TOWARDS COMMUNICATION-EFFICIENT DISTRIBUTED TRAINING SYSTEMS

by

XINCHEN WAN

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy
in Computer Science and Engineering

February 2025, Hong Kong

TOWARDS COMMUNICATION-EFFICIENT DISTRIBUTED TRAINING SYSTEMS

by

XINCHEN WAN

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

ABSTRACT

As the scaling law persists consistently, distributed training has become the standard methodology to manage the exponential increase in model size and training data. Following this trend, distributed training systems are developed to handle the complexities and scale of distributed training and to embrace the computation powers of multiple devices. However, the communication remains one of the major challenges in these systems. The communication issues manifest in two key domains: the varied communication patterns across training paradigms during the model training stage, and the limited communication performance imposed by CPU-based processing during the data and model management stages.

This dissertation delineates my research efforts on building communication-efficient distributed training systems through paradigm-specific optimizations for the model training stage and hardware-accelerated optimizations for the data and model management stages. The resulting optimized systems are full-stack solutions tailored for hardware-accelerated Graph Neural Network (GNN) and Large Language Model (LLM) training.

For the sampling-based GNN training paradigm, we propose DGS, a communication-efficient graph sampling framework. Its key idea is to reduce network communication cost by sampling neighborhood information based on the locality of the neighbor nodes in the cluster, and sampling data at both node and feature levels. As a result, DGS strikes a balance between communication efficiency and model accuracy, and integrates seamlessly with distributed GNN training systems.

For the full-graph GNN training paradigm, we design G3, a scalable and efficient full-graph training system. G3 incorporates GNN hybrid parallelism to scale out full-graph training with meticulous peer-to-peer intermediate data sharing, and accelerates the training process by balancing workloads among workers through locality-aware iterative partitioning, and overlapping communication with computation through a multi-level pipeline scheduling algorithm. Although initially tailored for GNN training, we believe the fundamental principle of peer-to-peer sharing data in hybrid parallelism can be generalized to other training tasks.

For the LLM training paradigm, we introduce Hermod, a near-optimal coflow scheduler that orchestrates all coflow types at the cluster level during LLM training. Hermod employs the insight that coflows are characterized by three model factors, and designs 1) model factor-driven inter-coflow scheduling to align with the LLM training DAG, and 2) matching-based intra-coflow scheduling to maximize transmission rates for high-priority coflows. We believe our revisit of coflow scheduling in LLM training will inspire further LLM training optimizations.

To enhance communication performance in distributed services, we present LEO, a hardware-accelerated communication framework for distributed services. LEO offers 1) a communication path abstraction to describe various distributed services with predictable communication performance across edge accelerators, 2) unified APIs and wrappers to simplify programming experience with automatic communication configuration, and 3) a built-in multi-path communication optimization strategy to enhance communication efficiency. We believe LEO will serve as a stepping stone for the development of hardware-accelerated distributed services in distributed training systems.

Authorization

XINCHEN WAN

25 February 2025

TOWARDS COMMUNICATION-EFFICIENT DISTRIBUTED TRAINING SYSTEMS

by

XINCHEN WAN

This is to certify that I have examined the above Ph.D. thesis

and have found that it is complete and satisfactory in all respects,

and that any and all revisions required by

the thesis examination committee have been made.

Prof. Kai Chen, Thesis Supervisor

Prof. Xiaofang Zhou, Head of Department

Department of Computer Science and Engineering

25 February 2025

ACKNOWLEDGMENTS

Completing my Ph.D. degree has been one of the most memorable journeys in my life, and I am profoundly appreciative of the professional guidance, technical contributions, advice, and encouragement I have received from many people.

First and foremost, I would like to express my deepest gratitude to my supervisor, Prof. Kai Chen, for his continuous guidance and support throughout my Ph.D. study. He provided me with the freedom to explore research topics that intrigued me and connected me with experts in both academia and industry. When writing papers, he helped me revise the draft and offered valuable suggestions to enhance clarity. His insights, meticulousness, and commitment to top-notch research have deeply influenced and shaped my research pursuits and enduring values.

I would like to thank Prof. Christopher Kin Ying Leung, Prof. Song Guo, Prof. Qiong Luo, Prof. Wei Zhang, and Prof. Hong Xu for serving on my thesis defense committee. I also extend my gratitude to my thesis proposal and qualifying examination committee members: Prof. Wei Wang, Prof. Yangqiu Song, Prof. Binhang Yuan, Prof. Qifeng Chen, and Prof. Brahim Bensaou for their valuable feedbacks to improve my research work.

I have been very fortunate to collaborate with many outstanding researchers in academia. I am grateful to Prof. Xin Jin and Prof. Guyue Liu from Peking University and Prof. Yiming Zhang from Xiamen University for their invaluable help and guidance in my research.

The vibrant life at HKUST has greatly enriched my Ph.D. endeavor. I am grateful to my labmates in the HKUST iSING group, both current and former: Liu Yang, Xiaoyu Hu, Zilong Wang, Hao Wang, Di Chai, Yilun Jin, Jinbin Hu, Bowen Liu, Xudong Liao, Kaiqiang Xu, Zhenghang Ren, Wenxue Li, Decang Sun, Xinyang Huang, Feiyuan Zhang, Xinyu Yang, Yijun Sun, Pengbo Li, Yuxuan Li, Qijing Li, Yunxuan Zhang, Xiangzhou Liu, Wei Bai, Li Chen, Hong Zhang, Shuihai Hu, Gaoxiong Zeng, Junxue Zhang, Yiding Wang, Han Tian, Chaoliang Zeng, Weiyan Wang, Wenxin Li, Baochen Qiao, Jiacheng Xia, Qinghe Jing, Bairen Yi, Ge Chen, Zhaoxiong Yang, Jingrong Chen, Duowen Liu, Ding Tang, Zhuoyi Peng, Yiqing Ma, Cengguang Zhang, Shuowei Cai, Xiaodian Cheng, Mingxuan Fan, Yuxuan Qin, Meilu Zhu, and Lihuan Hui, etc. I am especially grateful to Chaoliang Zeng, Zilong Wang, Xudong Liao, Han Tian, Liu Yang, Xiaoyu Hu, and Hao Wang for their invaluable assistance in both my research and personal life. I also thank Qinghe Jing and Duowen Liu for their companionship during my early Ph.D. years, and Kaiqiang Xu, Jingrong Chen, and Xinyu Yang for their valuable research assistance. Many thanks to the seniors, Hong Zhang, Shuihai Hu, Bairen Yi, Junxue Zhang, and Jiacheng Xia for their guidance and help. I am also grateful to other friends at HKUST: Jiayang Li (who accompanied with me for years), Peng Xu, Xin Liu, Haoyang Li, Wei Dong, Jialun Cao, and Hang Xu etc. Finally, I extend my gratitude to the CSE staffs for their continuous support.

The internships at ByteDance and Huawei are invaluable to my Ph.D. journey. I would like to express my heartfelt gratitude to Dr. Layong Luo for his hands-on guidance in building AI interconnects. His rigorous research attitude deeply influences me. I am also grateful to Yimin Jiang for his generous help in clarifying training details. I appreciate the years of collaboration with Luyang Li and cherish the memories of research discussions and leisure activities. I extend my thanks to other colleagues at ByteDance: Chuanxiong Guo, Qingsong Ning, Feng Ning, Kui Ji, Yaodong Han, Bin Xu, Ke Cheng, Tianhao Wang, Qian Zhao, Shideng Zhang, Haoran Wei, Peng Xie, and Xin Wei, etc. I thank Zilong Wang, again, and Tong Wei as my roommates during the internship. At Huawei, I am grateful to Dong Lin, Shoushou Ren, Haibo Wang, and Junyi Zhang for their invaluable guidance.

