NeutronTP: Load-Balanced Distributed Full-Graph GNN Training with Tensor Parallelism

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

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

Zeyu Ling
Northeastern University, China
lingzeyu@stumail.neu.edu.cn

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

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

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

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

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

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

ABSTRACT

Graph neural networks (GNNs) have emerged as a promising direction. Training large-scale graphs that relies on distributed computing power poses new challenges. Existing distributed GNN systems leverage data parallelism by partitioning the input graph and distributing it to multiple workers. However, due to the irregular nature of the graph structure, existing distributed approaches suffer from unbalanced workloads and high overhead in managing cross-worker vertex dependencies.

In this paper, we leverage tensor parallelism for distributed GNN training. GNN tensor parallelism eliminates cross-worker vertex dependencies by partitioning features instead of graph structures. Different workers are assigned training tasks on different feature slices with the same dimensional size, leading to a complete load balance. We achieve efficient GNN tensor parallelism through two critical functions. Firstly, we employ a generalized decoupled training framework to decouple NN operations from graph aggregation operations, significantly reducing the communication overhead caused by NN operations which must be computed using complete features. Secondly, we employ a memory-efficient task scheduling strategy to support the training of large graphs exceeding single GPU memory, while further improving performance by overlapping communication and computation. By integrating the above techniques, we propose a distributed GNN training system NeutronTP. Our experimental results on a 16-node Aliyun cluster demonstrate that NeutronTP achieves 1.29×-8.72× speedup over state-of-the-art GNN systems including DistDGL, NeutronStar, and Sancus.

PVLDB Reference Format:
Xin Ai, Hao Yuan, Zeyu Ling, Qiange Wang, Yanfeng Zhang, Zhenbo Fu, Chaoyi Chen, Yu Gu, Ge Yu. NeutronTP: Load-Balanced Distributed Full-Graph GNN Training with Tensor Parallelism. PVLDB, 18(2): 173 - 186, 2024.
doi:10.14778/3705829.3705837

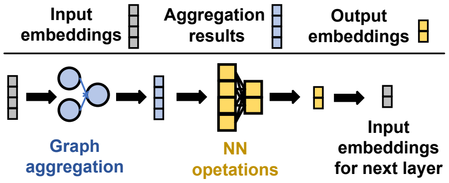


Figure 1: Illustration of a single-layer computation process in a GNN model, including graph aggregation operations and neural network (NN) operations.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/AiX-im/NeutronTP.

1 INTRODUCTION

Graph Neural Networks (GNNs) have demonstrated remarkable effectiveness in machine learning tasks [9, 45, 48, 49]. Graph-structured data serves as the input for GNNs, where each vertex is associated with a high-dimensional feature vector. The expressive power of GNNs stems from their ability to learn from relationships between data samples, whereas traditional DNNs are trained on individual samples [57]. Figure 1 illustrates the computational process of GNNs, involving graph aggregation and neural network (NN) operations. In each GNN layer, obtaining a vertex's new embedding entails aggregating its neighbors' embeddings from the previous layer (or neighbors' features at layer 0) and then applying NN operations. By iteratively performing these two steps, the GNN model can capture structural information from multi-hop neighbors.

Recently, full-graph GNN training, which involves training on the entire graph, has emerged as a promising GNN training method for its effectiveness brought by full-neighbor aggregation semantics and full-batch gradient descent [16, 25, 40, 42, 43]. Given the massive scale of graphs generated from applications, large-scale parallel and distributed computing becomes imperative for handling GNNs effectively [21, 32, 50]. A common approach to scaling GNN training on large-scale graph data is data parallelism, where the graph data is partitioned across different workers for parallel training [16, 25, 29, 36, 38, 40, 42, 43, 51, 54, 58].

Despite that partitioning graph data enables distributed GNN systems to handle large-scale data, it also constitutes a primary

Proceedings of the VLDB Endowment, Vol. 18, No. 2 ISSN 2150-8097.
doi:10.14778/3705829.3705837

constraint on the performance of GNN data parallelism. Firstly, as illustrated in Figure 2 (a), the irregular nature of graph data makes it challenging to ensure load balance when partitioning the workload. Many survey papers [21, 32, 50] highlight workload imbalance as a primary challenge, both in mini-batch and full-graph training. Secondly, the edges among data samples (i.e., vertices) lead to complex cross-worker vertex dependencies since graph aggregation may require neighbor data located on remote workers [16, 43]. Existing systems adopt methods such as cross-worker neighbor replication [11, 51, 54, 58] and neighbor communication [16, 25, 29, 38, 40] to manage vertex dependencies. As a result, the efficiency of GNN data parallelism is constrained by redundant computations and substantial communication overhead [16, 40, 43].

In this paper, we leverage tensor parallelism for distributed GNN training, eliminating cross-worker vertex dependencies by partitioning features instead of the graph structure. GNN tensor parallelism efficiently balances workload by evenly partitioning vertex features along dimensions. As illustrated in Figure 2 (b), GNN tensor parallelism divides vertex features according to the number of workers. Different workers are responsible for GNN training with feature slices of the same dimension, achieving complete computational load balance. GNN tensor parallelism involves two communication operations: `gather` and `split`. They collect complete embeddings for NN operations at each model layer, as NN operations include some non-linear operations that cannot be partially computed, and then redistribute the embedding slices back to the corresponding workers. These two communication operations involve all vertices. We only need to ensure that each worker handles the communication task of the same number of vertices to achieve load-balanced communication.

We further enhance the efficiency of GNN tensor parallelism by optimizing communication and memory overhead. Firstly, we employ a generalized decoupled training approach to reduce communication overhead, avoiding frequent execution of two communication operations in each model layer. Inspired by existing decoupled GNN training methods [4, 23, 56], we decouple NN operations from graph aggregation operations, confining `split` and `gather` operations to occur before and after consecutive graph operations, significantly reducing communication overhead. Additionally, we support decoupled training of complex models [37] through the precomputation of edge attention, providing generalized support for decoupled training. Secondly, we employ a memory-efficient task scheduling strategy to reduce memory overhead, mitigating out-of-memory errors caused by loading the entire graph topology during training. This strategy offers a lightweight subgraph logical partitioning method and further enhances performance by overlapping the computation and communication of different subgraphs.

By integrating the above techniques, we propose NeutronTP, a distributed GNN training system that achieves a well-balanced workload. We make the following contributions in this paper.

- We propose a distributed GNN training method based on tensor parallelism, which eliminates cross-worker vertex dependencies and achieves complete load balancing.
- We propose a generalized decoupling training method to separate NN operations from graph aggregation, significantly reducing communication frequency in GNN tensor parallelism.

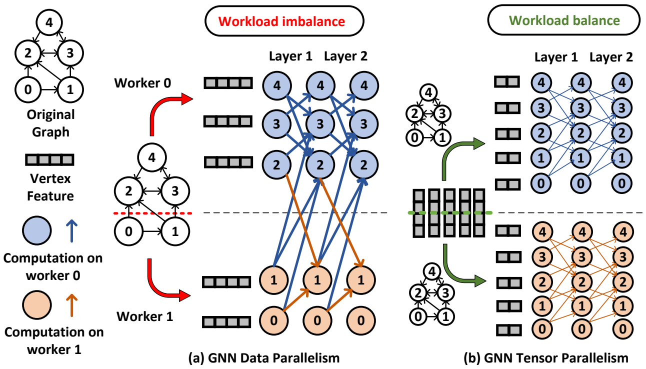


Figure 2: GNN data parallelism vs. GNN tensor parallelism. The thickness of the arrows and the size of the circles are positively proportional to the feature/embedding dimension and indicate the computation volume of GNN training.

- We propose a memory-efficient task scheduling strategy to support large-scale graph processing and overlap the communication and computation.
- We develop NeutronTP, a distributed system for full-graph GNN training that utilizes tensor parallelism to achieve fully balanced workloads and integrates a series of optimizations to achieve high performance.

We evaluate NeutronTP on a 16-node Aliyun GPU cluster. The experimental results show that NeutronTP outperforms the state-of-the-art GNN systems on homogeneous graphs, achieving 1.29×-6.36× speedups over DistDGL [54], 4.68×-8.72× speedups over NeutronStar [43], and 3.41×-4.81× speedups over SANCUS [29]. Additionally, NeutronTP achieves 6.15× speedups over DistDGLv2 [55] on heterogeneous graphs.

The rest of this work is organized as follows. Section 2 describes the background and motivations. Section 3 provides a detailed description of the proposed GNN tensor parallelism. Section 4 gives an overview of NeutronTP and describes the generalized decoupling training method and the memory-efficient task scheduling strategy. Section 5 presents results. Section 6 presents a discussion on related work. Section 7 concludes the paper.

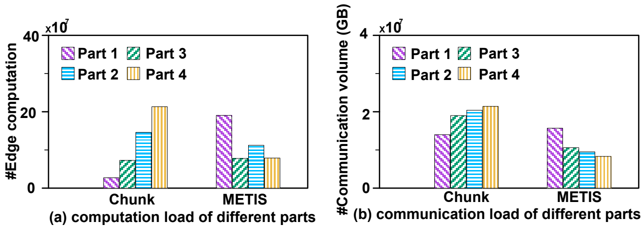
2 BACKGROUND AND MOTIVATION

2.1 Graph Neural Networks

Graph-structured data is input to GNNs, with each vertex having a high-dimensional feature vector. A typical GNN computes low-dimensional embeddings for vertices through multiple layers, aiding tasks like node classification and link prediction. Each layer includes an aggregation and an update phase [57]. In a GNN with L layers, during layer l's aggregation phase, each vertex v aggregates its neighbors' embeddings from layer $l-1$ and its own to produce a_v^l using an AGG function:

$$a_v^l = AGG(h_u^{l-1} | \forall u \in N_{in}(v) \cup \{v\}) \quad (1)$$

where $N_{in}(v)$ represents the incoming neighbors of vertex v, h_v^l represents the embedding vector of vertex v at l-th layer, and h_v^0 is the input feature of vertex v. Next, during the update phase, each



(a) computation load of different parts

(b) communication load of different parts

Figure 3: GNN training workload of 4 partitions under different partitioning methods. (2-layer GCN on Reddit)

vertex computes its output embedding vector h_v^l by applying an $UPDATE$ function to the aggregation result a_v^l:

$$h_v^l = UPDATE(W_v^l, a_v^l) \tag{2}$$

After L layers, each vertex's feature vector becomes a low-dimensional embedding of its neighbors up to L hops away.

