NeutronTask: Scalable and Efficient Multi-GPU GNN Training with Task Parallelism

Zhenbo Fu*
Northeastern Univ, China
fuzhenbo@
stumail.neu.edu.cn

Xin Ai*
Northeastern Univ, China
aixin0@
stumail.neu.edu.cn

Qiange Wang
National University of
Singapore, Singapore
wang.qg@nus.edu.sg

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

Shizhan Lu
Northeastern Univ, China
lushizhan@
stumail.neu.edu.cn

Chaoyi Chen
Northeastern Univ, China
chenchaoy@
stumail.neu.edu.cn

Chunyu Cao
Northeastern Univ, China
caochunyu@
stumail.neu.edu.cn

Hao Yuan
Northeastern Univ, China
yuanhao@
stumail.neu.edu.cn

Zhewei Wei
Renmin University
of China, China
zhewei@ruc.edu.cn

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

Yingyou Wen
Neusoft AI
Research, China
wenyy@neusoft.com

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

ABSTRACT

Graph neural networks (GNNs) have emerged as a promising method for learning from graph data, but large-scale GNN training requires extensive memory and computation resources. To address this, researchers have proposed using multi-GPU processing, which partitions graph data across GPUs for parallel training. However, vertex dependencies in multi-GPU GNN training lead to significant neighbor replications across GPUs, increasing memory consumption. The substantial intermediate data generated during training further exacerbates this issue. Neighbor replication and intermediate data constitute the primary memory consumption in GNN training (i.e., typically accounting for over 80%). In this work, we propose GNN task parallelism for multi-GPU GNN training, which reduces neighbor replication by partitioning training tasks in each layer across different GPUs rather than partitioning the graph structure. This approach only partitions the graph data within individual GPUs, reducing the memory requirements of single tasks while overlapping subgraph computation across different GPUs. Shared neighbor embeddings among different subgraphs can be efficiently reused within a single GPU. Additionally, we employ a task-decoupled GNN training framework, which decouples different training tasks to manage their associated intermediate data independently and release it as early as possible to reduce memory usage. By integrating these techniques, we propose a multi-GPU GNN training system, NeutronTask. Experimental results on a 4×A5000 GPU server show that NeutronTask effectively supports billion-scale full-graph GNN training. For small graphs where the training data fits into the GPUs, NeutronTask achieves 1.27× - 5.47× speedup compared to state-of-the-art GNN systems including NeutronStar and Sancus.

PVLDB Reference Format:
Zhenbo Fu, Xin Ai, Qiange Wang, Yanfeng Zhang, Shizhan Lu, Chaoyi Chen, Chunyu Cao, Hao Yuan, Zhewei Wei, Yu Gu, Yingyou Wen, Ge Yu. NeutronTask: Scalable and Efficient Multi-GPU GNN Training with Task Parallelism. PVLDB, 18(6): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:

Table 1: Memory consumption of graph topology (Topo), feature (Feat), model parameters (Params), and intermediate data (Intr) for a 3-layer GCN training.

Dataset	#hidden	Params	Topo	Feat	Intr
ogbn-products	256	0.4MB	0.24GB	0.9GB	10.68GB
ogbn-papers100M	128	0.25MB	6.4GB	52.97GB	335.95GB

The source code, data, and/or other artifacts have been made available at https://github.com/iDC-NEU/NeutronTask.

1 INTRODUCTION

Graph Neural Networks (GNNs) have demonstrated superior performance across various practical applications such as social networks [29, 68, 75], recommendation systems [15, 67, 74], financial fraud detection [61], protein structure analysis [18], drug prediction [37, 49], traffic forecasting [6, 28], and natural language processing [42, 73]. GNN training consists of two tasks in each model layer: **embedding transformation** (T) and **graph propagation** (P). Each vertex applies T to update vertex embeddings and uses P to propagate neighbor embeddings. By iteratively executing these two tasks, GNN can learn rich structural information of data samples.

Recently, full-graph GNN training has emerged as a promising GNN training method as it provides more stable model quality compared to mini-batch training [27, 52, 55, 60, 63–65, 77]. However, full-graph GNN training requires extensive memory and computational resources [60, 63]. Considering the continuously increasing size of real-world graphs, researchers have proposed employing multi-GPU processing to meet the resource-intensive requirements of large-scale GNN training [8, 9, 19, 27, 43, 54, 65, 71]. These systems typically employ data parallelism, partitioning the input graph

*Equal contribution.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.
doi:XX.XX/XXX.XX

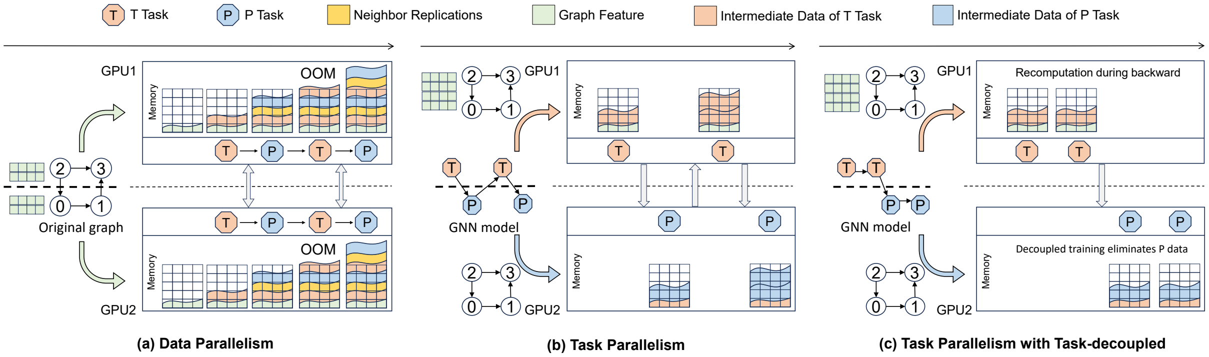

T Task
P Task
Neighbor Replications
Graph Feature
Intermediate Data of T Task
Intermediate Data of P Task
GPU1
GPU2
OOM
Memory
Original graph
GNN model
Recomputation during backward
Decoupled training eliminates P data
(a) Data Parallelism
(b) Task Parallelism
(c) Task Parallelism with Task-decoupled

Figure 1: GNN data parallelism vs. GNN task parallelism. The square matrix inside the GPU represents GPU memory space, and the arrows between GPUs represent communication. (a) GNN data parallelism requires caching neighbor replications and intermediate data to complete backward propagation, leading to out-of-memory (OOM) errors. (b) GNN task parallelism partitions training tasks across GPUs rather than partitioning the graph, reducing neighbor replications. (c) GNN task parallelism with task-decoupled breaks the alternated execution of the T and P tasks, releasing the intermediate data of P tasks in advance.

to enable parallel computation across multiple GPUs and handle remote neighbor aggregation by caching neighbor replicas. Despite the significant performance improvements achieved by multi-GPU processing, handling large-scale graphs remains a challenge due to the substantial disparity between the limited memory capacity of GPUs and the extensive memory consumption. As shown in Figure 1(a), this memory consumption primarily comes from neighbor replication (in yellow) and intermediate data (in orange and blue), accumulating with the increasing number of GNN tasks.

The neighbor replication arises from inter-GPU graph partition in data parallelism. When an entire graph is partitioned across multiple GPUs, the common neighbors between subgraphs must be stored multiple times on different GPUs. The neighbor replication increases rapidly with the number of graph partitions. Taking the ogbn-papers100M dataset as an example, when the number of partitions reaches 8, the size of neighbor replication increases by 200.1GB. Since each model layer requires additional memory to cache neighbor embeddings, the memory consumption associated with neighbor replication increases with the deeper model layers.

The intermediate data includes the vertex embeddings of various tasks [63]. During forward propagation, each model layer generates vertex embeddings, which are cached and consumed in the gradient computation of each layer during backward propagation. In GNN models, T tasks and P tasks are executed alternately. Although linear P tasks is not directly involved in parameter updates, the input of T tasks is the output of P tasks, and the output of P tasks is required when calculating the gradient of the parameter in T tasks. Therefore, the vertex embeddings of T tasks and P tasks both need to be cached. We conduct experiments using NeutronStar [64] in a single-CPU environment to evaluate the memory consumption of intermediate data. As shown in Table 1, for a 3-layer GCN training, the intermediate data averages 86.38% of total memory usage.

In this paper, we reduce the memory consumption caused by neighbor replication and intermediate data through two key techniques. Firstly, we propose task parallelism for GNN training to reduce neighbor replications through inter-GPU task partition and intra-GPU graph partition. As illustrated in Figure 1(b), GNN task parallelism partitions T and P tasks in each model layer across different GPUs, such that each GPU only needs to handle one type of task. Intra-GPU graph partition splits each training task over the entire graph into smaller processing units (subgraphs), ensuring each task can be sequentially scheduled on a single GPU, without exhausting the GPU memory. Concurrently, the subgraph computation can be overlapped across different GPUs to enhance parallelism. The shared neighbor embeddings among different subgraphs can be efficiently reused within a single GPU. As a result, GNN task parallelism effectively reduces neighbor replication by avoiding inter-GPU graph partition. Secondly, we propose a task-decoupled GNN training framework to reduce the memory consumption of intermediate data. The framework transforms the alternating execution mode of T tasks and P tasks into a sequence where continuous P tasks follow continuous T tasks. As illustrated in Figure 1(c), this framework not only reduces intermediate data generated by linear P tasks but also facilitates the use of traditional DNN recomputation techniques [12] to reduce the runtime memory consumption of intermediate data. In addition, the framework offers a flexible task scheduling model that allocates different numbers of GPUs to T and P tasks to meet the different resource requirements of T and P tasks. By integrating the above techniques, we propose NeutronTask, a multi-GPU system for full-graph GNN training. The experimental results on a 4-GPU (A5000) server demonstrate that NeutronTask can train large-scale graphs and achieve performance improvements compared to Sancus [43] and NeutronStar [64].

Our primary contributions are summarized as follows:

- We propose GNN task parallelism, which reduces neighbor replication by partitioning tasks across GPUs and partitioning graph of each task within a GPU, instead of partitioning graph data across GPUs.
- We propose a task-decoupled GNN framework that decouples T and P tasks. Then, we utilize the recomputation technique and flexible resource allocation to improve memory efficiency.
- We develop NeutronTask, a multi-GPU accelerated system that reduces memory usage by 49% − 67% and achieves 1.27 × −5.47× speedup compared to the state-of-the-art GNN training system.