Beyond the academic realm, I would like to express my appreciation for the constant support from my friends: Rui Zhang (who generously helped me when I was in Seattle), Depin Wang, Jinze Li, Runtong Zhang, Kaijie Tang, Junda Zhang, Qingyuan Wang, Yichuan Wang, Jiadong Lin, Yuyi Zhong, Bingjie Tang, Yuanhao Cheng, Tianyi Hu, and Guangrui Li, etc. I would not have been able to overcome the challenges and difficulties in my life without their emotional encouragement.

Last but not least, I would like to thank my family for their unwavering love and support throughout my journey. I would like to express my hearfelt gratitude to my grandparents, parents, elder brother, sister-in-law, and my little niece and nephew. Without them, none of my achievements would have been possible. I am deeply indebted to my fiancee Dongmin Shi for her love, understanding, support, encouragement, and sacrifices. I dedicate this dissertation to my beloved grandfather for his precious love and guidance. I miss him so much.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Recent years have witnessed an explosive use of deep neural networks (DNNs) in multiple application domains such as computer vision (CV) [68], natural language processing (NLP) [148], and graph tasks [66, 91, 149], etc. As the scaling law for DNNs [86] persists consistently, both the model and dataset sizes are stipulated to grow exponentially, resulting in distributed training as the *de-facto* methodology embraced to make the training time manageable, *e.g.*, from months to hours. Following this trend, distributed training systems are developed to handle the complexities and scale of distributed training for AI researchers. These systems leverage the power of multiple computing resources, such as GPUs, CPUs, and DPUs across several machines, to parallelize and accelerate the training tasks. There have been tremendous distributed training systems supported vividly by communities, ranging from general machine learning systems such as TensorFlow [30], PyTorch [102, 127], and MXNet [44], to specific distributed training libraries including Horovod [141], BytePS [80], DeepSpeed [137], DGL [158, 196], and Ray [121].

The conventional workflow of distributed training systems encompasses the following stages: 1) data loading for randomly shuffling data and loading them into batches in the systems; 2) model training, the core stage, for parallelizing forward and backward propagation, loss calculation, and parameter updates; 3) checkpointing which periodically stores the model parameters during training for the avoidance of interruptions during training; and 4) model exporting to export the trained model for production deployment.

Communication is involved throughout these stages and hence plays a pivotal role in distributed training systems. The communication processes include: the data synchronization across different nodes, gradient aggregation, and embedding synchronization during the model training stage, and the network scheduling across different hardware during data and model management stages, etc. Consequently, the effectiveness of distributed training systems is largely dependent on its ability to maximize the efficiency of communication [40, 80, 124].

1.1 Communication Issues in Distributed Training Systems

In model training stage. The communication patterns during the model training stage vary significantly across training paradigms, leading to diverse communication challenges. For instance, Graph Neural Network (GNN) training requires both graph-related operations for neighbor aggregation and neural network computations at each layer [78, 116], whereas large language model (LLM) training employs hybrid parallelisms, such as data, pipeline, tensor, sequence, and expert parallelisms [103]. This paradigm diversity results in distinct communication bottlenecks. In GNN training, embedding synchronization across workers poses a major challenge, as its volume scales exponentially with the number of GNN layers. Researches indicate that this synchronization can consume up to 90% of the total training time [40]. For LLM training, the simultaneous adoption of multiple parallelisms leads to overlapping operations at the cluster level. Profiling results from production environments [41] and research studies confirm that such overlaps lead to mismatches between cluster resources and the LLM training dependency graph, ultimately hindering cluster-wide performance [100, 130].

In data & model training stages. For data & model training stages in distributed training, they require efficient data access across multiple machines and robust data storage solutions [44, 101, 123], typically facilitated by classical distributed services such as key-value stores, distributed file systems, and elastic block storage in cloud environments. These storage solutions are designed to provide scalable and reliable access to data, which is essential for the smooth operation of distributed training workflows. However, traditional CPU-based processing introduces significant performance bottlenecks in throughput and latency. Recent studies [70] indicate that data loading and preprocessing account for up to 29.5% of total training time, while correctness tests contribute an additional 19.0%, leading to GPU inefficiencies where nearly half of the computation time is spent waiting for data. To address these inefficiencies, distributed training systems must incorporate more efficient communication mechanisms, such as hardware-accelerated solutions, to enhance data transfer efficiency and overall system performance [90, 119, 139, 154, 168].

Summary. In conclusion, the different communication issues above raises the following question: *how to build communication-efficient distributed training systems to address the* *varied communication issues encountered during end-to-end distributed training?* This dissertation answers this question affirmatively through paradigm-specific optimizations for model training stage, and hardware-accelerated optimizations for data and model management stages. By applying these targeted "apple-to-apple" optimizations to their respective stages, the distributed training systems with communication efficiency is achievable.

1.2 Contributions

We summarize the key contributions of this dissertation in building communication-efficient distributed training systems through paradigm-specific optimizations for the model training stage and hardware-accelerated optimizations for the data and model management stages, as shown in Figure 1.1.

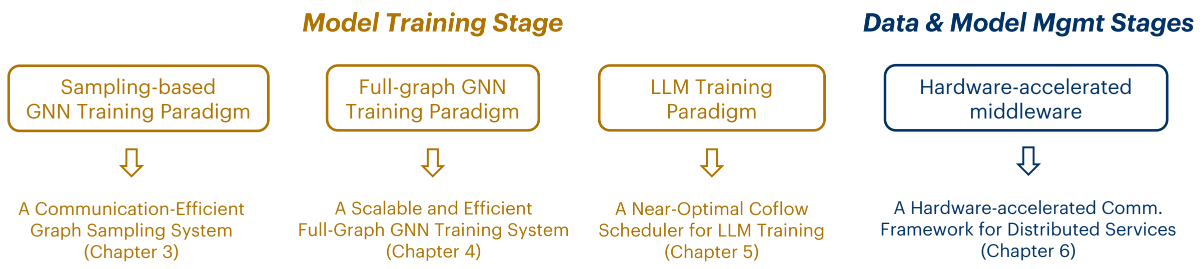

Model Training Stage
Data & Model Mgmt Stages
Sampling-based GNN Training Paradigm
Full-graph GNN Training Paradigm
LLM Training Paradigm
Hardware-accelerated middleware
A Communication-Efficient Graph Sampling System (Chapter 3)
A Scalable and Efficient Full-Graph GNN Training System (Chapter 4)
A Near-Optimal Coflow Scheduler for LLM Training (Chapter 5)
A Hardware-accelerated Comm. Framework for Distributed Services (Chapter 6)

Figure 1.1: The contributions of this thesis.

1.2.1 A Communication-Efficient Sampling System for Distributed GNN Training

Distributed GNN training tends to generate huge volumes of communication. To reduce communication cost, the state-of-the-art sampling-based techniques sample and retrieve only a subset of the nodes. However, our analysis shows that current sampling algorithms are still inefficient in network communication for distributed GNN training, which is mainly because of three problems: first, they overlook the locality of the sampled neighbor nodes in the cluster; second, they sample data only at the coarse-grained *graph node* level; and third, some mechanisms they adopted fall short in distributed scenarios.