Both the AGG and $UPDATE$ functions can be neural networks, which are updated during training. For simple GNN models, such as GCN [18], which only incorporate vertex-associated NN operations. The computational formulas for GCN are as follows:

$$AGG : a_v^l = \sum_{u \in N_{in}(v)} \left(\frac{1}{\sqrt{deg_{in}(v) \cdot deg_{out}(u)}} \cdot h_u^{l-1} \right) \tag{3}$$

$$UPDATE : h_v^l = \sigma(W^l a_v^l) \tag{4}$$

, where $deg_{out}(u)$ is the out-degree of vertex u, and $deg_{in}(v)$ denotes the in-degree of vertex of vertex v. The update phase applies standard DNN operations, including a matrix multiplication and a ReLU activation function (i.e., σ) to the aggregation result.

For complex GNN models, such as GAT [37], which include edge-associated NN operations and vertex-associated NN operations. The computational formulas for GAT are as follows:

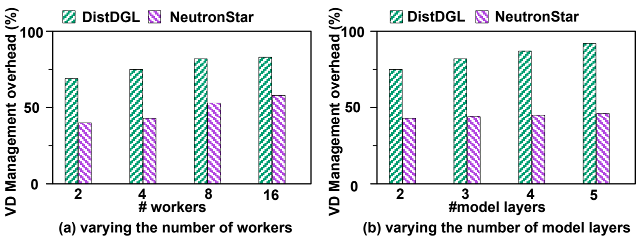
$$AGG : \begin{cases} a_{uv}^l = softmax(\widehat{\sigma}(a^T[W^l h_u^{l-1} || W^l h_v^{l-1}])) \\ a_v^l = \sum_{u \in N_{in}(v)} (a_{uv}^l h_u^{l-1}) \end{cases} \tag{5}$$

$$UPDATE : h_v^l = \sigma(W^l a_v^l) \tag{6}$$

The aggregate phase first needs to assign an edge weight a_{uv} for each incoming edge to vertex v. This process involves concatenating (i.e., $[\cdot||\cdot]$) and mapping (i.e., a^T) the parameterized representations of the source u and destination v to derive edge-wise attention coefficients. Then, these coefficients are fed to a LeakyReLU activation function (i.e., $\widehat{\sigma}$) and use a softmax function to compute normalized edge weight for subsequent neighborhood aggregation. The update phase is the same as GCN.

2.2 Distributed GNN Training with Data Parallelism

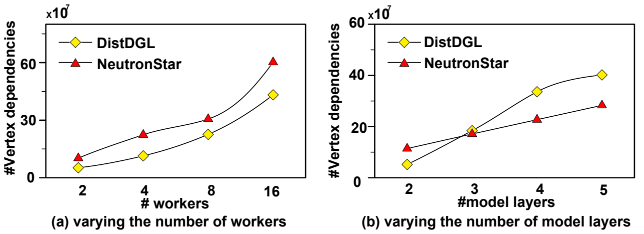
When dealing with large-scale graphs, single machines' limited memory and computational resources become bottlenecks for large-scale GNN training. Distributed computing offers sufficient computational resources, thereby enhancing training efficiency. Existing GNN systems [16, 24, 40, 42, 43, 54, 58] leverage data parallelism by partitioning the input graph and distributing it to multiple workers to train the same GNN model collaboratively. However, due to the graph aggregation operations in GNNs, which create vertex dependencies across these partitions, these graph partitions cannot be processed independently. In NeutronStar [43], the authors summarize the current GNN systems into two categories according to the way they manage vertex dependencies: Dependency Cache (DepCache) methods and Dependency Communication (DepComm) methods. The GNN systems [11, 51, 54, 58] employ the DepCache methods to replicate data of neighboring nodes from partitions outside to the local worker, enabling independent GNN training locally but incurring redundant computations. In contrast, systems [16, 25, 29, 38, 40] employing the DepComm methods collect data of neighboring vertices through communication from remote workers. While avoiding redundant computations, these systems incur necessary communication costs. The irregular nature of the graph causes extensive cross-worker vertex dependencies in data parallelism and thereby increases the difficulty of high-performance distributed GNN training. We summarize two major limitations below.



(a) varying the number of workers

(b) varying the number of model layers

Figure 4: The VD management overhead of DistDGL and NeutronStar



(a) varying the number of workers

(b) varying the number of model layers

Figure 5: The number of vertex dependencie of DistDGL and NeutronStar

Limitation #1:Workload imbalance. Distributed GNN training with data parallelism is prone to workload imbalance due to the skewed interconnection structure of graphs. To confirm our analysis, we employ Chunk-based graph partitioning and METIS-based graph partitioning in NeutronStar [43], evaluating the balance between computational and communication loads across each partition. The results are shown in Figure 3. The chunk-based graph partitioning strategy, employed by systems such as NeuGraph [24], ROC [16], and NeutronStar [43], divides the graph into chunks where nodes are arranged with consecutive IDs. Although this method achieves vertex balance, it may lead to significant workload imbalance because it does not account for the edge distribution among workers. On the other hand, the METIS algorithm, utilized by DistDGL [54], SANCUS [29], and BNS-GCN [38], aims to minimize cross-worker edges (i.e., edge-cuts) in its partitioning decisions. However, focusing on minimizing edge cuts does not guarantee balanced remote and local vertices within each partition, resulting in varied communication and computation loads across workers.

Limitation #2:High overhead in managing cross-worker vertex dependencies. Managing vertex dependency (VD) constitutes the primary overhead in GNN data parallelism. Furthermore, as the cluster scale expands or model layers deepen, the proportion of VD management overhead further increases. To confirm our analysis, we analyze VD management overhead in DistDGL [54] and NeutronStar [43] when altering the cluster scale or model layer. We measure the proportion of VD management overhead by accounting for communication time and redundant computation time and quantify VD's scale by calculating communication and redundant computation edges. As shown in Figure 4, across different cluster sizes and model layers, the VD management overhead in DistDGL and NeutronStar averages 80.6% and 46.5% of the total execution time, respectively. Furthermore, as shown in Figure 5, no matter increasing the number of partitions and workers or deepening models, the VD scale can be substantially increased. When the number of workers scales from 2 to 16, the time proportion of VD management overhead in DistDGL and NeutronStar increases by 1.21× and 1.45×, respectively, with the VD scale increasing by 8.1× and 6.2×, respectively. Similarly, as increases the model layer from 2 to 5, the time proportion of VD management overhead in DistDGL and NeutronStar increases by 1.22× and 1.06×, respectively, with the VD scale increasing by 7.7× and 3.0×, respectively.

2.3 Opportunity: Tensor Parallelism

Avoiding partitioning the graph structure across different workers is crucial to mitigating the limitations outlined above. Therefore, we exploit tensor parallelism for distributed GNN training, partitioning vertex features instead of the graph structure, thereby eliminating cross-worker vertex dependencies while ensuring load balancing. Inspired by DNN tensor parallelism [33] that supports DNN training with large-scale model data by partitioning model parameters instead of data samples. We extend tensor parallelism to distributed GNN training and change the partitioning target from model parameters to vertex data, as the memory overhead of GNN training mainly stems from vertex data (i.e., features and embeddings), while model data is typically small. Past DNN works employ multidimensional partitioning, such as 2D [47], 2.5D [41], and 3D [3] partitioning, to reduce the communication and memory overhead in tensor parallelism. These multidimensional partitioning methods further partition DNN data samples into multiple disjoint subsets for matrix operations with model parameter subsets. However, due to the unique graph aggregation training semantics of GNNs, further partitioning GNN data samples (i.e., graph data) reintroduces vertex dependencies. This renders us unable to directly apply past DNN tensor parallelism methods to optimize the efficiency of GNN tensor parallelism.

Recent works in vertical feature partitioning. We note that some recent studies [6, 11] explore vertical feature partitioning in distributed GNN training. However, they employ feature partitioning approach only in partial training processes and focus on reducing the feature communication overhead (See Section 6 for details). This feature partitioning approach cannot guarantee load balancing in the end-to-end training because most training still employs data parallelism. In contrast, we explore tensor parallelism throughout the entire training process, vertically partitioning both features and embeddings across all layers, achieving complete load balancing.

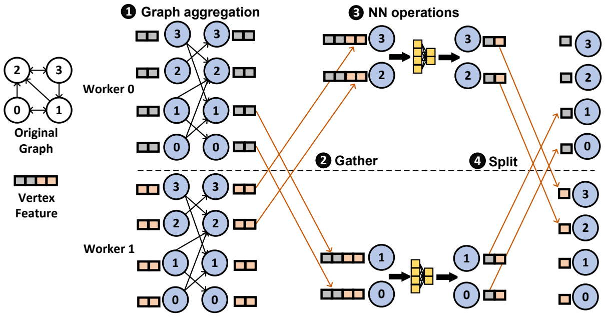


Figure 6: GNN tensor parallelism workflow for a single layer.