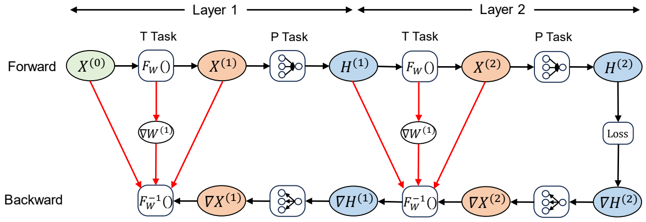


Figure 2: The data flow of a 2-layer GCN. The ellipses represent vertex embeddings at different stages, which need to be cached in the GPU memory. The red arrows connect to the dependent data, representing parameter gradient computations in backward propagation.

2 PRELIMINARIES

2.1 Graph Neural Network

Graph data are utilized to manage real-world data due to their ability to express entities and their relationships efficiently [3, 14, 20–22, 36, 39, 57]. As input to GNNs, graph data can be represented as $G = (V, E)$, where V and E represent vertex and edge sets, respectively, and each vertex contains a feature vector X_v, where $(v \in V)$.

GNN Models. GNN models consist of multiple layers, which generate embeddings by leveraging the structural and feature information of the graphs. Each layer of the GNN model consists of T and P tasks. T tasks apply neural networks to extract the information of vertices or edges, generating updated embeddings. P tasks include scattering vertex embeddings to edges or neighbors. We formalize these tasks using the aggregate-update computation pattern:

$$X_v^{(l)} = UPDATE(X_v^{(l-1)}, W^{(l)}), \tag{1}$$

$$H_v^{(l)} = AGGREGATE(\{X_u^{(l)} | \forall u \in N_{in}(v)\}), \tag{2}$$

where $X_v^{(l-1)}$ and X_v^l represents the embedding of vertex v in the $(l-1)$-th layer and l-th layer respectively. $X_v^{(0)}$ represents the input feature of vertex v. $N_{in}(v)$ represents the incoming neighbors of vertex v. Model parameters exist only in T tasks, while P tasks are responsible solely for propagating and aggregating vertex information. The model parameters are updated through gradient computation during the backward propagation.

Different GNN models are characterized by different functions of $UPDATE$ and $AGGREGATE$. For instance, GCN [29] aggregates neighbor embeddings by the Symmetric Normalized Laplacian in P tasks and uses neural networks to update vertex embeddings in T tasks. The formula for a single-layer GCN is as follows:

$$H^{(l+1)} = \sigma\Big(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}W^{(l)}\Big), \tag{3}$$

where σ represents the activation function, $W^{(l)}$ represents the neural network parameter of the l-th layer. The $\tilde{A}$ and $\tilde{D}$ correspond to $A + I$ and $D + I$, where A represents the adjacency matrix, D represents the degree matrix and $D_{ii} = \sum^j A^{ij}$.

Intermediate Data in GNN Training. Intermediate data in full-graph GNN training refers to the inputs and outputs of T and P tasks for each GNN layer, which is generated during the forward propagation and cached to compute gradients for T tasks during backward propagation [63]. Figure 2 is the data flow in a 2-layer GCN [29] model. In forward propagation, each vertex updates its embedding to generate updated results (*i.e.,* $X^{(i)}$) by the T task, which generates intermediate data of the neural network model for gradient computation. Then, each vertex aggregates neighbor embeddings to generate aggregation results (*i.e.,* $H^{(i)}$). Both neighbor embeddings and aggregated results need to be cached. The final layer's vertex embeddings (*i.e.,* $H^{(2)}$) compute the loss value based on truth data labels, generating gradient vectors (*i.e.,* $\nabla H^{(2)}$), whose dimensions match those of the vertex embeddings. Backward propagation starts from the final layer to the first, computing model parameter gradients using vertex gradients and embeddings. Each layer applies the chain rule for vertex gradient computation, including gradient propagation and derivative computation. Intermediate data from forward propagation must be retained until gradient computation for the corresponding backward layer is completed.

Table 2: Neighbor replication factor (α) and total memory consumption (MC) in the ogbn-papers100M dataset under varying numbers of partitions (3-layer GCN).

Partitions	1	2	4	8	16	32	64
NR	1	1.25	1.52	2.13	3.02	4.46	6.34
MC (/GB)	335.95	380.22	428.03	536.05	693.65	948.65	1281.56

2.2 Multi-GPU GNN Systems with Data Parallelism

Given the resource-intensive requirement of large-scale GNN training, researchers have employed multi-GPU processing to accelerate the training process. As shown in Figure 3, a common approach to scaling GNN training across multiple GPUs is data parallelism, which partitions the graph data across multiple GPUs to enable parallel training. During GNN training, P tasks may require access to neighbor vertices located on remote GPUs, leading to substantial vertex dependencies (the dashed vertices in Figure 3). To manage these dependencies, GNN data parallelism involves inter-GPU communication to fetch remote vertex data, followed by creating local replicas of these remote vertices to facilitate local computation.

Neighbor replication refers to each GPU creating replicas of neighbors from remote GPUs, which increases memory consumption. We evaluate the memory consumption of neighbor replication in the ogbn-papers100M dataset under varying numbers of partitions. We quantify the size of neighbor replication using the replication factor α, defined as the average number of replicas per vertex. The results are presented in Table 2, where the scale of neighbor replication rises rapidly with the number of partitions. As the number of partitions expands from 1 to 64, neighbor replication causes the total memory consumption to increase by 3.81×.

Despite multi-GPU platforms offering increased memory availability, the substantial volume of neighbor replication and intermediate data still hinders the effectiveness and efficiency of GNN data parallelism in handling large-scale GNN training. For real-world graphs, the data often exceeds the memory capacity even with multiple GPUs. As shown in Table 2, training a 3-layer GCN model on the ogbn-papers100M dataset requires at least 428.03GB of available memory when the number of partitions is 4. Even with 4 NVIDIA A100 GPUs (each with 80GB), there remains a discrepancy between

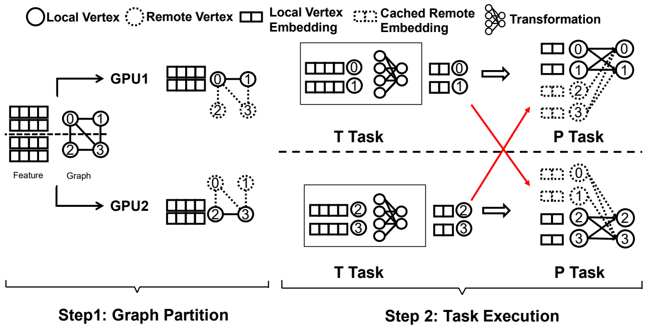


Figure 3: The data parallelism training flow for a single-layer GNN and the generation of neighbor replications.

the available and required memory. Some recent works [27, 64] propose offloading data storage to the CPU to alleviate GPU memory consumption. However, these frameworks introduce frequent data switches between the CPU and GPU. Given the low-bandwidth PCIe connection between the CPU and GPU, such frequent data transfers can significantly impact performance [63].

In summary, the performance and scalability of existing frameworks are still limited by the substantial neighbor replication across multiple GPUs. Avoiding the partitioning of graph data across different GPUs is crucial to addressing these issues. In this paper, we explore task parallelism approaches for multi-GPU GNN training by fully partitioning GNN tasks before partitioning the graph structure. This strategy minimizes graph partitions and reduces communication and memory overhead from neighbor replication.

3 THE NEUTRONTASK

We propose NeutronTask, a multi-GPU system for large-scale GNN training, which reduces memory consumption caused by neighbor replication and intermediate data through two key techniques. Figure 4 provides an architecture overview of NeutronTask.

GNN Task Parallelism. NeutronTask designs GNN task parallelism, which reduces the substantial memory consumption caused by neighbor replication. Firstly, we partition the T and P tasks of each model layer across different GPUs, limiting each GPU's memory usage to the intermediate data of the allocated tasks, rather than managing all tasks and neighbor replications as in data parallelism. Secondly, we employ the intra-GPU graph partition to address the issue where the memory requirements of a single training task exceed the GPU memory. Then, we design a vertex dependency management approach with cross-subgraph neighbor reusing and random subgraph grouping, caching neighbors on the same GPU to prevent replication. Additionally, the subgraph computation and communication can be overlapped to improve performance. Thirdly, we propose a hybrid parallelism approach to efficiently utilize scenarios where the number of GPUs exceeds the number of tasks. By adopting these strategies, GNN task parallelism significantly enhances the scalability and efficiency of multi-GPU GNN training.

Task-decoupled GNN Training Framework. Full-graph GNN training caches intermediate data, which can only be released during backward propagation. The linear P task does not contain parameters, and caching its intermediate data is caused by the alternating execution of T and P tasks. To address this, we propose a task-decoupled GNN training framework, which extends the decoupled training to task parallelism. Specifically, we separate T and P tasks within the GNN model, executing all T tasks first, followed by all P tasks. This framework reduces intermediate data from P tasks and facilitates using traditional DNN recomputation techniques [12] to reduce intermediate data from T tasks. Additionally, we provide a flexible task scheduling model, which adjusts the number of GPUs executing T and P tasks and meets different resource requirements to enhance training performance.

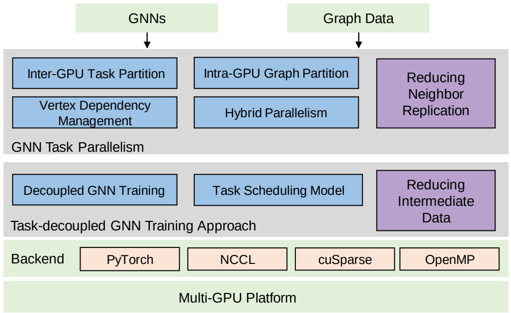


Figure 4: NeutronTask overview.