In Chapter 3, we propose a graph sampling framework, DGS, for distributed GNN training, which effectively reduces network communication cost while preserving the final GNN model accuracy. To achieve this, DGS samples neighborhood information based on the locality of the neighbor nodes in the cluster, and samples data at the levels of not only graph nodes but also node features based on *explanation*. Specifically, DGS constructs an explanation graph which preserves the relationship between the local graph and remote nodes, and leverages the recently-proposed model explanation technique to design an online explanation scheme that interprets the importance of nodes and features.

We have implemented a fully functional DGS prototype, and evaluation results show that DGS achieves up to 1.25× throughput speedup over the state-of-the-art FastGCN and reduces the communication cost by up to 28.3%, while preserving the final model accuracy almost the same as that of full-batch training.

1.2.2 A Scalable and Efficient Full-Graph GNN Training System

Graph Neural Networks (GNNs) have emerged as powerful tools to capture structural information from graph-structured data, achieving state-of-the-art performance on applications such as recommendation, knowledge graph, and search. Graphs in these domains typically contain hundreds of millions of nodes and billions of edges. However, previous GNN systems demonstrate poor scalability because large and interleaved computation dependencies in GNN training cause significant overhead in current parallelization methods.

In Chapter 4, we design G3, a distributed system that can efficiently train GNNs over billion-edge graphs at scale. G3 introduces *GNN hybrid parallelism* which synthesizes three dimensions of parallelism to scale out GNN training by sharing intermediate results peer-to-peer in fine granularity, eliminating layer-wise barriers for global collective communication or neighbor replications as seen in prior works. G3 leverages *locality-aware iterative partitioning* and *multi-level pipeline scheduling* to exploit acceleration opportunities by distributing balanced workload among workers and overlapping computation with communication in both inter-layer and intra-layer training processes.

We show via a prototype implementation and comprehensive experiments that G3 can achieve as much as 2.24× speedup in a 16-node cluster, and better final accuracy over prior works.

1.2.3 A Near-Optimal Coflow Scheduler for LLM Training

Training large language models (LLMs) generates diverse coflows within a cluster, requiring optimized scheduling to enhance communication-computation overlap and minimize training time. Existing coflow schedulers are limited by endhost-only scheduling, and fail to address contentions across and within coflows, leading to suboptimal communication schedules and inefficient bandwidth utilization.

In Chapter 5, we present Hermod, a near-optimal coflow scheduler that orchestrates all coflow types at the cluster level during LLM training. The key insight behind Hermod is that coflows can be uniquely characterized by three model factors—microbatch ID, coflow type, and layer ID—enabling optimal scheduling decisions. Leveraging this insight, Hermod employs a two-layer strategy: (1) model-factor driven inter-coflow priority scheduling to align with the LLM training DAG, and (2) matching-based intra-coflow scheduling to maximize transmission rates within the highest-priority coflows.

Our fully functional prototype achieves significant job completion time reductions, with up to 1.44$\times$ and 2.07$\times$ speedups over state-of-the-art schedulers in testbed and simulation experiments, respectively.

1.2.4 A Hardware-Accelerated Communication Framework for Distributed Services

Distributed services increasingly employ high-performance edge accelerators to offload message-level computation and improve system performance. However, integrating hardware acceleration into these services poses significant communication challenges due to poor portability, complex configuration, and under-utilized resources issues.

In Chapter 6, we present LEO, a hardware-accelerated communication framework for distributed services. LEO facilitates portability across both application and hardware via introducing a *communication path* abstraction, which is capable of describing multiple applications with predictable communication performance across diverse hardware. Additionally, LEO exposes unified APIs and wrappers to ease the programming for developers and enable automatic communication configuration. Moreover, LEO incorporates a built-in multi-path communication optimization to enhance communication efficiency.

We have implemented a prototype of LEO and evaluated it with four case studies on an FPGA-based hardware platform. Our results show that LEO is both generic and efficient for multiple distributed services, yielding up to 2.62× speedup over baselines with negligible overhead.

1.3 Organization

The remainder of this thesis is organized as follows. Chapter 2 briefly introduce the background of distributed training systems and communication in these systems. Chapter 3 introduces DGS, where we present the detailed design, implementation, and evaluation for a communication-efficient sampling framework for distributed GNN training. Chapter 4 describes G3, where we propose a GNN hybrid parallelism for addressing the complex data dependency in GNN training, and accommodate the associated performance issues accordingly. Chapter 5 presents Hermod with near-optimal scheduling policies for both inter- and intra-coflow scheduling. Chapter 6 presents LEO, a hardware-accelerated communication framework for distributed services. Finally, we conclude the thesis and discuss future work in Chapter 7.

CHAPTER 2

BACKGROUND

In this chapter, we introduce the background of distributed training systems and hardware-accelerated distributed services. We first go through the basic concepts of distributed training system, including the basics of distributed training and workflow of the training systems (§2.1). Then we introduce the hardware-accelerated distributed services (§2.2).

2.1 Basic Concepts of Distributed Training System

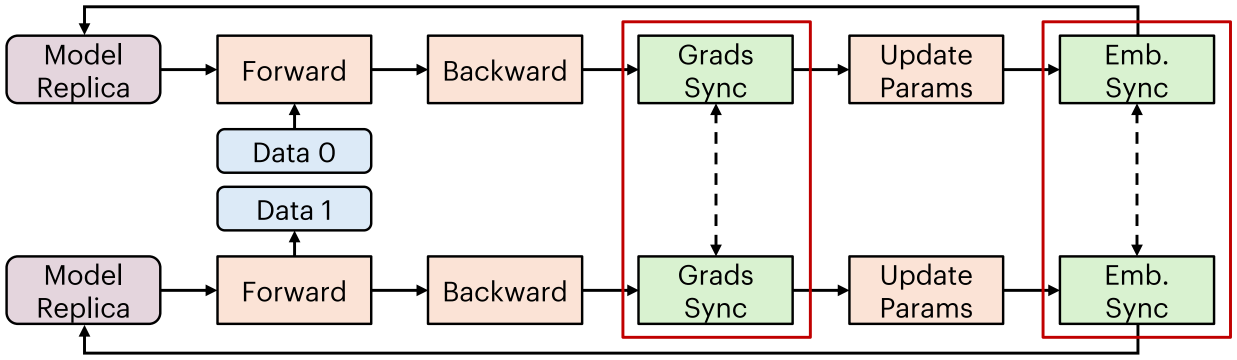


Figure 2.1: Conventional data parallel training. [81]

Distributed training. A conventional data parallel training schema is shown in Figure 2.1. Initially, each worker possesses a replica of the model and optimizer states and a subset of the training data. During training, the model replica in each worker executes the forward and backward propagation in parallel. After the gradients are computed, the workers synchronize and aggregate them in a layer-wise manner, and update the model upon completion of each iteration. The communication in training centers around the gradient synchronization and embedding synchronization. The former typically follows BSP manner for accuracy guarantee, facing challenges of communication stalls and dynamic bandwidth contention [124,160]. And the latter suffers from huge communication volume determined by the model type and size [123, 155].

Workflow of distributed training systems. The classical workflow of distributed training systems consists of the following crucial stages, with each stage requiring efficient distributed services:

- Data Loading: Once preprocessed, data is loaded into the distributed system in batches. This stage is critical for efficient memory management and ensuring that the computational resources are effectively utilized without bottlenecks, requiring distributed file system and elastic block storage for data access and distribution.
- Model Training: The core of the workflow, model training, involves forward propagation, loss calculation, backpropagation, and weight updates. In distributed training, this process is parallelized across multiple nodes, necessitating constant communication between them to synchronize model updates and gradients. To ensure efficient communication, distributed services such as key-value store and GPUDirectRDMA [9] are used to store and exchange model parameters and gradients.
- Checkpointing: Periodically, the state of the model (weights, parameters) is saved. This not only provides a means to resume training in case of interruptions but also allows for the evaluation of the model at different stages of training. Such checkpoints are stored in a distributed file system or elastic block storage for durability and accessibility.
- Model Exporting: After training, the model is exported into a format that can be deployed in production. This step may also involve additional optimizations to improve performance in real-world applications. The exported model is stored in a distributed file system or elastic block storage for durability and deployment.