3 GNN TENSOR PARALLELISM

In this section, we provide a detailed exposition of the workflow involved in GNN tensor parallelism and elucidate its advantages and challenges through workload analysis.

3.1 GNN Tensor Parallelism Workflow

Unlike GNN data parallelism, which partitions the graph topology, GNN tensor parallelism vertically partitions vertex features along dimensions, where each worker is responsible for the complete graph GNN training of different feature slices with the same dimensional size. Figure 6 illustrates the single-layer training workflow for GNN tensor parallelism. Initially, the feature vectors of the vertices are evenly partitioned among all workers according to their feature dimensions, i.e., each worker holds $\frac{D}{N}$ dimensions of the feature vectors where D is the total number of vertex feature dimensions, and N is the number of workers. The GNN tensor parallelism directly leverages the structure of the raw graph for its computations. Specifically, at each layer of the GNN model, every vertex aggregates information from all its neighboring vertices along the incoming edges and then applies NN operations.

Each worker stores the complete graph structure and conducts full-neighbor aggregation operations locally (❶). Before the start of NN computations, a `gather` operation is performed to obtain complete vertex embeddings because nonlinear operations in NN computations cannot be partially computed (❷). To ensure the uniform distribution of NN computation tasks and communication tasks across different workers, each worker is responsible for computing and communicating with $\frac{V}{N}$ vertices, where V represents the total number of vertices and N denotes the number of workers. All workers initiate `gather` operations simultaneously, sending local embedding slices to the corresponding workers. Subsequently, each worker commences the NN computation tasks for local vertices (❸). Finally, upon completion of NN computations, all workers initiate `split` operations simultaneously to re-segment local embeddings and send them to the corresponding workers to continue the next layer of GNN training (❹). The `gather` and `split` operations are implemented by collective communication libraries such as NCCL [28] and Gloo [8].

3.2 Workload Analysis

GNN tensor parallelism achieves a well-balanced computation and communication load by evenly partitioning vertex features along dimensions. In the computation process, each worker handles the full-graph aggregation operations of the same dimension feature slices and performs NN computations for the same number of vertices. The time and space complexities of graph aggregation operation are $O(E\frac{D}{N})$ and $O(V\frac{D}{N})$, respectively. The time and space complexities of NN operation are $O(\frac{V}{N}D^2)$ and $O(\frac{V}{N}D)$, respectively, where V and E represent the total number of vertices and edges, N denotes the number of workers, and D denotes the feature dimension, for simplicity, we assume that the feature dimensions are uniform across all layers. In the communication process, during the `gather` phase, each worker receives the embedding slices of local NN computation vertices from the other $N-1$ workers. The time and space complexities are $(N-1)\cdot\frac{V}{N}\cdot\frac{D}{N}\approx O(\frac{VD}{N})$ and $O(\frac{VD}{N^2})$, respectively. The `split` phase can be understood as the inverse process of the `gather` phase, and it has the same time and space complexity as the `gather` phase.

We further analyze the total computation and communication load of GNN tensor parallelism. GNN tensor parallelism maintains the same total computational load as single-machine full-graph training without any redundant computations. Regarding communication load, GNN tensor parallelism performs `split` and `gather` operations at each layer to communicate embedding slices with the other $(N-1)$ workers. The total communication load for GNN tensor parallelism is $N\times 2(N-1)\frac{VD}{N^2}L\approx 2VDL$, where L denotes the number of model layers. The total communication load of GNN data parallelism is $\sum_{i=1}^{N}|R_i|DL$, where R_i denotes the remote vertices of worker i. As the number of workers increases, the remote vertices $(\sum_{i=1}^{N}|R_i|)$ rise significantly, often exceeding $2V$ [42]. In contrast, the total communication volume in GNN tensor parallelism remains relatively constant with worker increases, typically having a lower communication load than data parallelism.

In GNN tensor parallelism, more memory is used to replicate the graph structure to eliminate cross-worker vertex dependencies and ensure load balancing. This overhead is generally acceptable since the primary memory consumption in GNN training comes from vertex data, including features, embeddings, and gradients [42]. For example, in the Ogbn-paper dataset, the graph topology size is 6.4 GB, while vertex features occupy 82.7 GB. GNN tensor parallelism distributes all vertex data across different workers by either dimension or vertex count.

3.3 Challenges

The benefits of GNN tensor parallelism come with challenges that must be overcome to fully exploit acceleration opportunities.

Challenge #1: Frequent collective communication. Compared to GNN data parallelism, GNN tensor parallelism involves more rounds of communication (i.e., twice per layer) to gather and split vertex embeddings. This frequent communication may impact computation efficiency due to substantial layer-wise synchronization. Therefore, reducing the overall communication frequency is crucial for effectively implementing GNN tensor parallelism.

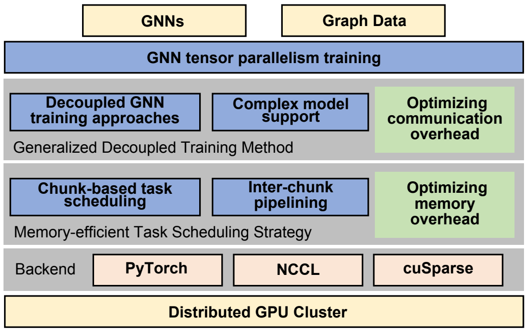

GNNs
Graph Data
GNN tensor parallelism training
Decoupled GNN training approaches
Complex model support
Optimizing communication overhead
Generalized Decoupled Training Method
Chunk-based task scheduling
Inter-chunk pipelining
Optimizing memory overhead
Memory-efficient Task Scheduling Strategy
Backend
PyTorch
NCCL
cuSparse
Distributed GPU Cluster

Figure 7: NeutronTP system overview.

Challenge #2: Processing the entire graph on a single worker. GNN tensor parallelism becomes impractical when a single GPU memory cannot accommodate the entire graph and corresponding embedding slices. To support the training of large-scale graphs, we need to offload the training data to the CPU main memory. This requires the further design of a task scheduling strategy and consideration of integration with pipeline techniques to minimize latency when accessing data in CPU main memory.

4 THE NEUTRONTP

We present NeutronTP, a distributed system for full-graph GNN training that utilizes tensor parallelism and addresses the challenges outlined in Section 3.3 through two critical functions. Figure 7 provides an architectural overview of NeutronTP.

Generalized decoupled training method. The main reason for challenge #1 lies in the coupled training patterns of NN and graph aggregation operations, which frequently switch between using complete embeddings and embedding slices. Therefore, we decouple NN operations from graph aggregation operations, restricting collective communication to the beginning and end of consecutive graph operations, thereby reducing communication frequency. To address the issue that existing decoupling training methods [4, 23, 56] do not support complex GNN models involving edge-associated NN operations, we further explore a decoupling approach for these operations, providing a generalized decoupling training method. Specifically, before initiating the graph aggregation, we precompute edge-associated NN operations for each edge.

Memory-efficient task scheduling strategy. To address challenge #2, we further partition subtasks within each worker to perform fine-grained GNN training. NeutronTP employs a memory-efficient task scheduling strategy that reduces runtime memory consumption through chunk-based task scheduling and further enhances performance by inter-chunk pipeline. Specifically, within each worker, we partition the entire graph logically into multiple chunks that can fit into GPU memory, with each chunk containing a set of vertices with contiguous IDs along with all their incoming edges. During training, each worker schedules chunks onto the GPU in the same order, maintaining load balance for GNN tensor parallelism. Without compromising the layer-wise synchronization barrier, the inter-chunk pipeline overlaps the computation and communication of different chunks.

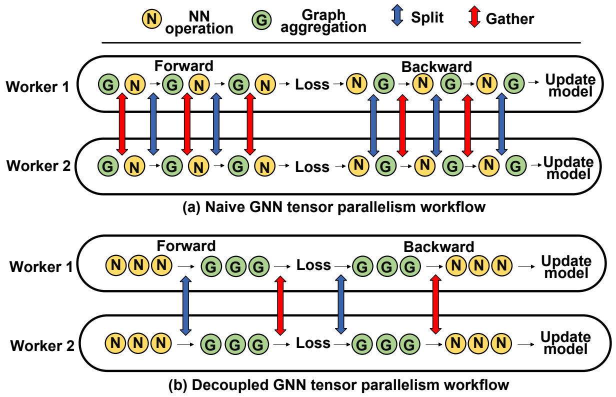
4.1 Generalized Decoupled Training Method

GNN tensor parallelism requires frequent collective communication for gathering and splitting embeddings by dimensions between NN computation (requiring the embedding split by vertices) and graph propagation (requiring the embedding split by dimensions), which results in substantial layer-wise data synchronization overhead. To improve communication efficiency for high performance training, NeutronTP employs a decoupled training method that reorganizes the execution order by moving NN computation to the beginning or end of the computation graph. This design reduces the collective communication of switching data organizations.

4.1.1 Decoupled GNN Training Approaches. Previous works [13, 45, 52] have indicated that the expressive power of GNNs originates from NN operations and graph operations themselves, not their coupling execution. Moreover, the coupling execution of graph aggregation and NN operations may lead to over-smoothing problems when training deep GNN models [19, 23, 52]. Therefore, some decoupled GNN approaches [19, 23, 30, 34, 45] advocate separating NN operations from graph aggregation. This decoupling method has been shown to effectively enhance both model accuracy and scalability in deep model training. A recent study [56] has also applied decoupled training methods to dynamic GNN training, achieving significant scalability and performance. However, existing decoupling training methods [19, 23, 30, 34, 45] typically focus only on decoupling vertex-associated NN operations and do not support decoupling edge-associated NN operations. For complex models incorporating edge-associated NN operations, such as GAT [37], graph aggregation in itself may introduce non-linear operations that cannot be partially computed.