4 GNN TASK PARALLELISM

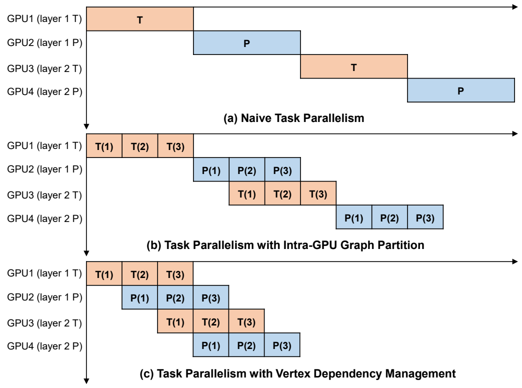
In this section, we provide a detailed design of GNN task parallelism, including task partition and intra-GPU graph partition.

4.1 Task Partition

Based on the following observations, we propose partitioning tasks across GPUs rather than partitioning the graphs. Firstly, T and P tasks have distinct computational characteristics and resource requirements. T tasks involve contiguous storage and matrix multiplication of vertex embeddings and model parameters, making them compute-intensive. P tasks involve frequent random memory access, making them memory-intensive. Therefore, partitioning tasks on the same GPU leads to significant memory requirements and prevents full utilization of GPUs. In contrast, partitioning tasks across GPUs allows for tailored optimizations based on the specific computational and memory characteristics of T and P tasks, ultimately improving performance. Secondly, the intermediate data generated by T and P tasks serve different roles. Storing them separately has no impact on model training. The intermediate data of T tasks is to compute neural network parameter gradients, which is the main objective of GNN training. The intermediate data of P tasks is used to link the computation graphs between two T tasks.

Initial Partitioning Setting. Based on the above analysis, we partition the T and P tasks of each model layer across different GPUs. This requires coordinating the allocation by considering the relationship between the number of GPUs and tasks. Based on the determined number of GPUs (N_{GPU}) and GNN tasks (N_{task}), we provide three initial partitioning strategies:

- **When** $N_{GPU} < N_{task}$**:** We allocate the same type of tasks to the same GPU, so each GPU handles only one computational characteristic (either compute-intensive or memory-intensive), maximizing resource utilization.



(a) Naive Task Parallelism

(b) Task Parallelism with Intra-GPU Graph Partition

(c) Task Parallelism with Vertex Dependency Management

Figure 5: GNN task parallelism with intra-GPU graph partition on the forward propagation of a 2-layer GNN model. (i) represents the i-th subgraph.

- **When $N_{GPU} = N_{task}$:** Each GPU executes a single task, reducing memory requirement by avoiding neighbor replication.
- **When $N_{GPU} > N_{task}$:** We use a hybrid parallelism approach, grouping GPUs to use data parallelism within groups and task parallelism between groups. The neighbor replication occurs only within P task groups.

4.2 Intra-GPU Graph Partition

GNN task parallelism execution flow is illustrated in Figure 5(a), where four GPUs are responsible for T tasks and P tasks in each model layer, respectively. Initially, the entire graph data is processed by GPU1 for the T tasks, and the results are then transferred to GPU2 for the P tasks. We propose an intra-GPU graph partition to address situations where the intermediate data generated by the allocated tasks exceeds a single GPU memory capacity and to overlap the computation of GPUs. All GPUs use the same partitioning strategy to partition the entire graph into multiple subgraphs that fit within a single GPU memory and then process them sequentially.

With intra-GPU graph partition, the training flow of GNN task parallelism is illustrated in Figure 5(b). The input graph is partitioned into three subgraphs, each storing a disjoint set of vertices along with their vertex dependencies. Vertex dependencies necessitate that the P tasks of each subgraph involve vertex embeddings from other subgraphs in their computations. For instance, GPU2 cannot execute the P task for subgraph1 until the T task of all subgraphs in GPU1 has been completed.

Vertex Dependency Management Approach. To reduce the waiting time of GPUs executing P tasks and enable the task-level pipeline parallelism illustrated in Figure 5(c), we design a vertex dependency management approach. Firstly, cross-subgraph neighbors located within the same GPU can be reused. As illustrated in Figure 5(b), after GPU2 executes the P tasks for subgraph1, it caches the neighbors that belong to subgraph2 and subgraph3. When subgraph2 and subgraph3 are executed, these neighbors can be reused. Since these neighbors are cached only once within GPU2, it avoids neighbor replication. Secondly, to overlap the computation between subgraphs as shown in Figure 5(c), we apply an approach similar to ClusterGCN [13], which skips the aggregation of uncomputed neighbor embeddings and reconfigures the edges within each subgraph at the beginning of every epoch to ensure that all neighbors are computed. As shown in Figure 6, we divide the entire graph into multiple chunks, where $C[i][j]$ represents the graph structure with destination vertices in chunk i and source vertices in chunk j. Before the start of each epoch, multiple chunks are randomly regrouped into a subgraph, allowing the edges between these chunks to be trained in the current epoch. From a probabilistic perspective, as the number of training epochs increases, different chunks have a chance to combine into one subgraph, meaning all edges have a probability of being computed.

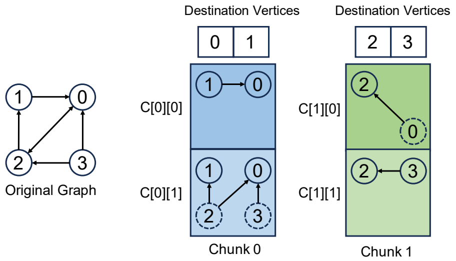


Figure 6: Chunk-based graph structure.

Algorithm 1 outlines the flow for executing the P task using vertex dependency management. To begin with, we initialize the aggregated vertex embeddings to zero (line 1). Then, we shuffle the chunks and group every k chunks into a subgraph (*i.e.,* $k = \frac{M}{S}$) (line 3). In the task executing stage, we traverse each subgraph s, with chunk $i \in [s*k, (s+1)*k]$ belonging to the current subgraph (lines 5-21). For the chunks before subgraph s, we reuse the cached neighbor embeddings for aggregation (lines 10-13), while the chunks after s will be dropped. It is worth mentioning that we utilized sparse matrix multiplication with cuSparse to execute P tasks, which is suitable for leveraging the CPU capability. After completing the aggregation computation within each chunk of a subgraph, we cache the subset of vertices that have dependencies on chunks following the subgraph s (line 19). Finally, we return the aggregated vertex embeddings of the P task (line 22).

The Effectiveness of Task Parallelism. For data parallelism, the boundary vertices of each subgraph (B_i) may be replicated on other GPUs and each GPU caches the embeddings of the remote neighbors and local vertices. Assuming the average replication count is β, where $1 < \beta \leq (M-1)$, the memory requirement for P tasks can be formalized as $MR_{dp} = \sum_i^M (\beta \cdot |B_i| + |V_i|) \cdot L \cdot |d|$, where M represents the number of GPUs, L represents the model layers, and $|d|$ represents the average dimension of hidden layers. For task parallelism, each subgraph is loaded onto each GPU sequentially, and the memory requirement for P tasks is $MR_{tp} = \sum_{i=1}^M (|B_i| + |V_i|) \cdot L \cdot |d|$. Since $\beta > 1$, we have $MR_{tp} < MR_{dp}$. Therefore, task parallelism has lower memory requirements than data parallelism.

4.3 Hybrid Parallelism

We propose hybrid parallelism to address the scenario where the number of GPUs exceeds the number of GNN tasks. Hybrid parallelism combines data parallelism and task parallelism through a grouping mechanism, with data parallelism within groups and task parallelism between groups. This approach maximizes resource

Algorithm 1 The P tasks using vertex dependency management.

```
Input: Graph chunks $C[i][j]$, The number of chunks $M$, The number of subgraphs $S$, Vertex embeddings of each chunk $i$ $X_i$
Output: Aggregated vertex embeddings: $H_0, H_1, \cdots, H_M$
Init $H_0, H_1, \cdots, H_M$;
// Every $k$ chunks will be grouped into a subgraph ($k = \frac{M}{S}$)
$k, C' = Chunk\_Shuffle(C, M, S)$;
// Traverse each subgraph
for $s \leftarrow 0, \ldots, S-1$ do
    // Traverse all chunks within this subgraph
    for $i \leftarrow s * k, \ldots, (s+1) * k$ do
        // Traverse all chunks prior to this subgraph
        for $j \leftarrow 0, \ldots, (s+1) * k$ do
            if $j < s * k$ then
                //Reuse the embeddings of computed subgraphs
                $X = \text{load}(\{X_j | \forall j \in C'[i][j]\})$;
                $h = AGG\_with\_reuse(X)$;
            else
                $h = AGG(\{X_j\} | \forall j \in C'[i][j]\})$;
            end if
            $H_i.add(h)$;
        end for
        Caching $X_i$;
    end for
end for
return $H_0, H_1, \cdots, H_M$;
```

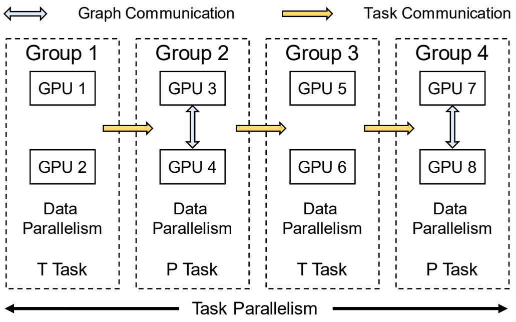


Figure 7: An illustration of hybrid parallelism.

utilization on multi-GPU platforms while minimizing neighbor replication compared to solely data parallelism.

The grouping mechanism is illustrated in Figure 7. All GPUs are divided into groups according to the number of tasks, with each group handling one task. Within each group, data parallelism is used. Taking the ogbn-papers100M from Table 2 as an example, the issue in Figure 7 causes only 1.25× neighbor replication. Compared to data parallelism with 8 GPUs (2.13×), hybrid parallelism reduces neighbor replications by 1.7×. This is because only the group handling the P task generates neighbor replications. Consequently, hybrid parallelism reduces neighbor replication by minimizing graph partitions, making it advantageous for training large-scale graphs.

Hybrid parallelism executes task parallelism between groups. The chunk-based graph structure shown in Figure 6 can be directly applied to hybrid parallelism. Specifically, we set the chunk count as a multiple of GPUs in a group, ensuring each GPU processes an equal share for workload balance. It is worth mentioning that if each group consists of a single GPU, hybrid parallelism degrades into GNN task parallelism. In summary, hybrid parallelism is utilized when there are a large number of GPUs. The core idea is to use idle GPUs to help heavily workload GPUs complete training tasks.