To keep pace with the ever-increasing training speeds of GPUs, the services employed in the above stages are often hardware-accelerated inline in the datapath to facilitate low-latency and high-throughput communication, as depicted in Figure 2.2.

2.2 Hardware-accelerated Distributed Services

Given the rapid increasing computational capacity of GPUs, *e.g.*, from 14 TFLOPS of V100 to 2250 TFLOPs of B100 in terms of FP16 [8], the software-optimized distributed services

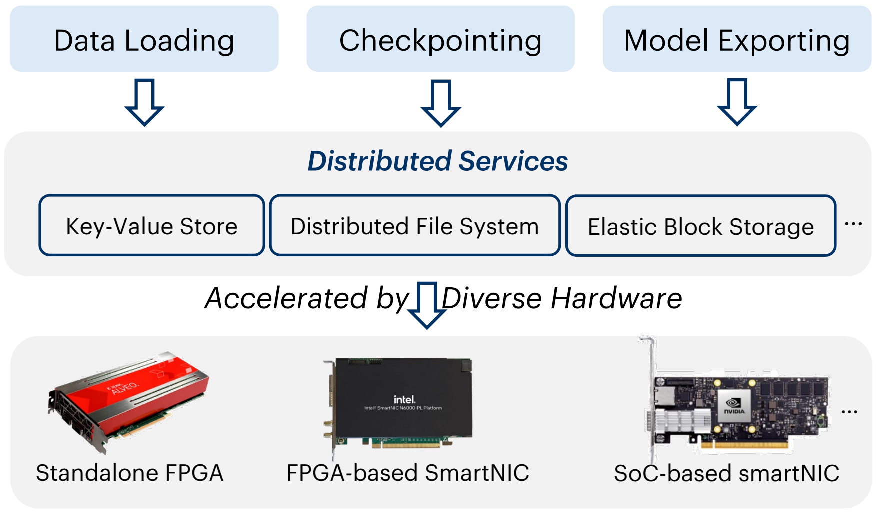


Figure 2.2: Hardware-accelerated distributed services.

often fail to consistently satisfy the performance demands of distributed training systems. Therefore, system developers deploy multiple types of domain-specific accelerators, *e.g.*, FPGA [42], SoC [111], and programmable switch [83] to explore the best performance opportunity for different distributed services. Next we discuss the exploration of two types of widely deployed hardware accelerators at endhost, *i.e.*, FPGA and SoC, in distributed services.

FPGA-accelerated systems. FPGA is a critical component in smartNIC that provides high throughput and flexibility for in-network processing. Specifically, Microsoft has widely deployed FPGAs in its datacenters [42] to accelerate the network stack in Azure cloud [59]. KV-Direct [97] improves key-value performance via leveraging programmable NIC to extend RDMA primitives to support remote key-value operations. [192] proposes to offload compaction operations to FPGAs for LSM-based key-value stores.

SoC-accelerated systems. While FPGA-based accelerator showcases its high throughput and energy efficiency in network processing, the complexity in terms of its programming raises the bar for system developers. SoC-based smartNICs, on the other hand, provides a more user-friendly programming interface and is widely deployed in multiple distributed services. Multiple systems have been proposed leveraging SoC-based smartNICs. Specifically, LineFS [90] decomposes DFS operations into execution stages to support parallel datapath execution pipelines on the smartNIC. Solar [119] accelerates EBS services by unifying packet processing and storage virtualization pipelines on DPUs. Moreover, iPipe [111], Floem [131], and Wei et al. [168] also propose solutions for accelerating key-value store and distributed file system.

CHAPTER 3

DGS: A COMMUNICATION-EFFICIENT GRAPH SAMPLING FRMEWORK

Graphs are pervasive in many real-world applications, such as social network analysis [72], molecular structure generation [182], and recommendation systems [69, 184]. Recent years have witnessed a surge of works that extend deep neural networks (DNNs) to extract structural information from graphs. These new methods, known as Graph Neural Networks (GNNs), have achieved satisfactory performance in various graph-related tasks including node classification, link prediction, and graph classification [66, 91, 149, 191].

GNNs express structural information by combining classical NN operations (*e.g.*, convolution and matrix multiplication) with iterative neighborhood aggregation. Figure 3.1 illustrates the computation process upon one node: a GNN layer (i) aggregates the embeddings of the node's one-hop neighbors, and (ii) transforms the aggregated embedding through NN operations to update the node's embedding. The two operations iterate across each of the n GNN layers, and the node's last-layer embedding contains information from all its n-hop neighbors.

Training GNNs in large graphs is challenging because the neighborhood aggregation procedure can involve a large number of multi-hop neighbor nodes, quickly exceeding the memory limitation of a single device. Therefore, large GNN training tasks are usually processed in a distributed manner, where the graph is first partitioned into subgraphs and then trained in parallel (each subgraph on one worker) with necessary communication. However, while the memory constraint is relieved, iterative neighborhood aggregation makes inter-worker communication a bottleneck for GNN training. As the number of neighbor nodes can be exponentially large when increasing the number of GNN layers, the total communication volume increases drastically and affects the training efficiency significantly.

To reduce communication overhead in distributed GNN training, researchers have proposed the sampling-based techniques [140], which sample and retrieve only a subset of the features and thus generate less communication than obtaining the whole set of features. However, our analysis (§3.1) finds that current sampling algorithms [43, 45, 66, 177,185,199] are still far from being communication-efficient for distributed GNN training, mainly because of the following problems.

First, existing sampling algorithms overlook the locality of the neighbor nodes in the cluster, because they are initially designed for low memory footprint in *mini-batch training* on a single worker but not for distributed scenarios. Simply migrating these algorithms to distributed GNN training overlooks the difference between accessing local and remote nodes, and thus leads to extra communication traffic.

Second, existing sampling algorithms sample data at the relatively coarse-grained *node* level to guarantee high training accuracy. However, our experiments (§3.1.4) show that for popular sampling algorithms like GraphSAGE random neighbor sampling [66], as many as 30% features can be reduced with no more than 0.5% accuracy loss.

Third, some mechanisms adopted in sampling algorithms can severely lower the performance of distributed training. For example, LADIES [199] requires layer-wise adaptive compute and update of global sampling probability during training. While the overhead of such update is tolerable in a single worker, it becomes overwhelming in distributed scenario because of the frequent and heavy synchronization of probability tensors among workers.

Based on the above analysis, in this paper we study the graph sampling problem for efficient distributed GNN training. The basic idea is to minimize the communication overhead of each GNN training iteration without affecting the accuracy. To this end, we propose *DGS*, a distributed graph sampling framework that samples neighborhood information at the levels of both graph nodes and node features while preserving the final GNN model accuracy. DGS first constructs an explanation graph which preserves the relationship between the local graph and remote nodes to facilitate the explanation process. It then designs an online explanation scheme that interprets the importance of nodes and features leveraging the state-of-the-art explanation technique [179] with minor system overhead.