We extend the decoupled GNN training method by precomputing all the attention coefficients required for each edge. This approach further decouples edge-associated NN operations from graph aggregation, thereby supporting the training of complex models. Specifically, before the graph aggregation operation starts in this round, it computes attention coefficients using data parallelism. Since the computation of edge attention coefficients requires complete vertex embeddings and involves all edges, we employ data parallelism to compute attention coefficients for all incoming edges of local vertices on each worker. After the computation is completed, the attention coefficients are shared among all workers. For other stages, the approach remains consistent with simple GNN models, allowing the use of GNN tensor parallelism. With the above design, we can perform edge-associated NN computations before initiating the graph aggregation operation.

4.1.2 Decoupled GNN Tensor Parallelism. For the given input GNN model, NeutronTP provides its corresponding decoupled training mode and applies tensor parallelism for training. Specifically, after specifying the model layers L, in each epoch, NeutronTP first performs L rounds of NN operations on each vertex to obtain the low-dimensional vertex embeddings. For complex models incorporating edge-associated NN operations, NeutronTP further computes attention coefficients for all edges to be used in subsequent graph aggregation operations. Upon completing the L rounds of NN operations, NeutronTP performs a `split` operation to restore tensor parallelism, where each worker holds partial embedding dimensions for all vertices. Subsequently, each worker utilizes embedding slices to complete L rounds of full-graph aggregation operations. Upon completion of the graph aggregation operations, the forward propagation is executed, followed by a `gather` operation to collect complete vertex embedding to ensure the correct execution of the loss function. Backward propagation follows the inverse process of forward propagation, where NeutronTP still needs to perform `split` and `gather` operations before and after L rounds of graph aggregation operations, respectively.



(a) Naive GNN tensor parallelism workflow

(b) Decoupled GNN tensor parallelism workflow

Figure 8: An illustrative example for showing communication frequency of naive GNN tensor parallelism and decoupled GNN tensor parallelism (3-layer GNN).

Figure 8 illustrates the comparison in overall communication frequency between naive GNN tensor parallelism and decoupled GNN tensor parallelism. For a 3-layer GNN model, the naive GNN tensor parallelism requires 10 rounds of collective communication, and the frequency of communication increases linearly as the number of model layers increases. In contrast, the decoupled GNN tensor parallelism only requires 4 rounds of collective communication, regardless of the number of model layers. Additionally, after multiple NN operations, the vertex embeddings involved in graph aggregation typically have lower dimensions compared to the raw features, further reducing the collective communication overhead.

Decoupled GNN tensor parallelism is particularly effective for message-passing based GNNs such as GCN [18], GraphSAGE [12], GAT [37], and GIN [46]. These models rely on updating and aggregating vertex features across the graph, making them ideal for GNN tensor parallelism, which efficiently partitions features and balances loads. The decoupled training method reduces communication overhead by decoupling update and aggregation processes. However, our approach may not directly benefit GNNs that do not rely on message passing, such as spectral-based GNNs (e.g., ChebNet [35]) and GNNs with global attention mechanisms (e.g., Graph Transformer [7]). By focusing on message-passing based GNNs, NeutronTP enhances training efficiency and scalability, demonstrating broad applicability within widely-used GNN models.

4.1.3 Convergence Analysis. In this section, we provide a theoretical analysis of convergence guarantees for NeutronTP. Decoupled GNN training methods have been widely used by machine learning systems [19, 23, 30, 34, 45], and we present the theoretical analysis referring to the APPNP [19] and DAGNN [23]. The expressive

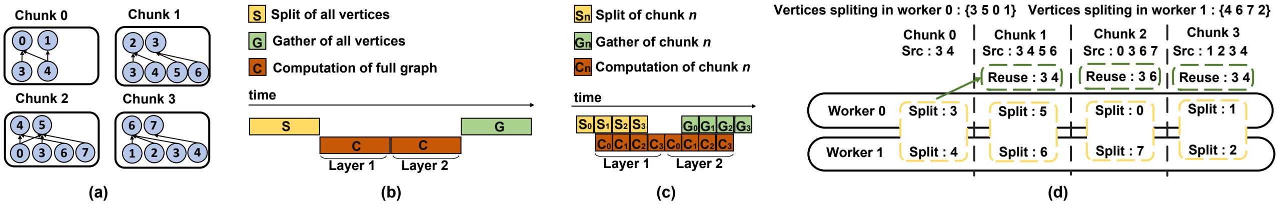


Figure 9: (a) An example of graph data partitioned into four chunks. (b) An example of the sequential execution of different phases. (forward computation for a 2-layer GNN) (c) An example of inter-chunk pipelining. (forward computation for a 2-layer GNN) (d) An example of partitioning the `split` operation into chunk-level communication tasks.

power of GNNs originates from NN operations and graph operations themselves, not their coupled execution. Thus, the decoupled GNN training method separates and sequentially executes these operations while maintaining comparable expressive power to original GNNs. Experimental results from DAGNN [23] demonstrate that training on the input vertex feature with NN operations can achieve a certain level of accuracy in node classification and applying graph aggregations in a decoupled manner can further enhance the performance. Therefore, we provide Assumption 1 as follows:

- **Assumption 1** The initial features have sufficient information for the machine learning task, and the graph aggregation operation can help the model learn structural information.
- **Theorem 1** Under Assumption 1, the decoupled GNN training method separates graph operations from NN operations. The convergence of the decoupled GNN training can be guaranteed by the convergence of the NN and graph operations.

Decoupled GNN training uses the iterative equation

$$\hat{L} = UPDATE(X) = MLP^k(X), \tag{7}$$

$$Z^0 = \hat{L}, \tag{8}$$

$$Z^k = AGG(Z) = \gamma \hat{A} Z^{k-1}, \tag{9}$$

where k represents the number of times the NN operations and graph operations are executed, X is input features. $UPDATE(\cdot)$ represents NN operation, i.e., X is fed into a multi-layer neural network to obtain the embeddings $\hat{L}$. To learn the structural information of the graph, decoupled GNN training performs multi-layer graph operations, i.e., $AGG(\cdot)$. $\hat{A}$ is symmetrically normalized adjacency matrix ($\hat{A} = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$, where A represents the adjacency matrix, D represents the degree matrix. Here, $\tilde{A}$ and $\tilde{D}$ correspond to $A + I$ and $D + I$, respectively, with I being the identity matrix). γ ($\gamma \in (0, 1]$) is the weight of edges, which can be computed through a self-attention mechanism as in GAT or weighted neighbor convolution as in GCN.

$MLP^k(X)$ is the convergent function. This is because multiple NN operations can be viewed as a traditional deep neural network model (i.e., multi-layer perceptron), whose convergence properties are well-established [1, 27]. Therefore, we only need to consider $AGG(Z)$. After k iterations of graph propagation, $AGG(Z)$ can be expressed as:

$$Z^k = \gamma^k \hat{A}^k \hat{L}. \tag{10}$$

If we take the limit $k \longrightarrow \infty$ in Formulation 10, the result tends to 0 since $\gamma_k \in (0, 1]$, the eigenvalues of $\hat{A}$ are the same as those of $\tilde{A}$, which can be proven through Gershgorin circle theorem [44] that the maximum eigenvalue is 1, i.e., $\|\hat{A}\| \leq 1$, and $\hat{L}$ is convergent, resulting in

$$Z^\infty = \gamma^\infty \hat{A}^\infty \hat{L} \longrightarrow 0, \tag{11}$$

the above concludes that the convergence is guaranteed.

In Section 5.7, we also evaluate the accuracy of decoupled GNN training, which performs comparably to coupled GNN training across different datasets.

4.2 Memory-efficient Task Scheduling Strategy

4.2.1 Chunk-based task Scheduling. To address challenge #2, NeutronTP employs chunk-based task scheduling, where the global graph topology is partitioned on each worker, and different workers simultaneously schedule the same subgraph. This does not incur cross-worker vertex dependencies, as each worker partitions the entire graph locally using the same strategy. All workers execute communication and computation tasks for each chunk in the same order to ensure load balancing. As shown in Figure 9 (a), each chunk comprises a subset of destination vertices with contiguous vertex IDs and all in-edges for each vertex, facilitating independent full-neighbor aggregation for each chunk. In GNN tensor parallelism, using chunk-based partitioning offers two advantages. Firstly, using chunk-based partitioning to obtain subgraphs is lightweight, as the graph topology only needs to be logically partitioned within local workers without modifying any physical storage locations. Secondly, ensuring load balancing merely requires scheduling chunks for computation in the same order across all workers, thus eliminating the need to handle load balancing between chunks.

It is worth mentioning that we only need to group the in-edges of destinations, as complex aggregations are performed only during the forward pass. During the backward pass, source vertices can accumulate gradients along out-edges through summation. Leveraging the associativity of summation operations, multiple copies of source vertices in different chunks can independently compute gradients and then be summed afterward. During the actual partitioning process, unlike traditional distributed training systems where chunks are divided among multiple workers, we do not specify a predetermined number of chunks to partition. Generally, to better utilize GPU resources and reduce scheduling overhead, we should aim to make each chunk as large as possible.

When using chunks as the scheduling unit, full-graph GNN training requires consideration of intermediate result management. Intermediate results are generated during forward computation and consumed in the gradient computation during backward pass [42]. To avoid exceeding the GPU memory capacity with intermediate results, they need to be sent back to CPU memory after forward computation and returned to GPU during backward computation.