5 TASK-DECOUPLED GNN TRAINING FRAMEWORK

In this section, we provide a task-decoupled approach and recomputation technique to further reduce memory usage of full-graph GNN training. Then, we give the overall workflow of NeutronTask.

5.1 Task-decoupled GNN Training

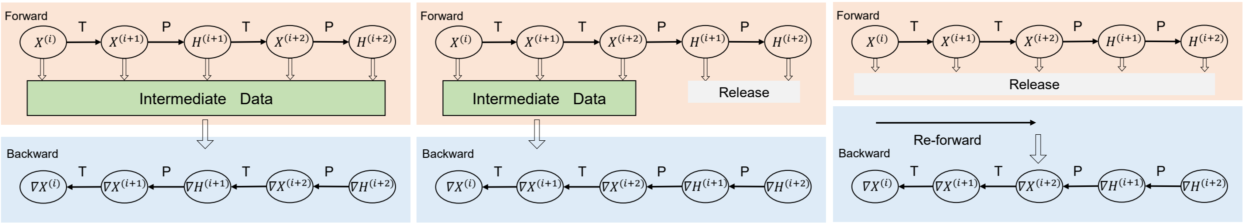
Decoupled GNN Techniques. Related studies [25, 66] show that the character with T tasks extracting feature information and P tasks learning structural information drives the efficiency of GNNs, rather than alternating execution mode. Therefore, some decoupled GNN models [11, 24, 30, 35, 48, 50, 66, 81, 83] advocate separating the execution of T and P tasks, achieving high model accuracy and scalability. A recent work (NeutronTP) [2] has extended decoupled GNN training to general GNN training, reducing the frequency of distributed communication caused by tensor parallelism while providing convergence proof to ensure model accuracy. Specifically, the general decoupled GNN training is as follows:

$$Y = g_\theta(X^{(0)}) \cdot e_\theta\big(g_\theta(X^{(0)}), A\big) \cdot f(\hat{A}), \tag{4}$$

where $g_\theta()$ represents vertex-based T tasks, transforming the input features into vertex embeddings. $e_\theta()$ represents the Scatter P task and edge-based T task, which send vertex embeddings to edges to compute edge weights. $f()$ represents P tasks, which perform multi-layer graph propagation using edge weight.

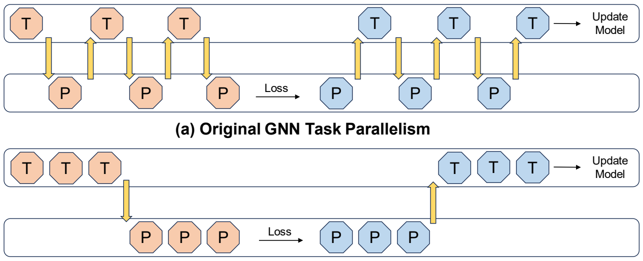
Task-decoupled Approach. In this paper, we further analyze the role of decoupled training in reducing memory usage during GNN training. We integrate task parallelism with the decoupled training and propose a task-decoupled approach that immediately releases the intermediate data generated by P tasks. As shown in Figure 8(b), the P tasks are extracted from the GNN model and executed consecutively after all T tasks. During backward propagation, consecutive P tasks propagate and aggregate vertex gradients, and their intermediate data does not need to be stored (*i.e.,* $H^{(i+1)}$ and $H^{(i+2)}$). Compared to the original GNN model that caches intermediate data from all P tasks (Figure 8(a)), the task-decoupled approach only requires reserving space for one P task, reducing the memory requirements. For complex GAT models, since the Scatter P task is executed before the edge T task, we must cache the edge embeddings generated by the Scatter P task, which consumes a significant memory resource. To address this challenge, we swap the execution order of the Scatter P task and the edge T task following the solution proposed by GATv2 [5], ensuring that all P tasks are executed consecutively so that release their intermediate data.

The flow of the task-decoupled approach is as follows. Given the L-layer GNN model, we first separate all T tasks from P tasks and execute the task partition based on the L-layer T and P tasks. Then, we load the subgraphs to execute the L-layer vertex-based T tasks and transfer the obtained vertex embeddings to the GPUs of P tasks. Subsequently, we execute L-layer P tasks, obtaining



(a) Original GNN Training (b) Task-decoupled GNN Training (c) Task-Decoupled GNN Training with Recomputation

Figure 8: The data flow of different training methods. Ellipses represent tensors generated by each operation. The data pointing to the green area represents intermediate data that needs to be cached in GPU memory.



(a) Original GNN Task Parallelism

(b) Task-decoupled GNN Task Parallelism

Figure 9: An illustrative example for comparing the communication frequency of original GNN task parallelism and task-decoupled GNN task parallelism (3-layer GNN).

aggregated vertex embeddings and calculating the loss. Backward propagation involves executing the previous process in reverse, using the obtained vertex gradients instead of vertex embeddings.

Furthermore, we analyze the communication complexity of data parallelism, task parallelism, and task-decoupled approach. Let L be the number of layers, M be the number of GPUs, $|V|$ be the number of vertices, and $|d|$ be the average embedding dimension. For data parallelism, which only communicates neighbor vertices, the communication complexity is $O((\alpha-1)|V||L||d|)$, where $\alpha \geq 1$ represents the replication factor. For task parallelism, which transfers all vertex embeddings between the T and P tasks, the communication complexity is $O((2L-1)|V||d|)$. For task-decoupled approach, the communication complexity is $O((M-1))|V||d|)$. Since α is proportional to the average degree, datasets with a higher average degree exhibit greater communication volume in data parallelism. When $N_{GPU} < N_{task}$, the task-decoupled approach can effectively address the frequent communication issues inherent in the alternating execution mode, as shown in Figure 9.

Recomputation Technique. Based on task-decoupled GNN training, T tasks are grouped together for execution, and this part can be seen as a traditional DNN operation. Therefore, we can use recomputation techniques [12] to reduce the intermediate data generated by T tasks. As is illustrated in Figure 8(c), the recomputation technique involves re-executing the T tasks during backward propagation instead of caching their intermediate data, reducing the runtime memory consumption in decoupled GNN training.

NeutronTask implements the recomputation technique into task-decoupled GNN task parallelism to reduce memory consumption. In forward propagation, after each GPU completes the T task for a subgraph, it immediately releases all intermediate data to reserve memory space for the next subgraph. In backward propagation, each subgraph performs recomputation and gradient propagation in parallel (recomputing the T task starting from the first GPU and propagating gradients starting from the last GPU). Note that the recomputation technique maintains the accuracy of the original approach because the recomputed intermediate data are identical to that generated during forward propagation.

The Effectiveness of Task-Decoupled Approach. For data parallelism, the memory requirement for intermediate data is $ID_{dp} = 2 \cdot |V||L|d|$, where $|V|$ is the number of vertices, L is the model layers, and $|d|$ is the average dimension of hidden layers. For task-decoupled approach, intermediate data is released after performing tasks and the memory requirement is $ID_{tp} = |d| \sum_{i=m}^{n} |V_i|$, where m and n represent the subgraphs being processed. Since $\sum_{i=m}^{n} |V_i| < |V|$, we have $ID_{dp} < ID_{tp}$. Therefore, task-decoupled approach reduces memory requirement for intermediate data.

5.2 Task Scheduling Model

NeutronTask integrates task parallelism and task-decoupled GNN training framework, minimizing runtime memory consumption during GNN training. In this section, we provide a task scheduling model that optimizes GPU resource utilization by adjusting the number of GPUs allocated to each task based on the workload. Initially, we analyze the factors that affect the execution time of T and P tasks. Firstly, T tasks run slower on graphs with larger input feature dimensions, while P tasks run slower on graphs with higher average degrees. Secondly, T tasks are compute-intensive, with performance primarily related to computational resources, whereas P tasks are memory-intensive, primarily associated with memory access speed [53].

Based on the above analysis, we design a task scheduling model. Firstly, we estimate the computation time of T and P tasks in a given GNN workload using the following formulas:

$$T_t = \frac{\sum_{l=1}^{L} 2 \cdot |V| \cdot d_v^{(l)} \cdot d_v^{(l+1)} + 2 \cdot |E| \cdot d_e \cdot d_e}{\text{FLOPS}} \tag{5}$$

$$T_p = \sum_{l=1}^{L} \Big(\gamma \cdot \frac{2 \cdot |V| \cdot \bar{d} \cdot d_v^{(l)}}{\text{FLOPS}} + (1-\gamma) \cdot \frac{|V| \cdot \bar{d} \cdot (1 + 2 \cdot d_v^{(l)}) + |V|}{\text{Bandwidth}} \Big) \tag{6}$$

where L is the number of GNN layers, *e.g.,* L vertex-based T task, L P tasks, and an edge-based T task in complex models. $|V|$ and

Algorithm 2 Workflow of NeutronTask for a single epoch.

```
Input: $G = (V, E)$, input feature $X^{(0)}$, Epoch $e$, the number of GPU $N$,
    the number of subgraph $S$
Output: Updated Parameter of GNN model $W$
1:  $\{T_k|, 0 < k < q\}, \{P_k|, q < k < N\} = Task\_Partition(N, G)$;
2:                       ▷ $P_k$ represents the number of P tasks in each GPU
3:  $\{SG[i][j] | 0 \le i, j < S\} = Graph\_Partition(G, S)$;
4:  SyncALLGPU();
5:  for each $s \in SG[i][j]$ do in pipeline
6:      Transfer feature to the first "T" GPU, i.e., $H_s^{(0)}$;
7:      for GPU $k = 0$ to $q - 1$ do
8:          $H_s^{(k)} = g_\theta(X_s, W_k, T_k)$ (or $\gamma = e_\theta(X_s, W_e)$);
9:          Transfer $H_s^{(k)}$ to next "T" GPU;
10:     end for
11:     Transfer $H$ from "T" GPU to first "P" GPU;
12:     for GPU $k = q$ to $N - 1$ do
13:         $H_s^{(k)} = f(\mathrm{H}, \hat{A}, P_k)$; // vertex dependency management
14:         Transfer $H_s^{(k)}$ to next "P" GPU;
15:     end for
16:     loss = $downstream\_task(H_s^{(k)})$;
17:     $\nabla H = loss.backward()$;
18:     $Re\_forward()$; // from GPU 0 to $q - 1$
19:     $P\_backward()$; // from GPU $N - 1$ to $q$
20:     $T\_backward()$ and $Update(W)$;
21: end for
```

$|E|$ are the number of vertices and edges in the graph, respectively. $d_v^{(l)}$ is the embedding dimensions of the vertices and edges at layer l, and d_e represents the last layer of T tasks ($d_e = d_v^{(L)}$). $\bar{d}$ is the average degree of vertices in the graph. FLOPS denotes the computational capacity of the GPU in terms of floating-point operations per second. Bandwidth is the memory bandwidth. γ represents the extent to which computation and communication overlap, typically determined empirically based on hardware characteristics.