We implement and evaluate DGS over 4 real-world datasets with popular GNN models [66,91,149]. Our results demonstrate that DGS can speed up distributed GNN training process by 1.25×-4.01×, while preserving the final model accuracy almost the same as that of full-batch training.

We summarize our contributions in this paper as follows:

- We analyze and identify the factors that determine the communication overhead in distributed GNN training, and illustrate the limitations of existing sampling algorithms when applied in a distributed scenario.
- We propose a novel sampling framework (DGS) which takes the locality of nodes in the cluster into consideration and applies sampling at both node and feature levels.
- We implement DGS and validate its effectiveness with extensive experiments. Experimental results demonstrate the effectiveness of DGS: it achieves up to 1.25× speedup over the state-of-the-art FastGCN and reduces the communication cost by up to 28.3%, while preserving the final accuracy almost the same as full-batch training.

3.1 Background and Motivation

3.1.1 Graph Neural Networks (GNNs)

GNNs emerge as a family of neural networks that perform *representation learning*: they take a graph as input and map each node into a d-dimensional vector, *a.k.a.*, an *embedding*. The embeddings are then used as inputs for downstream machine learning tasks, such as node classification, link prediction, and graph classification [66,91,149].

The computation process of GNN at one layer is illustrated in Figure 3.1. The GNN layer first aggregates the embeddings of the neighbor nodes calculated from the previous GNN layer, then applies classical NN operations such as matrix multiplication or convolution upon the aggregated result, and finally updates the embedding of that node. Formally, the neighborhood aggregation and NN operations are expressed as follows:

$$a_v^{(k)} = Aggregate^{(k)}(\{h_u^{(k-1)} \mid u \in \mathcal{N}(v)\}) \tag{3.1}$$

$$h_v^{(k)} = Update^{(k)}(h_v^{(k-1)}, a_v^{(k)}) \tag{3.2}$$

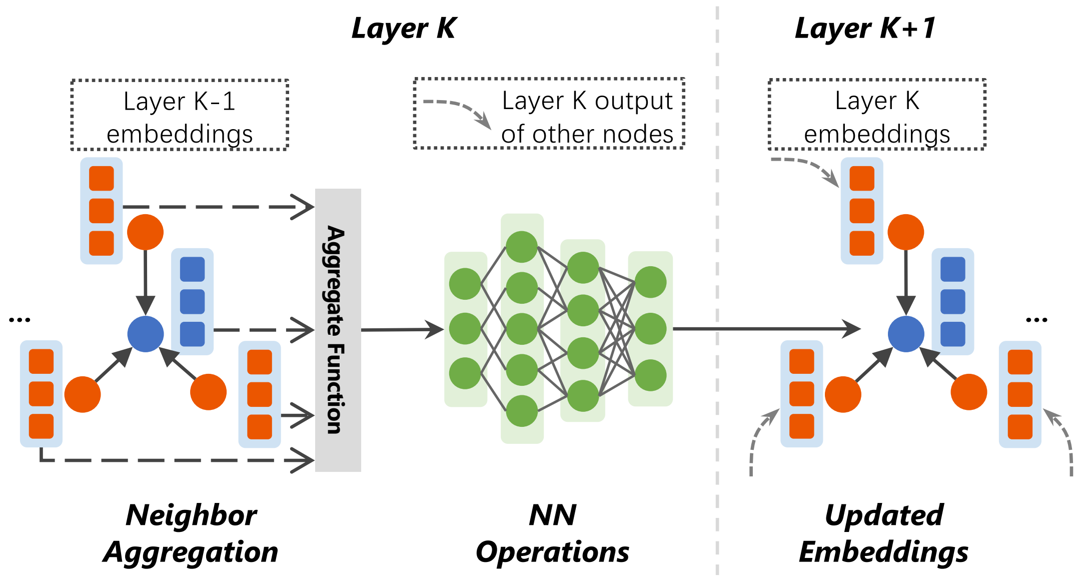


Figure 3.1: Computation process upon one node in GNN layer K by first aggregating its neighbors' embeddings, and then applying NN operations for its layer K output embedding.

where $h_v^{(k)}$ is the embedding of node v at the k-th GNN layer, $a_v^{(k)}$ denotes the activation output of the aggregated results, and $\mathcal{N}(v)$ represents the neighbors of v in the graph. For each node v at layer k, *Aggregate* first outputs $a_v^{(k)}$ by gathering the embeddings of its neighbors with an accumulation function, and then *Update* computes the node's new embedding from its previous embedding $h_v^{(k-1)}$ and aggregation result $a_v^{(k)}$. The above operations iterate from layer 1 to K.

3.1.2 Sampling Algorithms

Sampling algorithms are originally designed for mini-batch training to reduce the memory requirement [43, 66]. Mini-batch training allows for a small memory footprint during training and strikes a balance between small memory footprint and fast model convergence [147]. For GNNs, however, the iterative neighborhood aggregation tends to explode the GPU memory during training. To alleviate the memory issue, sampling algorithms are proposed to down-sample multi-hop neighborhood features of each mini-batch. For example, GraphSAGE [66] and VR-GCN [177] applies neighbor sampling upon each mini-batch. FastGCN [43], AS-GCN [74], and LADIES [199] adopts layer-wise sampling strategies which generate sampling decision layer-by-layer. ClusterGCN [45] and GraphSAINT [185] use subgraph sampling to extract subgraphs as training samples.

3.1.3 Distributed GNN Training

As graphs are becoming too large to fit into a single device, training tasks over them are usually processed in a distributed manner, *i.e.*, distributed GNN training (DGT).

Following the experience of distributed DNN training [80], DGT typically applies data parallelism among workers to parallelize the training process. In such paradigm, the input graph is partitioned into subgraphs with classical partition strategies [88], and the subgraphs are then sent to different workers. When the training process starts, each worker simultaneously trains GNN over its assigned graph partition and synchronizes model parameters at each iteration.

Due to the neighborhood aggregation introduced in §3.1.1, communication between workers not only contains information of conventional parameters/gradients, but also information of neighbor embeddings for aggregation. From the view of a worker in DGT, to compute all its nodes' embeddings at layer K, it requires all its neighbors' embeddings at layer K-1, which again requires their neighbors' embeddings at layer K-2 recursively until layer 1. The amount of data needed grows exponentially with the number of GNN layers, resulting in drastically increased communication volume and making DGT a network-intensive workload [60, 147].

Several techniques [60, 78, 116, 147, 164] have been proposed to alleviate the problem, among them the adoption of sampling in distributed scenario is a more promising one [140]. The idea of down-sampling neighbors at each subgraph helps decrease the communication volume across workers in DGT with minor model accuracy degradation [140, 196], thus effectively accelerating the global training process. As a matter of fact, sampling techniques have already been widely adopted in multiple distributed GNN systems [107, 196, 197].

Although the application of sampling to the distributed scenario seems natural, our analysis shows that current sampling algorithms are still far from communication-efficient for DGT.

3.1.4 Problems of Existing Sampling Algorithms

DGT Communication Cost Analysis. Given an input graph, DGT system divides it into n subgraphs and distributes them to workers for K-layer GNN training. The feature size at each node is $\mathcal{F}$, and $\mathcal{N}_{\mathcal{G}_i}^{j}$ denotes subgraph $\mathcal{G}_i$'s jth-hop neighbor nodeset. Following the iterative aggregation scheme in Equation 3.1,we formalize the communication cost of worker W_i as follows:

$$Comm(W_i) = (|\bigcup_{j=1}^{K} \mathcal{N}_{\mathcal{G}_i}^{j} - \mathcal{G}_i|) \times |\mathcal{F}| \tag{3.3}$$

Note that here we omit the communication cost caused by model synchronization because the network traffic generated by GNN models is trivial compared with the traffic generated by iterative neighborhood aggregation [196].