Algorithm 1 Workflow of NeutronTP for a single epoch

```
Input: Graph $G(V,E)$, Feature $\mathbf{h}^0$, Initial parameterized GNN layers
    $\{\mathbf{W}^0, \mathbf{W}^1 \dots \mathbf{W}^{L-1}\}$, cluster size $m$, chunk number $n$
Output: Updated parameterized GNN layers $\{\mathbf{W}^0, \mathbf{W}^1 \dots \mathbf{W}^{L-1}\}$
 1: $\{G_j | 0 \leq j < n\}$=chunk_partition($G$, $n$)
 2: $\{V_i, \mathbf{h}_i^0 | 0 \leq i < m\}$=distribute_vertex($\{G_j | 0 \leq j < n\}$, $\mathbf{h}^0$, $m$)
 3: for worker $i = 0$ to $m-1$ do in parallel
 4:     for layer $l = 0$ to $L-1$ do
 5:         $\mathbf{h}^{l+1}$ of $V_i$ = worker($i$).UPDATE($\mathbf{W}^l$, $\mathbf{h}^l$ of $V_i$)
 6:     for layer $l = 0$ to $L-1$ do
 7:         for chunk with id $j = 0$ to $n-1$ do
 8:             if layer == 0 then
 9:                 $\mathbf{h\_cut}_i^0$ of $N_j \leftarrow$Split($\mathbf{h}^L$ of $N_j$)
10:             $\mathbf{h\_cut}_i^{l+1}$ of $V_j$ = worker($i$).AGG($\mathbf{h\_cut}_i^l$ of $N_j$, $G_j$)
11:             if layer == $L-1$ then
12:                 $\mathbf{h}^L$ of $V_j \leftarrow$Gather($\mathbf{h\_cut}_i^L$ of $V_j$)
13: loss = downstream_task($\mathbf{h}^L$)
14: $\nabla\mathbf{h}^L$ = loss.backward()
15: for worker $i = 0$ to $m-1$ do in parallel
16:     for layer $l = L-1$ to 0 do
17:         for chunk with id $j = 0$ to $n-1$ do
18:             if layer == $L-1$ then
19:                 $\nabla\mathbf{h\_cut}_i^0$ of $V_j \leftarrow$Split($\nabla\mathbf{h}^L$ of $V_j$)
20:             $\nabla\mathbf{h\_cut}_i^l$ of $N_j$ = worker($i$).AGG($\nabla\mathbf{h\_cut}_i^{l+1}$ of $V_j$, $G_j$)
21:             if layer == 0 then
22:                 $\nabla\mathbf{h}^L$ of $N_j \leftarrow$Gather($\nabla\mathbf{h\_cut}_i^L$ of $N_j$)
23:     for layer $l = L-1$ to 0 do
24:         $\nabla\mathbf{h}^{l+1}$ of $V_i$ = worker($i$).UPDATE($\mathbf{W}^l$,$\nabla\mathbf{h}^l$ of $V_i$)
25: for layer $l = 0$ to $L-1$ do
26:     sync_and_update ($\mathbf{W}^l$) //parameter update
```

This frequent host-GPU data exchange may impact overall performance. Fortunately, benefiting from the decoupled GNN training approach, NeutronTP avoids generating intermediate results during consecutive graph aggregation operations. Thus, we only need to handle intermediate results produced during the NN operations. Considering that the computational overhead of GNN training typically lies in graph aggregation operations rather than NN operations [50], we push down NN operations to be executed on the CPU. This not only reduces a significant amount of host-GPU communication for intermediate results but also leverages CPU resources.

4.2.2 Inter-chunk Pipelining. Based on chunk-based task scheduling, we can further overlap the communication and computation processes of each chunk. Our plan involves overlapping each `split` and `gather` operation with the adjacent graph aggregation operation without disrupting the original layer-wise synchronization. As illustrated in Figure 9 (b), the full graph computation process and two collective communication operations need to be executed serially to ensure layer-wise synchronization. Benefiting from chunk-based task scheduling, we can further partition the two collective communication operations into chunk-level communication tasks to overlap with computation tasks. As shown in Figure 9 (c), the `split` operation pre-splits the embeddings of src vertices for each chunk, while the `gather` operation collects the embedding slices of dst vertices for each chunk. This chunk-level communication task requires further design to avoid redundant communication. Firstly, for each chunk, we evenly distribute its vertex-related communication tasks across all workers to ensure communication load balancing. However, due to duplicate src vertex sets within chunks, this may result in redundant communication operations for some vertices. Therefore, when assigning communication tasks for each chunk, NeutronTP checks whether the vertices within the current chunk have already been communicated in previous chunks. As illustrated in Figure 9 (d), upon detecting that the vertices have been communicated, NeutronTP directly reuses the previous communication results. After traversing each chunk and determining the vertex-related communication tasks assigned to each worker, we aggregate them into a non-redundant vertex set. Each worker executes the relevant communication tasks and NN computation tasks based on its local vertex set.

4.3 Overall Execution Flow in NeutronTP

Algorithm 1 outlines the overall execution flow. To begin with, NeutronTP employs a chunk-based task scheduling strategy to partition the graph topology into a series of chunks G_j, each containing a set of disjoint vertex sets V_j and their in-neighbor sets N_j, where j is the chunk id (line 1). Subsequently, NeutronTP evenly distributes the vertex-related tasks within each chunk across all workers. After deduplication, each worker obtains its local vertex work queue V_i, where i is the worker id, utilized for subsequent execution of NN computations and communication operations (line 2). During the forward pass, each worker first completes L rounds of NN operations to obtain local vertex embeddings (line 5). Next, all workers schedule chunks in the same order and begin by splitting the embeddings of their in-neighbor sets, i.e., $\mathbf{h}^L$ of N_j (Line 9). Upon completion of the splitting phase, each worker initiates graph aggregation operations using its local embedding slices, i.e., $\mathbf{h\_cut}_i^l$ of N_j (Line 10). Following L rounds of graph aggregation within the chunk, all workers gather the complete embeddings of destination vertices in each chunk, i.e., $\mathbf{h}^L$ of V_j (Line 12). Upon completion of all chunk computations and communication tasks, each worker initiates downstream tasks and computes gradients $\nabla\mathbf{h}^L$ (lines 13-14). The backward pass is the reverse process of the forward pass, requiring embedding `split` and `gather` operations before and after the graph aggregation operation, respectively.

5 EVALUATION

5.1 Experimental Setup

Environments. Our experiments are conducted on the Aliyun ECS cluster with 16 GPU nodes. Each node (ecs.gn6i-c16g1.4xlarge instance) is equipped with 16 vCPUs, 186GB DRAM, and 1 NVIDIA Tesla T4 GPU, running Ubuntu 18.04 LTS OS. The network bandwidth is 15 Gbps/s. Libraries CUDA 11.1, OpenMPI-3.0.2, PyTorch v1.5 backend, and cuDNN 7.0 are used in both clusters.

Datasets and GNN algorithms. Table 1 lists six graph datasets that we used in our evaluation, including three popular GNN datasets: Reddit [12], Ogbn-products [15], and Ogbn-paper [15], and one graph dataset Friendster [20]. Ogbn-mag [15] and Mag-lsc [14] are two heterogeneous graphs that we use to evaluate the heterogeneous GNN training. For graphs without ground-truth properties (Friendster), we use randomly generated features, labels, training

Table 1: Dataset description

Dataset	**|V|**	**|E|**	**ftr. dim**	**#𝕃**	**hid. dim**
Reddit (RDT)	0.23M	114M	602	41	256
Ogbn-products (OPT)	2.45M	61.68M	100	47	64
Ogbn-paper (OPR)	111.1M	1.616B	128	172	128
Friendster (FS)	65.6M	2.5B	256	64	128
Ogbn-mag (MAG)	1.9M	21M	128	349	64
Mag-lsc (LSC)	244.2M	1.7B	768	153	256

(65%), test (10%), and validation (25%) set division. We use two popular GNN models with different computation patterns, GCN [18] and GAT [37]. All of them are in a 2-layer structure. The vertex feature dimensions, hidden layer dimensions, and the number of labels of datasets are listed in Table 1.

Competitor systems. In our performance evaluation, we compare NeutronTP with two kinds of GNN training systems, i.e., mini-batch system and full-graph system. For the mini-batch system, we compare NeutronTP with DistDGL [54], a representative deep learning library for graphs. DistDGL relies on data sampling to reduce computation cost [54], which is set to execute a (25, 10) neighborhood sampling for the training. In such a configuration, DistDGL picks a maximum of 10 neighbors for the first hop of a vertex, and then a maximum of 25 neighbors for each of those 10. For the full-graph system, we compare NeutronTP with NeutronStar [43] and Sancus [29]. Neutronstar [43] designs hybrid vertex dependency management for a more balanced use of communication and computational resources. Sancus [29] reuses historical embedding for cross-worker vertices to reduce communication.

5.2 Overall Comparison

We conduct a comprehensive performance comparison with NeutronStar [43], DistDGL [54], and Sancus [29] on a 16-node cluster. We record the computation time, communication time, and the per epoch runtime. "max" indicates the longest computation and communication time among all workers, which typically determines the actual runtime of distributed training. Similarly, "min" indicates the shortest computation time and communication time among all workers. For systems employing pipelining techniques to overlap computation and communication (i.e., NeutronStar [43] and NeutronTP), the sum of the longest computation and communication time exceeds the total runtime. By meticulously logging these times, we gain insights into the workload of distributed training. The experimental results are summarized in Table 2.