Based on these measurements, we calculate the number of GPUs allocated to P tasks (N_p) and T tasks (N_t) as follows:

$$N_p = N_g \cdot \frac{T_p}{T_p + T_t}, \tag{7}$$

$$N_t = N_g - N_p, \tag{8}$$

where N_g is the total number of available GPUs. By task scheduling model, NeutronTask ensures optimal performance and efficient utilization of available hardware. Additionally, the task scheduling model is a pre-processing phase. The complexity of task scheduling is constant and its overhead is relatively small, accounting for about 0.01% of the total time for running 100 epochs.

5.3 Overall Execution Flow in NeutronTask

Algorithm 2 outlines a single epoch's execution in NeutronTask. To begin with, GNN tasks are allocated by the task scheduling model. The $\{T_k | 0 < k < q\}$ and $\{P_k | q - 1 < k < N\}$ represent the number of T and P tasks in each GPU, N represents the number of GPUs, and q represents the boundary point for handling two types of GPU, where GPUs with $GPU_{id} < q$ handle T tasks, and the remaining GPUs handle P tasks (line 1). Then, we employ intra-GPU graph partitioning to partition the input graph into multiple subgraphs (SG), where the organization of each SG resembles the chunk-based graph structure shown in Figure 6 (line 3).

Table 3: Dataset description. |V|, |E|, #F, #L, and #hidden represent the number of vertices, edges, feature dimensions, and labels, respectively. |TR| represents the ratio of train vertices.

Dataset	|V|	|E|	#F	#L	|TR|
cora [45]	2.70K	5.43K	1433	7	59.3%
reddit [23]	232.96K	114.62M	602	41	90.6%
ogbn-products [26]	2.45M	61.86M	100	47	8.03%
it-2004 [4]	41M	1.2B	256	64	25%
ogbn-papers100M [26]	111.06M	1.62B	128	172	1.1%
friendster [31]	65.6M	2.5B	256	64	25%

Before the start of each epoch, all GPUs are synchronized (line 4). During the training process, SGs are trained sequentially within a GPU and scheduled with pipeline parallelism across GPUs. The feature $H_s^{(0)}$ is transferred from the host to the GPU executing the first T task (line 6). Then, the embeddings of each SG are transmitted across GPUs in the predetermined order, performing the corresponding tasks (lines 7-15). In backward propagation, recomputation (line 18) and gradient propagation (line 19) can be executed in parallel, ultimately updating the model parameters (line 20).

6 EXPERIMENTAL EVALUATION

6.1 Experimental Setup

Environments. The multi-GPU experiments are conducted on a GPU server equipped with 2 Intel(R) Xeon(R) Silver 4316 CPUs, 377GB DRAM, and 4 NVIDIA A5000 (24GB) GPUs. Each CPU is connected to two GPUs via the PCIe link, and the multi-GPU devices are connected with PCIe 4.0x 16. The server runs Ubuntu 20.04 OS with GCC-9.4.0, CUDA 11.3, PyTorch 1.13.0, and NCCL backend.

Datasets and GNN Algorithms. Table 3 presents the parameters of real-world graphs used in our experiments. For graphs without properties (it-2004 and friendster), we use randomly generated features, labels, training (25%), test (25%), and validation (50%) set division. We use two popular GNN models (GCN [29] and GAT [56]). The hidden layer dimensions for reddit and ogbn-products are 256, while for cora, it-2004, ogbn-papers100M, and friendster, they are 128. The partitions are 4, while for the large graphs (it-2004, ogbn-papers100M, and friendster), the partitions are 32.

System for Comparison. We compare NeutronTask with three popular GNN systems: DGL [62], NeutronStar [64], and Sancus [43], all of which employ data parallelism. DGL uses mini-batch training to save memory by splitting training vertices into batches, each sampling a subset of neighbors for GPU training. In our evaluation, the batch size is set to 1024 and the fan-out is set to 10. NeutronStar divides the entire graph into subgraphs and loads them sequentially into the GPU. Sancus reduces communication by using historical embeddings and caches these embeddings locally.

6.2 Overall Comparison

We compare NeutronTask with DGL [62], Sancus [43], and NeutronStar [64] on a node with 4 GPUs to show the processing scale with limited GPU resources. The results are reported in Table 4.

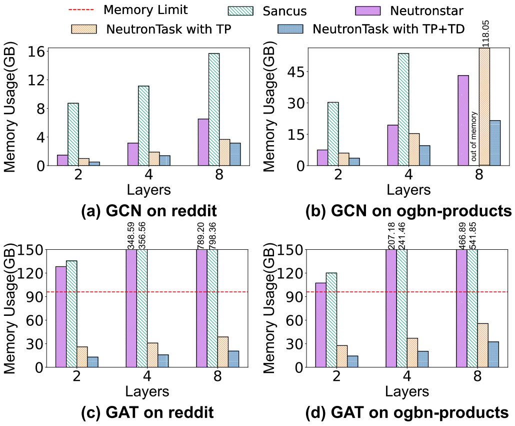
Table 4: Comparison of the per-epoch time (unit: s) with DGL, Sancus, and NeutronStar. (OOM represents out-of-memory)

Layers	Dataset	GCN				GAT			
		DGL	Sancus	NeutronStar	NeutronTask	DGL	Sancus	NeutronStar	NeutronTask
2	cora	0.025(3.13×)	0.017(2.13×)	0.013(1.66×)	0.008	0.021(2.1×)	0.023(2.3×)	0.028(2.8×)	0.010
	reddit	0.34(3.4×)	0.18(1.8×)	0.52(5.2×)	0.10	1.04(4.0×)	OOM	OOM	0.26
	ogbn-products	1.06(3.03×)	0.66(1.89×)	0.94(2.69×)	0.35	0.88(1.31×)	OOM	OOM	0.67
	it-2004	48.77(3.31×)	OOM	OOM	14.72	96.83(3.78×)	OOM	OOM	25.6
	ogbn-papers100M	2.76(0.13×)	OOM	OOM	25.88	10.71(0.2×)	OOM	OOM	54.84
	friendster	65.6(3.46×)	OOM	OOM	18.98	108.73(2.51×)	OOM	OOM	43.33
4	cora	0.044(2.32×)	0.03(1.58×)	0.05(2.63×)	0.019	0.0358(1.7×)	0.036(1.71×)	0.078(3.71×)	0.021
	reddit	0.625(3.29×)	0.36(1.89×)	0.92(4.84×)	0.19	1.96(5.45×)	OOM	OOM	0.36
	ogbn-products	1.378(2.46×)	1.82(3.25×)	1.75(3.13×)	0.56	3.86(4.71×)	OOM	OOM	0.82
	it-2004	64.4(3.45×)	OOM	OOM	18.69	OOM	OOM	OOM	47.43
	ogbn-papers100M	17.86(0.36×)	OOM	OOM	57.72	61.65(0.6×)	OOM	OOM	102.39
	friendster	OOM	OOM	OOM	35.31	OOM	OOM	OOM	75.95
8	cora	0.063(1.91×)	0.042(1.27×)	0.05(1.52×)	0.033	0.058(1.81×)	0.057(1.78×)	0.099(3.09×)	0.032
	reddit	1.546(4.83×)	0.71(2.22×)	1.75(5.47×)	0.32	OOM	OOM	OOM	0.41
	ogbn-products	OOM	OOM	6.23(5.32×)	1.17	OOM	OOM	OOM	1.58
	it-2004	96.45(2.58×)	OOM	OOM	37.45	OOM	OOM	OOM	77.85
	ogbn-papers100M	OOM	OOM	OOM	84.54	OOM	OOM	OOM	212.94
	friendster	OOM	OOM	OOM	77.71	OOM	OOM	OOM	152.96

Compared to DGL, NeutronTask achieves an average speedup of 2.87×. The sampling method of DGL faces the neighbor explosion problem [27], causing computation and memory costs to grow exponentially with model depth. Since DGL needs to store intermediate data, it encounters OOM errors when handling deep GNN models. In contrast, by utilizing the task-decoupled approach and recomputation techniques, NeutronTask can immediately release intermediate data after completing each task. Additionally, DGL's sampling process is time-consuming, leading to lower training efficiency than NeutronTask. For ogbn-papers100M, DGL performs better because it trains only on the 1.1% training set, while NeutronTask performs full-graph training, incurring higher computational overhead.

Compared to Sancus and NeutronStar, NeutronTask supports large-scale graph training and outperforms them. Sancus increases memory usage by caching historical embeddings locally. NeutronStar partitions subgraphs but still stores intermediate data in GPUs. NeutronTask reduces the memory consumption by task parallelism and task-decoupled approach, enhances system performance by efficient pipeline parallelism. Firstly, NeutronTask reduces neighbor replication by avoiding inter-GPU graph partition and reduces intermediate data by decoupling T tasks from P tasks. As a result, NeutronTask can train on all datasets. Secondly, each GPU transmits its computation results to the GPU handling the next task, hiding communication within pipeline computations. On successfully runs, NeutronTask achieves a speedup from 1.27× to 5.47×.

Our observations indicate that the performance advantage of NeutronTask over other systems increases as the model layer grows. For the 2-layer model, NeutronTask achieves an average 2.37× speedup over systems. For the 4-layer and 8-layer models, the speedups are 2.65× and 3.14×, respectively. As the number of layers increases, data parallelism must manage more remote neighbors (i.e., inter-GPU communication), while task parallelism reduces the communication overhead from these neighbors.



(a) GCN on reddit

(b) GCN on ogbn-products

(c) GAT on reddit

(d) GAT on ogbn-products

Figure 10: Memory reduction analysis, where each bar represents the peak memory consumption. "TP" indicates task parallelism, and "TD" indicates task-decoupled training.