By summing up the communication cost of all workers, we obtain the total communication cost:

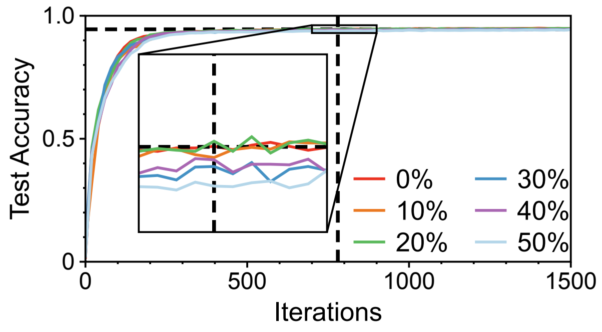
$$Comm_{total} = \sum_{i=1}^{n} Comm(W_i) = \sum_{i=1}^{n} n_{rmt}^{i} \times |\mathcal{F}| \tag{3.4}$$

where n_{rmt}^{i} denotes the size of $\mathcal{G}_i$'s K-hop neighborhood from remote workers.

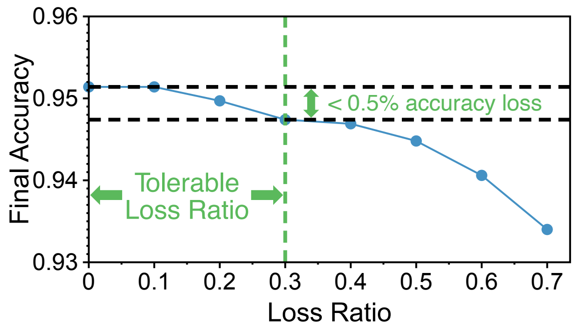
Based on Equation 3.4, we find out that the size of remote neighbor nodes and the size of features are the root cause of the high communication cost in DGT. Below we elaborate our three observations that prevent existing sampling algorithms from being communication-efficient.

Observation 1: *Existing sampling algorithms overlook the locality of the neighbor nodes in the cluster.* All sampling algorithms can be categorized as an optimization strategy that decreases the number of neighbor nodes at each worker for lower network traffic. These algorithms, however, consider all neighbor nodes the same but overlook the different overhead of accessing local and remote nodes. Such overlook may lead to extra communication overhead caused by accessing the remote nodes which contribute trivial to model convergence. Therefore, the locality of neighbor nodes in the cluster indicates a promising communication optimization opportunity.

Observation 2: *Existing algorithms only sample data at the coarse-grained node level rather than the fine-grained feature level.* Our second observation originates from the second factor in Equation 3.4, *i.e.*, the feature size $\mathcal{F}$. As all existing algorithms sample data at node-level, we consider the question of whether sampling data can be done at a finer granularity (*i.e.*, feature-level) with minor model accuracy compromise? As the features in many popular graph datasets [66,71] are pre-processed and expressed as sparse vectors, feature-level sampling may have no effect on the final accuracy. To verify this property, we run an experiment that trains a 2-layer GraphSAGE over Reddit [66], a popular graph dataset. We use the classical sample strategy of GraphSAGE, *i.e.*, 25 for the first layer and 10 for the second layer. Note that as the average node degree of Reddit is 984, our setting is an aggressive sampling strategy at node-level. During training, we randomly lose some values of the sampled nodes' features (*i.e.*, set those values to 0) with varying loss probabilities.



(a) Time-to-accuracy result.



(b) Final accuracy result.

Figure 3.2: Impact of random feature loss on model convergence: the 2-layer GraphSAGE converges to a tolerable final accuracy (no more than 0.5% accuracy loss) within the same iterations when the loss ratio is below 40%.

The result is shown in Figure 3.2, from which we can see that even after we apply an aggressive sampling strategy at node-level, as much as about 20% random loss of features has little impact on the model performance: the model can still achieve the same final accuracy within the same iterations. Besides, when the ratio is below 30%, the model can converge to approximately the same final accuracy with no more than 0.5% accuracy loss, within the same iterations. Therefore, the idea of sampling data at feature-level also indicates a promising optimization opportunity for DGT.

Note that in the above experiment we lose the features randomly without any intelligent selection. If we identify and select those features that may provide major contributions to model convergence, we can filter out more "unimportant" features from neighboring nodes to achieve much less communication traffic while still preserving the model accuracy.

Observation 3: *Some mechanisms adopted in existing algorithms may degrade the training performance.* Most sampling algorithms are designed by AI researchers with the assumption of running algorithms in a single host. However, when the scenario goes distributedly, some mechanisms may become a poor fit. For example, LADIES [199] requires layer-wise adaptive computation and update upon the global sampling probability during training. While the update overhead is trivial in a single worker, it becomes severe in distributed scenario due to the heavy synchronization of probability tensor among workers. Our experimental results (§3.4.1) also demonstrate that such overhead caused by synchronization can severely degrade the overall performance (up to 2.94× worse performance).

3.1.5 Opportunity and Challenges

Opportunity: The above three observations inspire us to design a new sampling algorithm for DGT which targets reducing the *less important* communication cost for each worker. That is, for a given subgraph, we need to identify a subset of remote nodes and features from its multi-hop neighborhood that contributes little to prediction, while keeping minimizing the synchronization frequency as much as possible in mind.

A recent study, GNNExplainer [179], is a natural fit for our problem. Its goal is to identify a subgraph G_s from a complete graph G_c and the associated features X_s from X_c that are important for the induced GNN's prediction $\hat{y}$. To achieve this, the model formulates the importance of node and feature by quantifying the change in the probability of prediction $\hat{y} = \Phi(G_c, X_c)$. This metric, known as mutual information (MI), is defined as:

$$max_{G_s} MI(Y, (G_s, X_s)) = H(Y) - H(Y|G = G_s, X = X_s) \tag{3.5}$$

where $H(Y)$ is the entropy term of the prediction. The above equation can be explained as: if masking one node from G_c strongly decreases the probability of prediction $\hat{y}$, MI

will become large, indicating that the removed node is important for the prediction. The feature mask can be explained in a similar way. The correctness of this model has been thoroughly verified in [179].

Though the explanation technique can help identify the important nodes and features for training, unfortunately, it is infeasible to run the explanation technique directly for DGT, mainly because of the following two challenges:

Challenge 1: *The original explanation is not designed for a subgraph.* We seek to find out a technique that can help us interpret the importance of remote nodes and sub-features upon a local subgraph, so that we can fetch the more important information and reduce the "unimportant" communication in DGT. However, the entity for which the explanation technique works is one single node in the graph or all nodes that have the same labels, rather than all nodes in the subgraph that are attached with different labels as expected. Directly using such technique is not feasible for our problem. A proper transformation of the subgraph is needed such that the transformed graph both preserves the original subgraph structure information and can be applied with explanation.

Challenge 2: *The original explanation training process is offline.* The original explanation technique operates offline: by inputting a trained GNN model and the prediction upon the node to be explained, the explanation process trains an explanation module iteratively and finally outputs the importance of the nodes and features. Clearly, DGT requires online explanation during training to guide its sample strategy. Further, even with online explanation the training process upon the explanation module may contend resources and hence affect the original GNN training performance. Therefore, it is necessary to parallelize the explanation process with GNN training while incurring minimal system overhead. Besides, the online explanation may require tensor synchronization among workers. As mentioned in observation 3, we should reduce such heavy synchronization as much as possible.

