liaoning Province (2024021148-JH3/501), and the Liaoning Province Key R&D Program Project (2019JH2/10100027). Yanfeng Zhang is the corresponding author.

REFERENCES

[1] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: scalable, low-cost training of massive deep learning models. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. ACM, 472–487.

[2] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: an efficient communication library for distributed GNN training. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. ACM, 130–144.

[3] Chunyu Cao, Xin Ai, Qiange Wang, Yanfeng Zhang, Zhenbo Fu, Hao Yuan, Mingyi Cao, Chaoyi Chen, Yingyou Wen, Yu Gu, and Ge Yu. 2025. NeutronHeter: Optimizing Distributed Graph Neural Network Training for Heterogeneous Clusters. *Proc. ACM Manag. Data* 3, 4, Article 257 (Sept. 2025), 29 pages. https://doi.org/10.1145/3749175

[4] Mingyi Cao, Chunyu Cao, Yanfeng Zhang, Zhenbo Fu, Xin Ai, Qiange Wang, Yu Gu, and Ge Yu. 2025. NeutronCloud: Resource-Aware Distributed GNN Training in Fluctuating Cloud Environments (Technical Report). https://github.com/Toaoc/NeutronCloud/blob/main/NeutronDyn_TechnicalReport.pdf. Technical Report.

[5] Sandro Cavallari, Vincent W. Zheng, Hongyun Cai, Kevin Chen-Chuan Chang, and Erik Cambria. 2017. Learning Community Embedding with Community Detection and Node Embedding on Graphs. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*. ACM, 377–386.

[6] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 257–266.

[7] Alibaba Cloud. 2025. VGPU Accelerated Instance Families - Elastic GPU Service. https://www.alibabacloud.com/help/en/elastic-gpu-service/latest/vgpu-accelerated-instance-families. Accessed: 2025-03-31.

[8] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. 1232–1240.

[9] Jian Du, Shanghang Zhang, Guanhang Wu, José M. F. Moura, and Soummya Kar. 2017. Topology adaptive graph convolutional networks. *CoRR* abs/1710.10370 (2017).

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

[11] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Yihong Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph Neural Networks for Social Recommendation. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 417–426.

[12] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 551–568.

[13] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. 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*. 1024–1034.

[14] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2016. Addressing the straggler problem for iterative convergent parallel ML. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 98–111.

[15] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

[16] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and bridging the gaps in current GNN performance optimizations. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*. ACM, 119–132.

[17] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of the Third Conference on Machine Learning and Systems, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org.

[18] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (1998), 359–392.

[19] Samir Khuller, Anna Moss, and Joseph Naor. 1999. The Budgeted Maximum Coverage Problem. *Inf. Process. Lett.* 70, 1 (1999), 39–45.

[20] HyungJun Kim, Chunggeon Song, HwaMin Lee, and Heonchang Yu. 2023. Addressing Straggler Problem Through Dynamic Partial All-Reduce for Distributed Deep Learning in Heterogeneous GPU Clusters. In *IEEE International Conference on Consumer Electronics, ICCE 2023, Las Vegas, NV, USA, January 6-8, 2023*. IEEE, 1–6.

[21] Thomas N. Kipf and Max Welling. 2017. 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*. OpenReview.net.

[22] Jérôme Kunegis and Andreas Lommatzsch. 2009. Learning spectral graph transformations for link prediction. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009 (ACM International Conference Proceeding Series)*, Vol. 382. ACM, 561–568.

[23] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. USENIX Association, 583–598.

[24] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN training on large graphs via computation-aware caching. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*. ACM, 401–415.

[25] Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. 2020. Prague: High-Performance Heterogeneity-Aware Asynchronous Decentralized Training. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. ACM, 401–416.

[26] Xupeng Miao, Xiaonan Nie, Yingxia Shao, Zhi Yang, Jiawei Jiang, Lingxiao Ma, and Bin Cui. 2021. Heterogeneity-Aware Distributed Machine Learning Training via Partial Reduce. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2262–2270.

[27] Xupeng Miao, Yining Shi, Zhi Yang, Bin Cui, and Zhihao Jia. 2023. SDPipe: A Semi-Decentralized Framework for Heterogeneity-aware Pipeline-parallel Training. *Proc. VLDB Endow.* 16, 9 (2023), 2354–2363.

[28] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. 2021. Synergy: Resource Sensitive DNN Scheduling in Multi-Tenant Clusters. *CoRR* abs/2110.06073 (2021).

[29] NVIDIA. 2025. Multi-Instance GPU (MIG) Technology. https://www.nvidia.com/en-sg/technologies/multi-instance-gpu/. Accessed: 2025-03-31.