6.3 Memory Reduction Analysis

We analyze the memory reduction achieved by task parallelism (TP) and task-decoupled training (TD), comparing with NeutronStar [64] and Sancus [43]. Figure 10 shows the results across 4 GPUs.

NeutronStar uses 19.9% – 43.7% of its memory to cache neighbor replication, while Sancus additionally caches historical embeddings by 32.7% – 43.3%. In contrast, NeutronTask reduces neighbor replication through TP. For reddit, TP reduces more neighbor replications than ogbn-products due to its higher average degree, which causes significant vertex dependencies across subgraphs. To cache the intermediate data, NeutronStar and Sancus use 17.6% – 28.9% of the memory, while NeutronTask leverages TD to release this memory early. Additionally, NeutronTask employs recomputation techniques to further reduce intermediate data of T tasks. For the

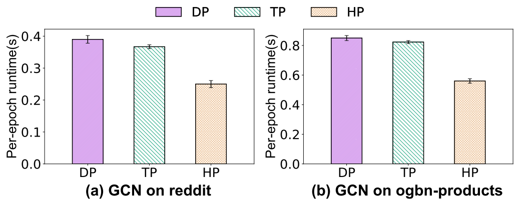


Figure 11: The performance comparison with error bars. The "DP" indicates data parallelism, "TP" indicates task parallelism, and "HP" indicates hybrid parallelism.

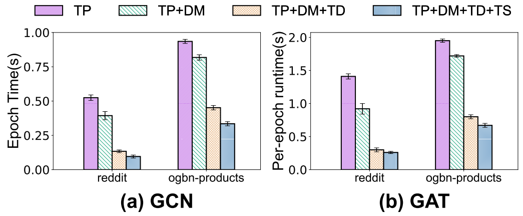


Figure 12: Performance gain analysis with error bars. "TP" indicates the naive task parallelism, "DM" indicates the vertex dependency management, "TD" indicates the task-decoupled GNN training, and "TS" indices the task scheduling model.

GAT model, both NeutronStar and Sancus exceed the total memory of 4 GPUs (96GB), and we compute their theoretical memory requirements. NeutronTask reduces memory usage by 81.5% – 97.4% by leveraging vertex weights for storage instead of storing a large number of edge embeddings. In summary, compared to these systems, NeutronTask reduces total memory usage by 49% – 67%.

6.4 Efficiency of the Hybrid Parallelism

NeutronTask employs hybrid parallelism (HP) to handle cases where the number of GPUs exceeds the number of tasks. We compare HP with data parallelism (DP) and task parallelism (TP) to analyze performance. The experiments are conducted on an Alibaba Cloud ECS server equipped with 128 vCPU cores, 512 GB of DRAM, and 8 A10 GPUs (each with 24 GB). The software setup is described in Section 6.1. We use a 2-layer GCN model, consisting of two P tasks and two T tasks. In this setup, DP uses NeutronStar as the baseline. TP only utilizes 4 GPUs. For HP, the GPUs are divided into 4 groups, with each group handling one task. Neither TP nor HP includes the task-decoupled approach. Figure 11 shows the results. Compared to DP, HP achieves a 1.52× - 1.56× speedup. This is because DP requires frequent communication with remote neighbors. However, HP reduces the communication overhead and overlaps communication with computation. Compared to TP, HP achieves a 1.46× - 1.47× speedup since TP uses only 4 GPUs, leaving the rest idle, while HP maximizes GPU utilization by grouping GPUs.

6.5 Performance Gain Analysis

We analyze the performance gain of NeutronTask with vertex dependency management (DM), task-decoupled GNN training (TD), and task scheduling model (TS). Using naive task parallelism (TP) as the baseline, we gradually integrate these optimizations. Figure 12 shows the results. Compared to TP, TP+DM achieves 1.6× - 1.8× speedups. As shown in Figure 5, DM reduces GPU bubble time in task parallelism by overlapping subgraph task execution across GPUs. Compared to TP+DM, TP+DM+TD achieves 1.8× - 2.9× speedups due to the inherent advantages of decoupled training. In the general GNN model, P tasks perform computations using vertex embeddings from each hidden layer. In contrast, TD first executes all T tasks, which may reduce vertex embedding dimensions. TD achieves greater performance gains for reddit than ogbn-products. This is because after T tasks, the vertex embedding dimensions in reddit dataset sharply decrease, directly reducing P task execution time. For the GAT model, TD provides greater performance gains than for the GCN model because it decouples edge weight computation for P and T tasks, further reducing inter-GPU communication frequency. Finally, compared to TP+DM+TD, TP+DM+TD+TS achieved 1.35× - 1.39× speedups by adjusting the number of GPUs executing different tasks, enhancing resource utilization.

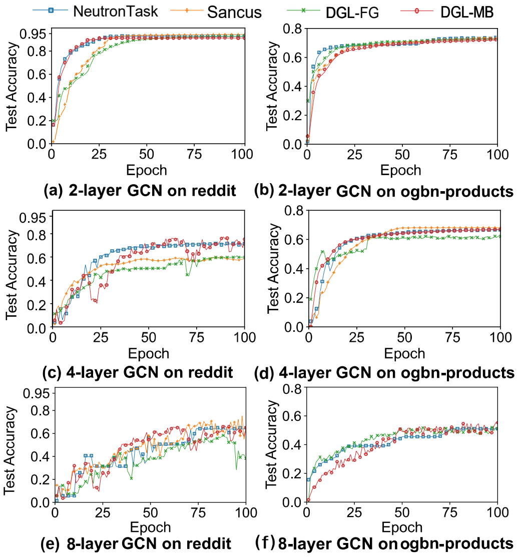
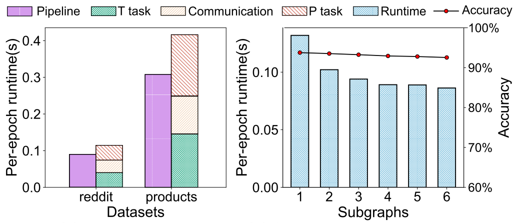


Figure 13: Epoch-to-accuracy. MB represents the mini-batch training, and FG represents the full-graph training.

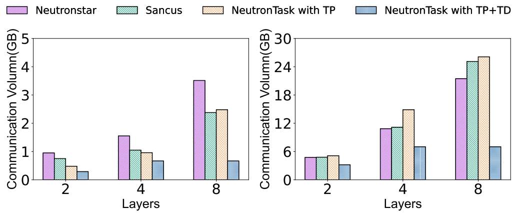
6.6 Accuracy Comparison

NeutronTask employs task-decoupled training, executing L layers of T tasks first, followed by L layers of P tasks, which impact model accuracy. We compare the model accuracy with DGL [62] and Sancus [43] by running GCN on reddit and ogbn-products. Figure 13 reports the epoch-to-accuracy results. After 100 epochs, the test accuracy reaches a stable state. NeutronTask achieves almost the same accuracy as other systems, demonstrating the effectiveness of task-decoupled GNN training in ensuring model accuracy. Regarding the convergence speed, NeutronTask achieves the highest accuracy in fewer epochs. Additionally, with a significantly faster per-epoch time, NeutronTask demonstrates better time-to-accuracy performance. Sancus has the slowest convergence speed due to the use of history embedding. For deeper layers, all methods suffer from decreased accuracy, as the increased number of P and T tasks leads to over-smoothing [7, 10, 16, 17, 40, 41, 69, 78] and over-fitting [32, 44, 70, 82]. Graphs with higher average degrees are more susceptible to over-smoothing. Therefore, reddit suffers from a more significant accuracy drop compared to ogbn-products.



(a) Pipeline Ablation (b) Sensitivity Analysis on Reddit

Figure 14: Pipeline performance and sensitivity analysis.



(a) GCN on reddit (b) GCN on ogbn-products

Figure 15: The communication volume. "TP" indicates task parallelism, and "TD" indicates task-decoupled approach.

6.7 Efficiency of Pipelining

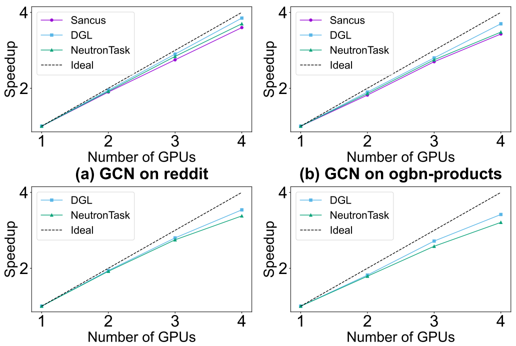
We conduct pipeline ablation experiments to demonstrate the effectiveness of the pipeline and analyze the impact of the number of subgraphs on system performance and model accuracy.

Ablation Study of the Pipeline. To evaluate the efficiency of pipelining shown in Figure 5(c), we compare it with the serial execution of 4 subgraphs. Figure 14(a) shows the result, the time elapsed on both T and P tasks is roughly the same, which exceeds the time elapsed on communication. By overlapping the communication and computation through pipelining, the total runtime can be significantly reduced (ranging from 1.2× to 1.5×).

Sensitivity Study of the Pipeline. In intra-GPU graph partition, the number of subgraphs is a configurable parameter that controls the memory consumption of training data and pipeline parallelism. To evaluate the impact of this parameter on training performance and accuracy, we run a 2-layer GCN on reddit, increasing the number of subgraphs from 1 to 6. As shown in Figure 14(b), as the number of subgraphs increases, the performance gradually improves, while accuracy decreases by less than 1%.

6.8 Communication Analysis

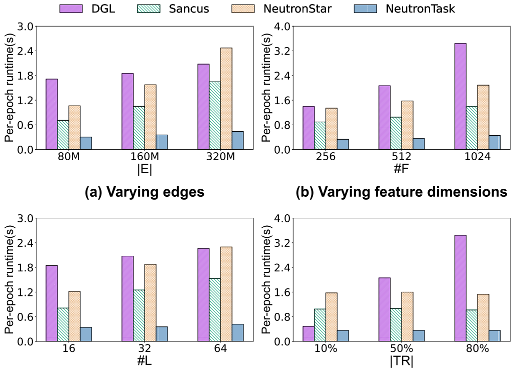
We evaluate the communication overhead of NeutronTask by integrating task parallelism (TP) and task-decoupled (TD). Figure 15 presents the experimental results compared to two full-graph data parallelism systems (NeutronStar and Sancus). For reddit, NeutronTask with TP reduces the communication overhead by an average of 36%, while for ogbn-products, NeutronTask with TP increases the communication overhead by an average of 17%. This is because when data parallelism runs on datasets with a higher average degree (reddit), there are more cross-GPU edges, resulting in a higher communication volume. Compared to other methods, NeutronTask with TP+TD reduces communication column by an average of 54.2%. This is because TD involves fewer communication frequencies.