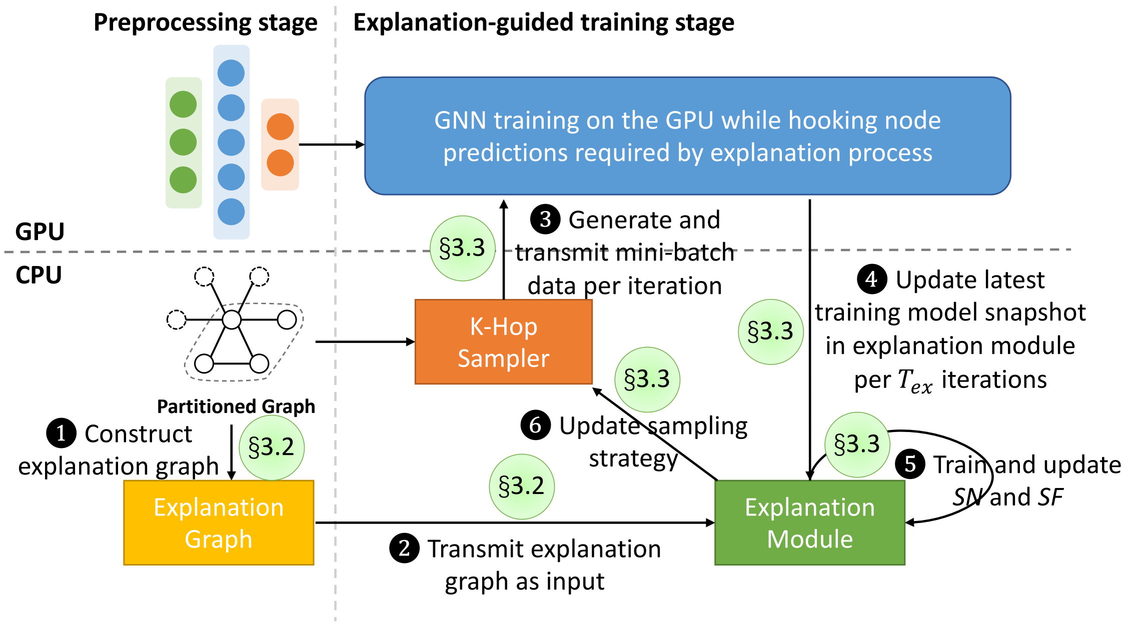

Preprocessing stage
Explanation-guided training stage
GNN training on the GPU while hooking node predictions required by explanation process
GPU
CPU
3 Generate and transmit mini-batch data per iteration
§3.3
K-Hop Sampler
4 Update latest training model snapshot in explanation module per T_{ex} iterations
§3.3
§3.3
Partitioned Graph
1 Construct explanation graph
§3.2
6 Update sampling strategy
§3.3
5 Train and update SN and SF
§3.2
Explanation Graph
Explanation Module
2 Transmit explanation graph as input

Figure 3.3: DGS overview. DGS covers two stages (preprocessing and explanation-guided training stages) and has six operations (❶-❻) for training.

3.2 Design

3.2.1 DGS Overview

DGS consists of two stages, namely preprocessing stage and explanation-guided training stage. Figure 3.3 shows an overview of DGS from the perspective of a worker.

(1) *Preprocessing stage*: When a job is submitted, a DGS worker manages to construct an explanation graph (❶) based on its assigned partitioned graph and generates the explanation module in CPU. The generated explanation graph is then transmitted into the explanation module for the latter online explanation (❷). Besides, the worker is also responsible for some conventional DGT preparations such as communication setup and model construction.

(2) *Explanation-guided training stage*: During training, the *k*-hop sampler initiates with a pre-defined neighbor sampling strategy (*e.g.*, random sampling [66] or importance sampling [43]). The sampler requests and gathers sampled node features. It then attaches them to the generated computation graph, and transmits all required minibatch data to GPU for GNN training in each iteration (❸). After every T_{ex} iterations, DGS updates the

GNN model duplicated in the explanation module using the latest snapshot of the training model, and infers the prediction of all nodes within the subgraph based on the model (❹). Note that we do not directly infer the predictions in GPU and then transfer out to CPU, as the size of GNN models is far smaller than the subgraph's node features, thereby reducing the transfer between CPU and GPU. Besides, the inference process in CPU is quick enough to satisfy our expectations. The explanation module in CPU then trains two masks, *i.e.*, sampled node mask (*SN*) and sampled feature mask (*SF*), given the node predictions and the loss function it predefined (in order to maximize MI between nodes) (❺). After the explanation module training is finished, *SN* and *SF* are transmitted to the k-hop sampler and interpreted to update sampling strategy (❻). The above operations repeat until the end of the training process.

3.2.2 Explanation Graph

As described in §3.1.5, current explanation technique [179] only supports explanation upon one node or a set of nodes that have the same labels, rather than all nodes in the subgraph as expected. Therefore, we should construct an explanation graph based on the given partitioned graph such that it can preserve the original graph structure information, especially the relationship between the local graph and remote nodes. We expect the explanation graph to have the following properties:

- Expressive graph structure: The explanation graph should be expressive to represent the relationship between local subgraph and remote nodes.
- Fast embedding interpretation: The operation for interpreting the predictions from the original graph toward the explanation graph should be fast. This is because such interpretation should be operated frequently (per T_{ex} iterations), hence a heavy interpretation may become a severe burden and degrade the performance.

We leverage graph pooling techniques [173, 181] and knowledge of graph partitioning [88] to achieve the above properties. Graph pooling is a popular technique used in GNN tasks, especially in graph classification tasks [181], to predict the label associated with the entire graph. It uses the generated node embeddings to encode the coarse-grained graph structure with a pre-defined differentiable pooling approach. As shown

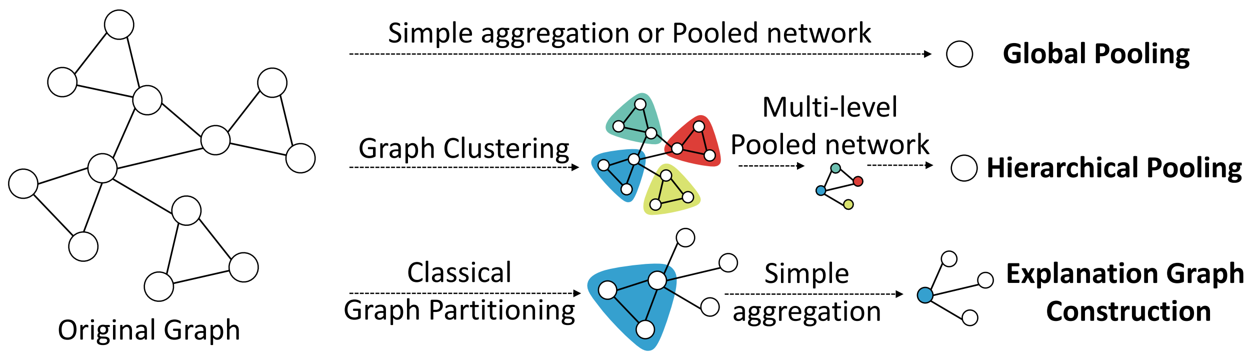

Original Graph
Simple aggregation or Pooled network
Global Pooling
Graph Clustering
Multi-level Pooled network
Hierarchical Pooling
Classical Graph Partitioning
Simple aggregation
Explanation Graph Construction

Figure 3.4: Graph pooling and explanation graph construction. Explanation graph construction is based on hierarchical pooling but leverages classical graph partitioning and simplified aggregation for efficiency.

in Figure 3.4, there are two conventional types of pooling manners, namely, (i) global pooling all the node embeddings together, *e.g.*, using a simple summation/average or neural network that operates over the whole graph; and (ii) hierarchical pooling node embeddings to a set of clusters, with each level training a dedicated pooling neural network.