Comparison with mini-batch system. Compared to DistDGL [54], NeutronTP exhibits superior performance across most datasets, achieving up to 6.23× speedup. The METIS partitioning used by DistDGL may result in certain workers having more vertices, leading to more frequent access by other workers. As shown in Table 2, DistDGL exhibits an imbalance in both computation and communication times among different workers, with disparities reaching up to 1.38× and 2.2×, respectively, thereby causing resource wastage in the less loaded workers. NeutronTP mitigates these issues by adopting tensor parallelism, achieving more balanced workloads while avoiding redundant computations and communications. On the Ogbn-paper dataset, DistDGL demonstrates a better performance as it trains only on 1.1% of the total vertices, resulting in reduced computational load compared to NeutronTP.

Table 2: Comparison with different systems a 16-node ECS cluster. "max" and "min" indicate the longest and shortest computation or communication times among workers, respectively. "total" indicates the per-epoch runtime.

Model	Dataset	System	Runtime (s)				
			Computation		Communication		**total**
			max	**min**	**max**	**min**	
GCN	RDT	DistDGL	0.15	0.11	2.12	1.38	2.27
		NeutronStar	0.86	0.77	1.17	0.87	1.92
		Sancus	0.35	0.31	0.82	0.71	1.17
		NeutronTP	0.39	0.38	0.19	0.18	**0.40**
	OPT	DistDGL	0.26	0.16	2.82	1.28	3.18
		NeutronStar	2.71	1.42	2.89	1.78	4.45
		Sancus	0.86	0.36	1.59	1.22	2.45
		NeutronTP	0.46	0.44	0.24	0.22	**0.50**
	OPR	DistDGL	5.35	4.19	20.1	11.21	**25.4**
		NeutronStar	-	-	-	-	OOM
		Sancus	-	-	-	-	OOM
		NeutronTP	95.8	95.2	53.6	49.4	134.4
	FS	DistDGL	136.4	118.9	323.4	197.5	459.5
		NeutronStar	-	-	-	-	OOM
		Sancus	-	-	-	-	OOM
		NeutronTP	74.3	73.5	32.9	29.4	**90.5**
GAT	RDT	DistDGL	0.75	0.52	2.17	1.49	2.92
		NeutronStar	-	-	-	-	OOM
		Sancus	-	-	-	-	OOM
		NeutronTP	0.92	0.88	0.48	0.42	**1.29**
	OPT	DistDGL	1.17	0.94	2.76	1.29	3.93
		NeutronStar	8.72	5.98	15.9	8.29	22.4
		Sancus	-	-	-	-	OOM
		NeutronTP	2.17	1.94	1.06	0.95	**3.03**
	OPR	DistDGL	8.40	6.48	21.1	11.7	**29.5**
		NeutronStar	-	-	-	-	OOM
		Sancus	-	-	-	-	OOM
		NeutronTP	154.3	136.4	98.9	84.7	235.4
	FS	DistDGL	157.8	110.4	419.8	283.7	577.6
		NeutronStar	-	-	-	-	OOM
		Sancus	-	-	-	-	OOM
		NeutronTP	115.2	92.5	72.1	61.4	**167.9**

Comparison with full-graph system. Compared to NeutronStar [43] and Sancus [29], NeutronTP demonstrates superior performance across all datasets, achieving speedups of up to 8.72× and 4.81×, respectively. They are both constrained by imbalanced workloads and communication overhead resulting from extensive cross-worker vertex dependencies. The computation and communication time gap between different workers in NeutronStar can reach up to 1.91× and 1.62×, respectively. For Sancus, these gaps can reach up to 2.38× and 1.18×, respectively. NeutronStar exhibits long communication times due to its chunk-based partitioning strategy, which has more cross-worker vertex dependencies, as described in Section 2.2. Sancus reduces communication overhead by reusing historical embeddings. However, when updating historical embeddings, Sancus sequentially triggers each worker to broadcast embeddings, sending all local embeddings to all workers, regardless of whether other partitions contain these vertices. This not only leads to prolonged waiting times for workers but also results in considerable redundant communication. Both NeutronStar and Sancus encounter out-of-memory errors when dealing with large-scale graphs and complex model due to their lack of intra-worker task scheduling strategies. The advantages of NeutronTP stem from two main factors: (1) Tensor parallelism training achieves a more balanced computation and communication load, while the decoupled GNN training approach significantly reduces communication overhead. (2) The chunk-based task scheduling strategy

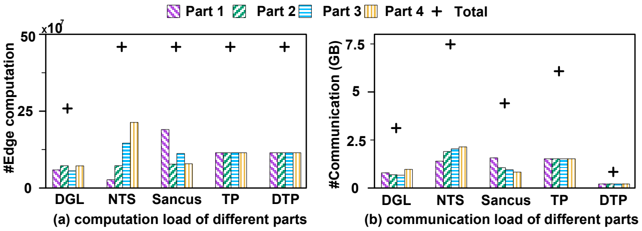


Figure 10: The computation/communication load of each partition in different systems. "TP" indicates NeutronTP with naive GNN tensor parallelism and "DTP" indicates NeutronTP with decoupled GNN tensor parallelism.

partitions local graph data into multiple chunks smaller than GPU memory and loads them sequentially, overlapping computation and communication while avoiding out-of-memory errors.

5.3 Computation and Communication Analysis

We evaluate the computational and communication loads of different systems to illustrate the load-balanced advantages of NeutronTP. Experiments are performed on the Reddit dataset using DistDGL, NeutronStar (NTS), Sancus, NeutronTP with naive GNN tensor parallelism (TP), and NeutronTP with decoupled GNN tensor parallelism (DTP). We train a 2-layer GCN on a 4-node cluster and record the workload for each worker as well as the overall workload. The communication load was measured based on the amount of data transferred, while the computational load was determined by the number of edges involved in the computation. Since NeutronTP utilizes feature slices during computation, whereas other systems use complete features, we scale the edge computation of NeutronTP accordingly. Experimental results are shown in Figure 10.

Regarding computational load, NeutronTP achieves complete load balancing by partitioning vertex data, as shown in Figure 10 (a). Regarding total computation, NeutronTP is comparable to other full-graph systems but surpasses the DistDGL. Although DistDGL reduces the computation through sampling, the sampling process incurs significant overhead and leads to decreased efficiency as the model layer increases (see details in Section 5.5). As illustrated in Figure 10 (b), regarding communication load, NeutronTP ensures load balance by assigning each worker an equal number of vertices for embedding `gather` and `split`. However, naive tensor parallelism requires embedding `gather` and `split` operations at each layer, leading to frequent communication. Benefiting from the decoupled tensor parallelism training approach, NeutronTP reduces the overall communication frequency while converting the communication entities into lighter-weight vertex embeddings, significantly reducing communication volume by up to 7.23 ×.

5.4 Performance Gain Analysis of NeutronTP

We analyze the performance gain of GNN tensor parallelism (TP), decoupled training method (DT), chunk-based task scheduling (CS), and inter-chunk pipelining (IP) on the GCN model with four datasets. To ensure a fair comparison, we start with a data parallelism baseline based on the NeutronTP codebase and gradually integrate the four optimization methods. The data parallelism baseline employs a chunk-based approach for graph partitioning. Figure 11 shows the normalized speedups. Compared to the baseline, the baseline+CS addresses the memory requirements of large-scale data by further partitioning chunks within each worker. Compared to the baseline+CS, TP achieves speedups ranging from 1.92× to 2.45× by implementing a more balanced workload. On the Friendster dataset, TP achieves the highest speedup, attributed to its inherent power-law distribution as a social network graph. The chunk-based partitioning strategy exacerbates severe workload imbalances in such graphs. Compared to the baseline+CS+TP, DT achieves speedups ranging from 2.56× to 4.47× by significantly reducing communication overhead. DT achieves a 4.47× speedup on the Reddit dataset, whereas on the Ogbn-paper dataset, the speedup is only 2.21×. This discrepancy is due to the embedding dimension in Reddit being significantly lower than the raw features, facilitating a reduction in communication volume through the early computation of NN operations. Lastly, IP provides speedups ranging from 1.1× to 1.5× by overlapping computation and communication. IP achieves an average speedup of 1.47× on the Reddit and Product datasets, while on the two larger datasets, the speedup averages only 1.11×. This is because large datasets require more chunks partitioned, leading to more frequent CPU-GPU communication.

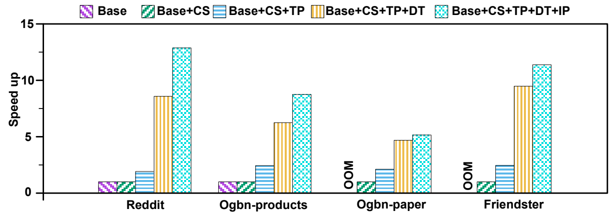


Figure 11: Performance gain analysis. "CS" indicates the chunk-based task scheduling, "TP" indicates the GNN tensor parallelism training, "DT" indicates the decoupled training method, and "IP" indicates the inter-chunk pipelining.

5.5 Scalability Analysis

Performance with varying cluster sizes. In this experiment, we compare NeutronTP with baselines when training GCN with different cluster sizes over two datasets. The results are shown in Figure 12. Across different cluster sizes, NeutronTP consistently outperforms the baselines. Specifically, as the cluster size increases from 2 to 16, NeutronTP achieves an average speedup of 6.33×, 5.97×, and 2.69× compared to DistDGL, NeutronStar, and Sancus, respectively. We observe that the execution time of NeutronTP, DistDGL, and NeutronStar decreases with an increase in the number of nodes. However, Sancus demonstrates poor scalability. That may be due to its communication implementation, where each worker needs to fetch the entire partition data from remote workers, even if only a small portion of the data is required. In contrast, NeutronTP adopts tensor parallelism to eliminate vertex dependencies and employs a decoupled training approach to reduce communication overhead, achieving nearly linear speedup. Specifically, as the number of nodes in the cluster increases from 2 to 16, NeutronTP achieves an average speedup of 6.33× and 4.97× on Reddit and Ogbn-products.