(a) GCN on reddit (b) GCN on ogbn-products

(c) GAT on reddit (d) GAT on ogbn-products

Figure 16: Scalability analysis when varying GPU number from 1 to 4 by running 2-layer GNN models.



(a) Varying edges (b) Varying feature dimensions

(c) Varying labels (d) Varying ratio of train vertices

Figure 17: Scalability analysis when varying dataset characteristics when fixing the number of vertices at 20M.

6.9 Scalability Analysis

Scalability with Varying GPUs. We evaluate the scalability of NeutronTask by varying the number of GPUs. Figure 16 reports the results of 2-layer GCN and GAT using DGL and Sancus as baselines. We observe that the performance of NeutronTask, DGL, and Sancus improves as the number of GPUs increases. NeutronTask leverages an efficient pipeline to achieve high scalability by overlapping computation and communication. Sancus has poor scalability because the increase in the number of GPUs leads to more graph partitions (*i.e.,* more vertex dependencies and inter-GPU communication). For GAT, Sancus encounters an OOM error. DGL achieves high scalability as the total number of batches remains fixed, and increasing the number of GPUs reduces the batches processed per GPU linearly.

Scalability with Varying Dataset Characteristics. We compare NeutronTask with DGL, NeutronStar, and Sancus by varying dataset characteristics. Specifically, we utilize the RMAT [28] to generate datasets and fix the number of vertices at 20M while varying the number of edges ($\#E$, 80M, 160M, 320M), the feature dimensions ($\#F$, 256, 512, 1024), the labels ($\#L$, 16, 32, 64), and the ratio of train vertices ($|TR|$, 10%, 50%, 80%). The results are shown in Figure 17. With varying $\#E$, $\#F$, $\#L$. $|TR|$, NeutronTask achieves speedup of 5.60×, 3.68×, and 4.52× compared to DGL, Sancus, and NeutronStar, respectively. The computational overhead of mini-batch training in DGL is proportional to $|TR|$, while the computational overhead of full-graph training is independent of $|TR|$.

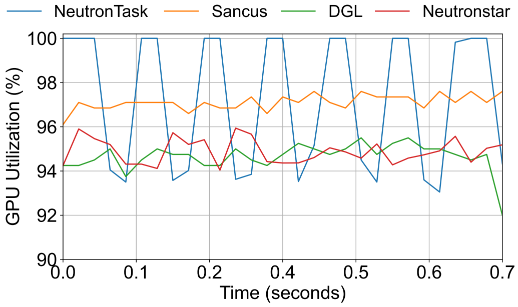


Figure 18: GPU utilization comparison. The average GPU utilizations are 97.24%, 97.11%, 94.65%, **and** 94.89% **for NeutronTask, Sancus, DGL, and NeutronStar, respectively.**

6.10 GPU Utilization

We evaluate GPU Utilization by running a 2-layer GCN on reddit. Figure 18 reports the results in a 0.7-second time window, with GPU utilization recorded every 1 millisecond. NeutronTask achieves higher GPU utilization (97.24% on average) compared to DGL (94.65%), Sancus (97.11%), and NeutronStar (94.89%), with consistently higher peak GPU utilization for most of the time. NeutronTask optimizes GPU allocation for T and P tasks by a task scheduling model but experiences some fluctuation due to pipeline bubble time. DGL shows the worst GPU utilization due to frequent random memory accesses during sampling.

6.11 Comparison with NeutronTP

We compare NeutronTask with NeutronTP, a tensor parallelism that evenly divides vertex features for load balancing. Moreover, we combine NeutronTask with tensor parallelism, grouping GPUs for tensor parallelism within each group while applying task parallelism across groups. Figure 19 reports the results with NeutronStar (data parallelism) as the baseline. Compared to NeutronStar, all methods improve performance. When training small-scale graphs, NeutronTP achieves an average speedup of 4.11× than NeutronTask. However, for large-scale graphs, the subgraph loading of NeutronTP reduces GPU utilization, leading to performance degradation and even memory exhaustion errors, while NeutronTask achieves 4.21× speedup on it-2004. By combining tensor parallelism, NeutronTask not only supports large-scale graph training but also further enhances performance. In summary, tensor parallelism and task parallelism are orthogonal techniques. Combining them can achieve better performance than using a single one.

7 LIMITATION AND FUTURE WORK

Currently, NeutronTask is designed for training large-scale graphs with limited GPU memory. When resources are sufficient or the graph scale is smaller, NeutronTask may need to be combined with other parallel methods to further enhance performance. Recently, various GNN parallel training methods, such as GNN tensor parallelism [2] and GNN model parallelism [59], have been proposed to enhance performance. However, evaluating the appropriate use cases for these methods and effectively combining them remains an open research problem. In future work, we aim to integrate existing techniques to enable automatic parallelism for GNNs, which would select suitable execution strategy (or combine multiple methods) to adapt to varying requirements of environments and data inputs.

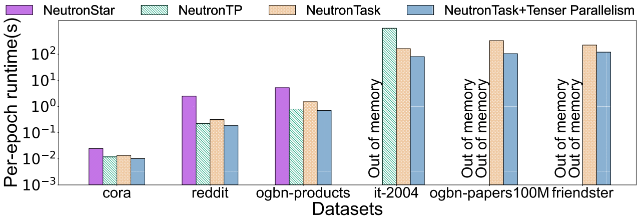


Figure 19: Comparison with various parallelization methods

8 RELATED WORK

GNN Data Parallelism. GNN data parallelism refers to partitioning graphs for parallel computation [1, 8, 19, 27, 33, 34, 38, 43, 46, 47, 51, 54, 58–60, 63–65, 71, 72, 76, 77, 79, 80, 84]. GNNLab [72], DUCATI [76], PaGraph [34], and XGNN [51] reduce CPU-GPU communication by utilizing GPU caching. P3 [19] and ByteGNN [79] improve sampling efficiency by optimizing data distribution. CAGNET [54], MGG [65], PipeGCN [59], and Sancus [43] optimize inter-GPU communication by specific parallel strategies. NeuGraph [38], ROC [27], and Hongtu [63] reduce GPU memory usage by subgraph scheduling and intermediate data management.

GNN Tensor Parallelism. Recently, NeutronTP [2] proposes tensor parallelism, ensuring that different workers process nearly identical workloads by partitioning vertex tensors instead of partitioning graphs (as vertex tensors have equal sizes and are easier to partition evenly). To address frequent communication, NeutronTP proposes a general decoupled training technique. In contrast, we employ decoupled training based on task parallelism, releasing intermediate data in advance to reduce memory requirements.

9 CONCLUSION

We present NeutronTask, a scalable and efficient system for full-graph GNN training on limited GPU memory. Firstly, NeutronTask provides task parallelism, which reduces neighbor replication and reorganizes intermediate data placement in multi-GPU GNN training by inter-GPU task partition and intra-GPU graph partition. Secondly, NeutronTask integrates a task-decoupled GNN training framework, which reduces the intermediate data in GNN training through task-decoupled GNN training and recomputation techniques. Our experiments demonstrate that NeutronTask can efficiently train on billion-scale graphs using just 4×A5000 GPU (each with 24GB of memory) by significantly reducing the memory consumption of neighbor replication and intermediate data.

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China (2023YFB4503601), the National Natural Science Foundation of China (U2241212, 62202088, 62372097, and 62461146205), and the Distinguished Youth Foundation of Liaoning Province (2024021148-JH3/501). Yanfeng Zhang and Qiange Wang are the corresponding authors.

REFERENCES

[1] Xin Ai, Qiange Wang, Chunyu Cao, Yanfeng Zhang, Chaoyi Chen, Hao Yuan, Yu Gu, and Ge Yu. 2024. NeutronOrch: Rethinking Sample-based GNN Training under CPU-GPU Heterogeneous Environments. *Proc. VLDB Endow.* 17, 8 (2024), 1995–2008.

[2] Xin Ai, Hao Yuan, Zeyu Ling, Qiange Wang, Yanfeng Zhang, Zhenbo Fu, Chaoyi Chen, Yu Gu, and Ge Yu. 2024. NeutronTP: Load-Balanced Distributed Full-Graph GNN Training with Tensor Parallelism. *Proc. VLDB Endow.* 18, 2 (2024), 173–186.

[3] Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, and Koray Kavukcuoglu. 2016. Interaction Networks for Learning about Objects, Relations and Physics. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain.* 4502–4510.

[4] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004.* ACM, 595–602.

[5] Shaked Brody, Uri Alon, and Eran Yahav. 2022. How Attentive are Graph Attention Networks?. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022.*

[6] Khac-Hoai Nam Bui, Jiho Cho, and Hongsuk Yi. 2022. Spatial-temporal graph neural network for traffic forecasting: An overview and open research issues. *Appl. Intell.* 52, 3 (2022), 2763–2774.

[7] Chen Cai and Yusu Wang. 2020. A note on over-smoothing for graph neural networks. *arXiv preprint arXiv:2006.13318* (2020).

[8] 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.* 130–144.

[9] Zhenkun Cai, Qihui Zhou, Xiao Yan, Da Zheng, Xiang Song, Chenguang Zheng, James Cheng, and George Karypis. 2023. DSP: Efficient GNN Training with Multiple GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023.* 392–404.

[10] Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. 2020. Measuring and Relieving the Over-Smoothing Problem for Graph Neural Networks from the Topological View. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020.* 3438–3445.

[11] Ming Chen, Zhewei Wei, Bolin Ding, Yaliang Li, Ye Yuan, Xiaoyong Du, and Ji-Rong Wen. 2020. Scalable Graph Neural Networks via Bidirectional Propagation. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual.*

[12] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[13] 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, KDD 2019, Anchorage, AK, USA, August 4-8, 2019.* 257–266.

[14] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain.* 3837–3845.