[30] PCI-SIG. 2010. Single Root I/O Virtualization (SR-IOV) Specification. https://pcisig.com/specifications/iov. Accessed: 2025-03-31.

[31] Hao Peng, Jianxin Li, Yu He, Yaopeng Liu, Mengjiao Bao, Lihong Wang, Yangqiu Song, and Qiang Yang. 2018. Large-Scale Hierarchical Text Classification with Recursively Regularized Deep Graph-CNN. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*. ACM, 1063–1072.

[32] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2023. Sancus: Staleness-Aware Communication-Avoiding Full-Graph Decentralized Training in Large-Scale Graph Neural Networks (Extended Abstract). In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*. ijcai.org, 6480–6485.

[33] Benjamin Recht, Christopher Ré, Stephen J. Wright, and Feng Niu. 2011. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*. 693–701.

[34] Jaeyong Song, Hongsun Jang, Hunseong Lim, Jaewon Jung, Youngsok Kim, and Jinho Lee. 2024. GraNNDis: Fast Distributed Graph Neural Network Training Framework for Multi-Server Clusters. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques, PACT 2024, Long Beach, CA, USA, October 14-16, 2024*. ACM, 91–107.

[35] Yujia Song, Ruyue Xin, Peng Chen, Rui Zhang, Juan Chen, and Zhiming Zhao. 2023. Identifying performance anomalies in fluctuating cloud environments: A robust correlative-GNN-based explainable approach. *Future Gener. Comput. Syst.* 145 (2023), 77–86.

[36] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2023. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*. USENIX Association, 497–513.

[37] Alok Tripathy, Katherine A. Yelick, and Aydin Buluç. 2020. 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*. IEEE/ACM, 70.

[38] Md. Vasimuddin, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj D. Kalamkar, Nesreen K. Ahmed, and

Sasikanth Avancha. 2021. DistGNN: scalable distributed training for large-scale graph neural networks. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021.* ACM, 76.

[39] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings.* OpenReview.net.

[40] Vultr. 2025. Vultr: High Performance SSD Cloud. Retrieved from https://www.vultr.com. Accessed: 2025-03-31.

[41] Cheng Wan, Youjie Li, Cameron R. Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. 2022. PipeGCN: Efficient Full-Graph Training of Graph Convolutional Networks with Pipelined Feature Communication. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022.*

[42] Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. 2023. Scalable and Efficient Full-Graph GNN Training for Large Graphs. *Proc. ACM Manag. Data* 1, 2 (2023), 143:1–143:23.

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

[44] Qiange Wang, Yao Chen, Weng-Fai Wong, and Bingsheng He. 2023. HongTu: Scalable Full-Graph GNN Training on Multiple GPUs. *Proc. ACM Manag. Data* 1, 4 (2023), 246:1–246:27.

[45] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. NeutronStar: Distributed GNN Training with Hybrid Dependency Management. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022.* ACM, 1301–1315.

[46] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin J. Barker, Ang Li, and Yufei Ding. 2023. MGG: Accelerating Graph Neural Networks with Fine-Grained Intra-Kernel Communication-Computation Pipelining on Multi-GPU Platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023.* USENIX Association, 779–795.

[47] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022.* USENIX Association, 945–960.

[48] Donglin Yang, Wei Rang, and Dazhao Cheng. 2020. Mitigating Stragglers in the Decentralized Training on Heterogeneous Clusters. In *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020.* ACM, 386–399.

[49] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018.* ACM, 974–983.

[50] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020.* OpenReview.net.

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

[52] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada.* 5171–5181.

[53] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. 2022. Multi-resource interleaving for deep learning training. In *SIGCOMM '22: ACM SIGCOMM 2022 Conference, Amsterdam, The Netherlands, August 22 - 26, 2022.* ACM, 428–440.

[54] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. 2022. ByteGNN: Efficient Graph Neural Network Training at Large Scale. *Proc. VLDB Endow.* 15, 6 (2022), 1228–1242.

[55] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. 2022. Distributed Hybrid CPU and GPU training for Graph Neural Networks on Billion-Scale Heterogeneous Graphs. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022.* ACM, 4582–4591.

[56] Qihua Zhou, Song Guo, Haodong Lu, Li Li, Minyi Guo, Yanfei Sun, and Kun Wang. 2021. Falcon: Addressing Stragglers in Heterogeneous Parameter Server Via Multiple Parallelism. *IEEE Trans. Comput.* 70, 1 (2021), 139–155.

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

[58] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. 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*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 301–316.

[59] Martin Zinkevich, Markus Weimer, Alexander J. Smola, and Lihong Li. 2010. Parallelized Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6-9 December 2010, Vancouver, British Columbia, Canada.* Curran Associates, Inc., 2595–2603.