Performance with varying model layers. In this experiment, we compare NeutronTP with baselines when training GCN with different model layers over two datasets in a 16-node cluster. For the 2, 3, and 4-layer models, the DGL sampling strategies were set to (25,10), (25,15,10), and (25,20,15,10) respectively. The results are

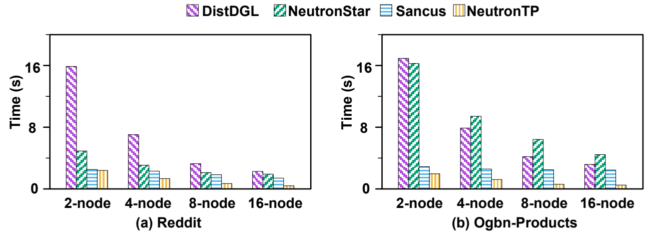


Figure 12: Per-epoch runtime of different systems with different cluster sizes.

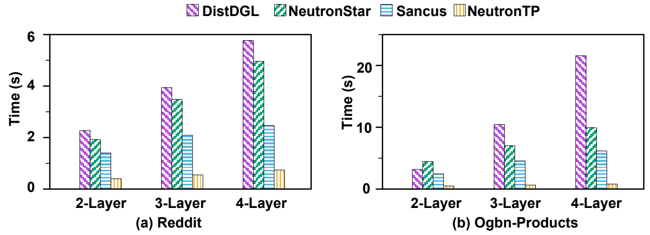


Figure 13: Per-epoch runtime of different systems with different model layers.

shown in Figure 13. We observe that the performance advantage of NeutronTP over other baselines gradually increased with the model depths. For the 2-layer model, NeutronTP achieves an average speedup of 5.99×. For the 3-layer and 4-layer models, the speedups were 8.65× and 11.13× respectively. This is because NeutronTP's tensor parallelism can effectively eliminate the cross-worker vertex dependencies among layers in GNN data parallel training, thereby removing substantial communication overhead. DistDGL experiences the most severe efficiency degradation because of the neighbor explosion problem [16], where the computation and memory requirements for mini-batch GNN training increase exponentially with the number of layers. NeutronTP outperforms DistDGL by up to 26.64× in the 4-layer model on the Ogbn-products dataset. This is because the average degree of Ogbn-products is much smaller than that of Reddit, making the graph topology sparser and causing a faster increase in the sampled subgraph size.

Performance with varying feature dimensions. In this experiment, we compare NeutronTP with baselines when training GCN with different input feature dimensions over two datasets in a 16-node cluster. The results are shown in Figure 14. We observe that the performance advantage of NeutronTP over other baselines gradually increased with the feature dimensions. For the 128-dimensional dataset, NeutronTP achieves an average speedup of 5,87×. For the 256-dimensional, 512-dimensional, and 1024-dimensional datasets, the speedups are 7.28×, 8.14×, and 12.74× respectively. When feature dimensions increase, GNN data parallelism suffers from significant communication overhead, particularly during the communication of raw features in the first layer. NeutronTP employs decoupled GNN tensor parallelism, which only gathers and splits vertex embeddings. Compared to GNN data parallelism, it reduces communication frequency and transforms features into lower-dimensional embeddings before communication.

5.6 GPU Utilization

We evaluate the GPU utilization during the training of GCN on Reddit for NeutronTP and baselines in a 16-node cluster. Figure 15 shows the results in a 20-second time window. The GPU utilization is recorded every 100 milliseconds and averaged in a 1-second interval. NeutronTP exhibits higher GPU utilization (62.85% on average) compared to DistDGL (19.91% on average), Sancus (37.67% on average), and NeutronStar (33.97% on average), with consistently higher peak GPU utilization for most of the time. DistDGL shows relatively low GPU utilization due to the sampling step involving a large amount of random access, which can be the bottleneck to limit GPU utilization. Additionally, baselines based on GNN data parallelism suffer from GPU idle time due to unbalanced workloads, resulting in decreased overall GPU utilization. NeutronTP experiences minimal GPU idle time, attributed to balanced workloads and a pipeline design between chunks, maximizing the overlap of communication and computation.

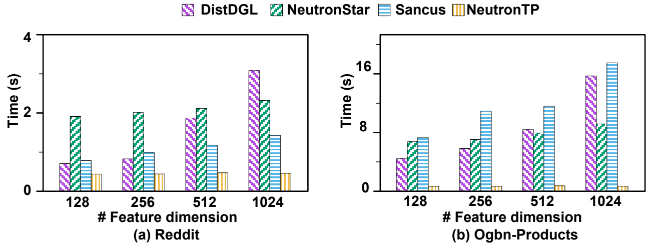


Figure 14: Per-epoch runtime of different systems with different feature dimensions.

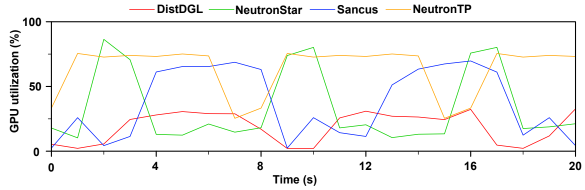
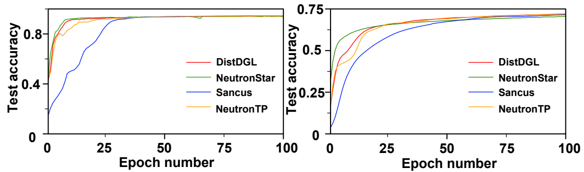


Figure 15: GPU utilization comparison. The average GPU utilizations are 62.85%, 19.91%, 33.97%, and 37.67% for NeutronTP, DistDGL, NeutronStar, and Sancus, respectively.

5.7 Accuracy Comparisons

The changes in the execution process introduced by the decoupled training method may impact model performance. We plot the epoch-to-accuracy curve on different systems for a GCN model over two datasets. The results are shown in Figure 16. After 100 epochs, the test accuracy reaches a stable state, NeutronTP and other baselines almost achieve the same test accuracy. However, NeutronTP converges slightly slower compared to the traditional GNN training method. Sancus exhibits the slowest increase in accuracy over epochs due to its use of historical embeddings. Additionally, it is worth noting that while many works [16, 25] have demonstrated that sampling strategies can lead to lower accuracy, we find that DistDGL performs close to full-graph training accuracy on commonly used datasets. We attribute this to the well-tuned parameters of DistDGL. Both mini-batch and full-graph training methods have their advantages. In summary, NeutronTP exhibits advantages when training deep GNNs or when the input graph includes a large proportion of training vertices. However, when the training set and the number of model layers are small, DistDGL still holds certain advantages.



(a) GCN on Reddit (b) GCN on Product

Figure 16: Epoch-to-accuracy.

5.8 Extension to heterogeneous graphs

NeutronTP can be naturally extended to heterogeneous graphs. We compare NeutronTP with DistDGLv2 [55] using two heterogeneous graphs [14, 15] and the R-GCN algorithm [31] in a 16-node cluster. DistDGLv2 extends DistDGL to support heterogeneous GNN training. R-GCN are designed to handle heterogeneous graphs, which consist of multiple types of edges. On the Ogbn-mag dataset, NeutronTP achieves a 6.15× speedup. On the Mag-lsc dataset, DistDGLv2 exhibits better performance as it trains on only 0.4% of the total vertices, reducing the computational load compared to NeutronTP. This is similar to the Ogbn-paper dataset, which includes only 1.1% training vertices, leading to significant differences in overall computation between full-graph and mini-batch training.

5.9 Training cost breakdown

We evaluate the training costs of different stages for both node classification and link prediction tasks using the GCN algorithm with the Reddit dataset. The results are presented in Table 4. We can observe that GNN computation is the primary cost, accounting for an average of 94% of the time in node classification and 79% in link prediction. By optimizing the GNN computation process, NeutronTP reduces end-to-end training time by 65% to 80% compared to NeutronStar. Therefore, the practical downstream tasks that benefit most from NeutronTP are those where GNN computation is a significant part of the overall training cost.

6 RELATED WORK

Full-graph GNN training. A set of GNN systems [5, 10, 16, 24, 29, 36, 39, 40, 42, 43] adopt full-graph GNN training to guarantee high accuracy. NeuGraph [24] defines a new, flexible SAGA-NN model to express GNNs. CAGNET [36] proposes 1.5D, 2D, and 3D graph partitioning to optimize the data distribution among GPUs. ROC [16] optimizes graph partitioning with online learning while managing memory with dynamic programming. DGCL [5] designs a communication planning algorithm that avoids conflict on various links. PipeGCN [39] reduces communication for boundary vertices by utilizing historical embeddings and effectively overlaps computation across different partitions. G3 [40] propose GNN hybrid parallelism to scale out GNN training with carefully scheduled peer-to-peer intermediate data sharing. Hongtu [42] designs a recomputation-cache-hybrid intermediate data management to significantly reduce the GPU memory requirement.