[15] 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.* 417–426.

[16] Guosheng Feng, Hongzhi Wang, and Chunnan Wang. 2023. Search for deep graph neural networks. *Inf. Sci.* 649 (2023), 119617.

[17] Wenzheng Feng, Jie Zhang, Yuxiao Dong, Yu Han, Huanbo Luan, Qian Xu, Qiang Yang, Evgeny Kharlamov, and Jie Tang. 2020. Graph Random Neural Networks for Semi-Supervised 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.*

[18] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. 2017. Protein Interface Prediction using Graph Convolutional Networks. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA.* 6530–6539.

[19] 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.* 551–568.

[20] Congcong Ge, Xiaoze Liu, Lu Chen, Baihua Zheng, and Yunjun Gao. 2021. LargeEA: Aligning Entities for Large-scale Knowledge Graphs. *Proc. VLDB Endow.* 15, 2 (2021), 237–245.

[21] Takuo Hamaguchi, Hidekazu Oiwa, Masashi Shimbo, and Yuji Matsumoto. 2017. Knowledge Transfer for Out-of-Knowledge-Base Entities : A Graph Neural Network Approach. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017.* 1802–1808.

[22] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation Learning on Graphs: Methods and Applications. *IEEE Data Eng. Bull.* 40, 3 (2017), 52–74.

[23] 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.

[24] Mingguo He, Zhewei Wei, and Ji-Rong Wen. 2022. Convolutional Neural Networks on Graphs with Chebyshev Approximation, Revisited. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022.*

[25] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yong-Dong Zhang, and Meng Wang. 2020. LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020.* 639–648.

[26] 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.*

[27] 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.*

[28] Weiwei Jiang and Jiayun Luo. 2022. Graph neural network for traffic forecasting: A survey. *Expert Syst. Appl.* 207 (2022), 117921.

[29] 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.*

[30] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. 2019. Predict then Propagate: Graph Neural Networks meet Personalized PageRank. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019.*

[31] Jure Leskovec and Rok Sosic. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Trans. Intell. Syst. Technol.* 8, 1 (2016), 1:1–1:20.

[32] Guohao Li, Matthias Müller, Ali K. Thabet, and Bernard Ghanem. 2019. DeepGCNs: Can GCNs Go As Deep As CNNs?. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019.* 9266–9275.

[33] Zhiyuan Li, Xun Jian, Yue Wang, Yingxia Shao, and Lei Chen. 2024. DAHA: Accelerating GNN Training with Data and Hardware Aware Execution Planning. *Proc. VLDB Endow.* 17, 6 (2024), 1364–1376.

[34] 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.* 401–415.

[35] Meng Liu, Hongyang Gao, and Shuiwang Ji. 2020. Towards Deeper Graph Neural Networks. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020.* 338–348.

[36] Xiaoze Liu, Junyang Wu, Tianyi Li, Lu Chen, and Yunjun Gao. 2023. Unsupervised Entity Alignment for Temporal Knowledge Graphs. In *Proceedings of the ACM Web Conference 2023, WWW 2023, Austin, TX, USA, 30 April 2023 - 4 May 2023.* 2528–2538.

[37] Yu-Chen Lo, Stefano E Rensi, Wen Torng, and Russ B Altman. 2018. Machine learning in chemoinformatics and drug discovery. *Drug discovery today* 23, 8 (2018), 1538–1546.

[38] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. 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.* 443–458.

[39] Diego Marcheggiani and Ivan Titov. 2017. Encoding Sentences with Graph Convolutional Networks for Semantic Role Labeling. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017.* 1506–1515.

[40] Yimeng Min, Frederik Wenkel, and Guy Wolf. 2020. Scattering GCN: Overcoming Oversmoothness in Graph Convolutional Networks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual.*

[41] Philipp Nazari, Oliver Lemke, Davide Guidobene, and Artiom Gesp. 2024. Entropy Aware Message Passing in Graph Neural Networks. *arXiv preprint*

arXiv:2403.04636 (2024).

[42] 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*. 1063–1072.

[43] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. SANCUS: Staleness-Aware Communication-Avoiding Full-Graph Decentralized Training in Large-Scale Graph Neural Networks. *Proc. VLDB Endow.* 15, 9 (2022), 1937–1950.

[44] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. 2020. DropEdge: Towards Deep Graph Convolutional Networks on Node Classification. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*.

[45] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective classification in network data. *AI magazine* 29, 3 (2008), 93–93.

[46] Yanyan Shen, Lei Chen, Jingzhi Fang, Xin Zhang, Shihong Gao, and Hongbo Yin. 2024. Efficient Training of Graph Neural Networks on Large Graphs. *Proc. VLDB Endow.* 17, 12 (2024), 4237–4240.

[47] Zeang Sheng, Wentao Zhang, Yangyu Tao, and Bin Cui. 2024. OUTRE: An OUT-of-core De-REdundancy GNN Training Framework for Massive Graphs within A Single Machine. *Proc. VLDB Endow.* 17, 11 (2024), 2960–2973.

[48] Indro Spinelli, Simone Scardapane, and Aurelio Uncini. 2021. Adaptive Propagation Graph Convolutional Network. *IEEE Trans. Neural Networks Learn. Syst.* 32, 10 (2021), 4755–4760.

[49] Jonathan M Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M Donghia, Craig R MacNair, Shawn French, Lindsey A Carfrae, Zohar Bloom-Ackermann, et al. 2020. A deep learning approach to antibiotic discovery. *Cell* 180, 4 (2020), 688–702.

[50] Dengdi Sun, Liang Liu, Bin Luo, and Zhuanlian Ding. 2023. GLASS: A Graph Laplacian Autoencoder with Subspace Clustering Regularization for Graph Clustering. *Cogn. Comput.* 15, 3 (2023), 803–821.

[51] Dahai Tang, Jiali Wang, Rong Chen, Lei Wang, Wenyuan Yu, Jingren Zhou, and Kenli Li. 2024. XGNN: Boosting Multi-GPU GNN Training via Global GNN Memory Store. *Proc. VLDB Endow.* 17, 5 (2024), 1105–1118.

[52] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. 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*. 495–514.

[53] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. 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*. 495–514.

[54] 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*. 70.

[55] 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*. 76.

[56] 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*.

[57] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. 2008. Extracting and composing robust features with denoising autoencoders. In *Machine Learning, Proceedings of the Twenty-Fifth International Conference (ICML 2008), Helsinki, Finland, June 5-9, 2008 (ACM International Conference Proceeding Series)*, Vol. 307. 1096–1103.

[58] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. 2023. MariusGNN: Resource-Efficient Out-of-Core Training of Graph Neural Networks. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*. ACM, 144–161.

[59] 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*.

[60] 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.

[61] Daixin Wang, Yuan Qi, Jianbin Lin, Peng Cui, Quanhui Jia, Zhen Wang, Yanming Fang, Quan Yu, Jun Zhou, and Shuang Yang. 2019. A Semi-Supervised Graph Attentive Network for Financial Fraud Detection. In *2019 IEEE International Conference on Data Mining, ICDM 2019, Beijing, China, November 8-11, 2019*. 598–607.

[62] 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. *arXiv preprint arXiv:1909.01315* (2019).

[63] 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.

[64] 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*. 1301–1315.

[65] 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*. 779–795.

[66] Felix Wu, Amauri H. Souza Jr., Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. 2019. Simplifying Graph Convolutional Networks. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research)*, Vol. 97. 6861–6871.

[67] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. 2023. Graph Neural Networks in Recommender Systems: A Survey. *ACM Comput. Surv.* 55, 5 (2023), 97:1–97:37.

[68] Wei Wu, Bin Li, Chuan Luo, and Wolfgang Nejdl. 2021. Hashing-Accelerated Graph Neural Networks for Link Prediction. In *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*. 2910–2920.

[69] Yujun Yan, Milad Hashemi, Kevin Swersky, Yaoqing Yang, and Danai Koutra. 2022. Two Sides of the Same Coin: Heterophily and Oversmoothing in Graph Convolutional Neural Networks. In *IEEE International Conference on Data Mining, ICDM 2022, Orlando, FL, USA, November 28 - Dec. 1, 2022*. 1287–1292.

[70] Chaoqi Yang, Ruijie Wang, Shuochao Yao, Shengzhong Liu, and Tarek F. Abdelzaher. 2020. Revisiting "Over-smoothing" in Deep GCNs. *CoRR* abs/2003.13663 (2020).

[71] Dongxu Yang, Junhong Liu, Jiaxing Qi, and Junjie Lai. 2022. WholeGraph: A Fast Graph Neural Network Training Framework with Multi-GPU Distributed Shared Memory Architecture. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022*. 54:1–54:14.

[72] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: a factored system for sample-based GNN training over GPUs. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. 417–434.

[73] Liang Yao, Chengsheng Mao, and Yuan Luo. 2019. Graph Convolutional Networks for Text Classification. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. 7370–7377.

[74] 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*. 974–983.

[75] 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.

[76] Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. 2023. DUCATI: A Dual-Cache Training System for Graph Neural Networks on Giant Graphs with the GPU. *Proc. ACM Manag. Data* 1, 2 (2023), 166:1–166:24.

[77] Yuhao Zhang and Arun Kumar. 2023. Lotan: Bridging the Gap between GNNs and Scalable Graph Analytics Engines. *Proc. VLDB Endow.* 16, 11 (2023), 2728–2741.

[78] Lingxiao Zhao and Leman Akoglu. 2020. PairNorm: Tackling Oversmoothing in GNNs. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*.

[79] 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.

[80] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *10th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms, IA3 2020, Atlanta, GA, USA, November 11, 2020*. 36–44.

[81] Yanping Zheng, Zhewei Wei, and Jiajun Liu. 2023. Decoupled Graph Neural Networks for Large Dynamic Graphs. *Proc. VLDB Endow.* 16, 9 (2023), 2239–2247.

[82] Kuangqi Zhou, Yanfei Dong, Kaixin Wang, Wee Sun Lee, Bryan Hooi, Huan Xu, and Jiashi Feng. 2021. Understanding and Resolving Performance Degradation in Deep Graph Convolutional Networks. In *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*. 2728–2737.

[83] Hao Zhu and Piotr Koniusz. 2021. Simple Spectral Graph Convolution. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

[84] 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 (2019), 2094–2105.