Load balancing study in distributed GNN training. Many survey papers [2, 21, 32] highlight workload imbalance as a primary challenge in distributed GNN training. Experimental studies [26, 50] have empirically demonstrated the prevalence of load imbalance. Many GNN systems attempt to address this issue by exploring graph partitioning strategies [17, 22, 36, 40, 53, 54]. SALIENT++ [17] extends the METIS partitioning approach with additional constraints to balance workloads while minimizing edge-cuts. PaGraph [22] and ByteGNN [53] employ streaming partitioning methods, selecting the optimal partition for each vertex individually. G3 [40] adopts an iterative partitioning approach, continuously exchanging vertices between different partitions. Given the complexity of real-world graph data, these methods often approximate load balance and may experience diminishing effectiveness with increasing graph power-law characteristics [26, 50]. Furthermore, the memory and computation overhead of these methods is considerable, especially for METIS and streaming partitioning, which may even surpass the training itself [6, 11, 50]. Therefore, we believe that partitioning features instead of graph data for GNN tensor parallelism is a promising distributed training approach.

Table 3: Comparison with DistDGLv2 on two heterogeneous graphs.

System	Runtime of R-GCN (s)	
	Ogbn-mag	Mag-lsc
DistDGLv2	36.3	**56.9**
NeutronTP	**5.9**	695.2

Table 4: The runtime breakdown (in seconds) with different tasks. (NC: node classification, LP: link prediction)

Task	System	Negative Sampling	GNN Computation	Classification Computation	Loss Calculation
NC	NeutronStar	-/-	**1.88/97%**	0.03/2%	0.01/1%
	NeutronTP	-/-	**0.36/90%**	0.03/7%	0.01/3%
LP	NeutronStar	0.07/3%	**2.12/90%**	0.11/5%	0.04/2%
	NeutronTP	0.07/9%	**0.53/67%**	0.15/19%	0.04/5%

Vertical feature partitioning. Some recent studies [6, 11] explore vertical feature partitioning in distributed GNN training. P3 [11] utilizes feature slices to complete the first graph aggregation operation, reducing feature fetching overhead. Du et al. [6] propose skipping feature fetching in some iterations, leveraging only partial feature dimensions for local training to achieve a trade-off between convergence error and feature communication time.

7 CONCLUSION

We present NeutronTP, a load-balanced and efficient distributed full-graph GNN training system. NeutronTP leverages GNN tensor parallelism for distributed training, which partitions feature rather than graph structures. Compared to GNN data parallelism, NeutronTP eliminates cross-worker vertex dependencies and achieves a balanced workload. To address the unique challenges of GNN tensor parallelism, NeutronTP employs a generalized decoupled training approach to significantly reduce communication overhead and a memory-efficient task scheduling strategy to reduce memory consumption while overlapping computation and communication. Extensive experiments demonstrate that our approach accelerates distributed GNN training significantly compared to GNN data parallelism while achieving comparable model accuracy.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (U2241212, 62072082, 62202088, 62072083, and 62372097), the 111 Project (B16009), and the Distinguished Youth Foundation of Liaoning Province (2024021148-JH3/501). Yanfeng Zhang and Qiange Wang are the corresponding authors.

REFERENCES

[1] Mark R. Baker and Rajendra B. Patil. 1998. Universal Approximation Theorem for Interval Neural Networks. Reliab. Comput. 4, 3 (1998), 235–239.

[2] Maciej Besta and Torsten Hoefler. 2024. Parallel and Distributed Graph Neural Networks: An In-Depth Concurrency Analysis. IEEE Trans. Pattern Anal. Mach. Intell. 46, 5 (2024), 2584–2606.

[3] Zhengda Bian, Qifan Xu, Boxiang Wang, and Yang You. 2021. Maximizing Parallelism in Distributed Training for Huge Neural Networks. CoRR abs/2105.14450 (2021).

[4] Aleksandar Bojchevski, Johannes Klicpera, Bryan Perozzi, Amol Kapoor, Martin Blais, Benedek Rózemberczki, Michal Lukasik, and Stephan Günnemann. 2020. Scaling Graph Neural Networks with Approximate PageRank. In KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020. ACM, 2464–2473.

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

[6] Bingqian Du, Jun Liu, Ziyue Luo, Chuan Wu, Qiankun Zhang, and Hai Jin. 2023. Expediting Distributed GNN Training with Feature-only Partition and Optimized Communication Planning. In IEEE INFOCOM 2024 - IEEE Conference on Computer Communications. IEEE, 1–10.

[7] Vijay Prakash Dwivedi and Xavier Bresson. 2020. A Generalization of Transformer Networks to Graphs. CoRR abs/2012.09699 (2020).

[8] Facebook. 2021. Gloo. https://github.com/facebookincubator/gloo.

[9] 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.

[10] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. CoRR abs/1903.02428 (2019). arXiv:1903.02428 http://arxiv.org/abs/1903.02428

[11] 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.

[12] 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, NeurIPS'17 Long Beach, CA, USA. 1024–1034.

[13] 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. ACM, 639–648.

[14] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. OGB-LSC: A Large-Scale Challenge for Machine Learning on Graphs. In Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual.

[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'20, December 6-12.

[16] 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 Machine Learning and Systems 2020, MLSys'20, Austin, TX, USA. mlsys.org.

[17] Tim Kaler, Alexandros-Stavros Iliopoulos, Philip Murzynowski, Tao B. Schardl, Charles E. Leiserson, and Jie Chen. 2023. Communication-Efficient Graph Neural Networks with Probabilistic Neighborhood Expansion Analysis and Caching. CoRR abs/2305.03152 (2023).

[18] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In 5th International Conference on Learning Representations, ICLR'17, Toulon, France, Conference Track Proceedings. OpenReview.net.

[19] 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. OpenReview.net.

[20] Jure Leskovec and Rok Sosic. 2016. SNAP: A General Purpose Network Analysis and Graph Mining Library. CoRR abs/1606.07550 (2016).

[21] Haiyang Lin, Mingyu Yan, Xiaochun Ye, Dongrui Fan, Shirui Pan, Wenguang Chen, and Yuan Xie. 2023. A Comprehensive Survey on Distributed Training of Graph Neural Networks. Proc. IEEE 111, 12 (2023), 1572–1606.

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

[23] 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. ACM, 338–348.

[24] 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, ATC'19, Renton, WA, USA. USENIX Association, 443–458.

[25] Vasimuddin Md, 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'21, St. Louis, Missouri, USA. ACM, 76.

[26] Nikolai Merkel, Daniel Stoll, Ruben Mayer, and Hans-Arno Jacobsen. 2023. An Experimental Comparison of Partitioning Strategies for Distributed Graph Neural Network Training. CoRR abs/2308.15602 (2023).

[27] Takato Nishijima. 2021. Universal Approximation Theorem for Neural Networks. CoRR abs/2102.10993 (2021).

[28] NVIDIA. 2021. Collective Communication Library (NCCL). https://developer.nvidia.com/nccl.

[29] 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.

[30] Emanuele Rossi, Fabrizio Frasca, Ben Chamberlain, Davide Eynard, Michael M. Bronstein, and Federico Monti. 2020. SIGN: Scalable Inception Graph Neural Networks. CoRR abs/2004.11198 (2020).

[31] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings.

[32] Yingxia Shao, Hongzheng Li, Xizhi Gu, Hongbo Yin, Yawen Li, Xupeng Miao, Wentao Zhang, Bin Cui, and Lei Chen. 2022. Distributed Graph Neural Network Training: A Survey. CoRR abs/2211.00216 (2022).

[33] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. CoRR abs/1909.08053 (2019).

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

[35] Shanshan Tang, Bo Li, and Haijun Yu. 2019. ChebNet: Efficient and Stable Constructions of Deep Neural Networks with Rectified Power Units using Chebyshev Approximations. CoRR abs/1911.05467 (2019).

[36] 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.

[37] 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'18, Vancouver, BC, Canada, Conference Track Proceedings. OpenReview.net.

[38] Cheng Wan, Youjie Li, Ang Li, Nam Sung Kim, and Yingyan Lin. 2022. BNS-GCN: Efficient Full-Graph Training of Graph Convolutional Networks with Partition-Parallelism and Random Boundary Node Sampling. In Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022. mlsys.org.

[39] 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. OpenReview.net.

[40] 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. https://doi.org/10.1145/3589288

[41] Boxiang Wang, Qifan Xu, Zhengda Bian, and Yang You. 2021. 2.5-dimensional distributed model training. CoRR abs/2105.14500 (2021).

[42] Qiange Wang, Yao Chen, Weng-Fai Wong, and Bingsheng He. 2023. HongTu: Scalable Full-Graph GNN Training on Multiple GPUs (via communication-optimized CPU data offloading). CoRR abs/2311.14898 (2023).

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

[44] Eric W Weisstein. 2003. Gershgorin circle theorem. https://mathworld. wolfram. com/ (2003).

[45] 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. PMLR, 6861–6871.

[46] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In 7th International Conference on Learning Representations, ICLR'2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net.

[47] Qifan Xu and Yang You. 2023. An Efficient 2D Method for Training Super-Large Deep Learning Models. In IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023, St. Petersburg, FL, USA, May 15-19, 2023. IEEE, 222–232.

[48] 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. AAAI Press, 7370–7377.

[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'18, London, UK. ACM, 974–983.

[50] Hao Yuan, Yajiong Liu, Yanfeng Zhang, Xin Ai, Qiange Wang, Chaoyi Chen, Yu Gu, and Ge Yu. 2023. Comprehensive Evaluation of GNN Training Systems: A Data Management Perspective. CoRR abs/2311.13279 (2023).

[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] Wentao Zhang, Ziqi Yin, Zeang Sheng, Yang Li, Wen Ouyang, Xiaosen Li, Yangyu Tao, Zhi Yang, and Bin Cui. 2022. Graph Attention Multi-Layer Perceptron. In KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022. ACM, 4560–4570.

[53] 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.

[54] 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. IEEE, 36–44.

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

[57] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. AI Open 1 (2020), 57–81.

[58] 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.