The hierarchical approach usually demonstrates better accuracy improvement [181], but at the cost of complex cluster assignment and sophisticated neural network layer. Leveraging graph partitioning, we adopt hierarchical pooling in an efficient way. We utilize the principle of balanced edge-cut for graph partitioning to cluster nodes, and simply employ the average pooling approach to encode the subgraph with local nodes' embeddings. Since classical partition algorithms such as METIS [88] are already capable of capturing community structure from graphs, we explicitly utilize the partition result so as not to introduce extra clustering overhead. Specifically, we identify the node type (local nodes and remote nodes), shrink all local nodes (*i.e.*, the community recognized by partition algorithm) into one subgraph node v_G, and construct links between it and all remote nodes. Besides, we add a self-loop edge toward v_G to guarantee that the node can help explain itself during explanation. While the average pooling operation upon embeddings is simple, recent works [173] and our evaluation results (see §3.4.2) demonstrate that the approach is sufficient for our expectation.

3.2.3 Online Explanation

In this subsection, we introduce the online explanation which helps us process the offline explanation technique in an online mode. We first describe the module plug-in designed to integrate the explanation module in DGS framework, and then elaborate the online explanation process in detail.

Module Plug-in. We directly employ the explanation module presented in [179], and use the output masks, *i.e.*, node mask (*SN*) and feature mask (*SF*), to update our sampling strategy. As introduced in §3.1.5, these masks represent the importance of nodes and features upon the subgraph node v_G induced in the explanation graph. DGS interprets *SN* and *SF* into two probability tensors, $\mathcal{SN}$ (sampled nodes) and $\mathcal{SF}$ (sampled features), based on the node mapping between the original graph and explanation graph. The value of each tensor in one dimension ranges between 0 and 1 and represents the importance of a remote node or a feature dimension. Thereafter, DGS is able to identify the important remote nodes and features leveraging these tensors, and sample them with built-in GNN system APIs and user-defined sampling values. As it may incur bias because we only sample remote nodes but remain all local nodes for GNN computation, we add ϵ (empirically ranging from 0 to $1/10$ of the total sampled number) as a deviation towards the number of local sampled nodes and remote sampled nodes while remaining the number of total sampled nodes unchanged.

Online Explanation. The online explanation algorithm of DGS is described in Algorithm 1. During training, the main training process operates in GPU, and the explanation process operates in CPU to avoid GPU contention.

In the training process, the sampler first performs sampling to generate a computation graph of each mini-batch at the beginning of every iteration. Initially, it employs a predefined classical neighbor sampling strategy (line 5-6) which operates only in the first T_{ex} iterations, and switches to our distributed sampling if $\mathcal{SF}$ and $\mathcal{SN}$ are set by explanation process (line 7-8). During distributed sampling, it fetches top-N_f sub-features with respect to $\mathcal{SF}$, from the sampled nodeset regarding the probability tensor $\mathcal{SN}$ with deviation ϵ, and generates the sampled minibatch computation graph. DGS then transmits the computation graph from CPU to GPU to instantiate distributed GNN training process

Algorithm 1: Online explanation algorithm (a per-partition view)

```
Input:
partition id $i$, subgraph partition $\mathcal{G}_i$, explanation graph $\mathcal{E}_i$, number of GNN layer
$K$, feature dimension $f$, node feature $X_i$, node predictions $h_i$, node sampling size
$N_n$, node sampling deviation $\epsilon$, feature sampling rate $N_f$, sampling update period
$T_{ex}$, explanation epoch $\mathcal{T}_E$, loss threshold $\mathcal{L}$, explanation module $\mathcal{F}_E$, initial model
$w^0$
Output:
trained model $w^T$
1  begin
2      for $t = 1 ... T$ do
3          do in parallel
               // Training process
4              for each batch do
5                  if not set $\mathcal{SN}$ or $\mathcal{SF}$ then
6                      Use conventional sampling strategy;
7                  else
8                      Fetch sampled nodeset with top-$N_f$ sub-features according to
                       $\mathcal{SN}$, $\epsilon$, and $\mathcal{SF}$ from remote workers;
9                  Construct $K$-layer computation graph;
10                 Process GNN computation;                                  ▷ in GPU
11                 Process AllReduce to share gradients;
12                 Update $w^t$;
               // Explanation process
13             for every $T_{ex}$ batches do
14                 Update latest model $w^t$ as $w_E^0$;                          ▷ in CPU
15                 Inference node predictions $h_{G_i}$ and interpret to $h_{\mathcal{E}_i}$ in $\mathcal{E}_i$;
16                 Update $X_{\mathcal{E}_i}$ in $\mathcal{E}_i$;
17                 for $t_e = 1 ... \mathcal{T}_E$ do
18                     $loss = \mathcal{F}_E(\mathcal{E}_i, X_{\mathcal{E}_i}, h_{\mathcal{E}_i}, p_n, p_f, w_E^{t_e})$;
19                     if $loss \leq \mathcal{L}$ then
20                         break
21                     Update $\mathcal{F}_E, w_E^{t_e}$;
22                 Update $\mathcal{SN}$ and $\mathcal{SF}$;
```

(line 9-12). The above operations iterate over time.

In the explanation process, the explanation module pulls the latest model from GPU every T_{ex} iterations, and updates its internal GNN model to infer node predictions (line 14). It then interprets the predictions and features based on the embedding interpretation described in §3.2.2 (line 15-16). Note that the pulled model is the latest model that generates a minimal loss, indicating a more correct node and feature masks to be explained. To avoid GPU contention with the main training process, the explanation module $\mathcal{F}_E$ is trained in CPU and generates explanation loss epoch by epoch (line 17-21). If the loss is below a pre-defined threshold $\mathcal{L}$, the explanation process terminates and updates $\mathcal{SN}$ and $\mathcal{SF}$ to guide future distributed sampling (line 22). Otherwise, DGS updates $\mathcal{F}_E$ and $w_E^{t_e}$, and loops the explanation training process until the end of $\mathcal{T}_E$ epochs. Eventually, the explanation process terminates when the training process finishes.

3.3 Implementation

We implement DGS on top of DGL [196], a popular open-source framework for DGT. We use DGL as a distributed framework for inter-worker message exchange and a graph propagation engine for graph-related operations, and use PyTorch [127] for neural network execution and model synchronization.

We extend DGL in multiple ways to support DGS design. First, we adopt DGL's probability-based sampling API and reuse its k-hop graph sampling service to achieve our sampling framework. We assign an appropriate tensor, *i.e.*, the tensor interpreted from $\mathcal{SN}$ (§3.2.3) to the service such that we can interpret the explanation toward DGS sampling strategy, which can then be executed in all corresponding workers.

Second, we add the feature selection function in DGL to support fine-grained feature sampling. We extend DGL's `KVStore` push/pull handler to support extracting the subtensors given the dimension and selected index in the pull requests. With this extension, the extracted subtensors can be replied via RPC library. We also extend the feature selection function in DGL's functional components, *e.g.*, `DistGraph` and `DistTensor`, for compatibility.

For system optimization, we use `sysv_ipc` [2] (a high-performance IPC library) to exchange the updated model and masks between processes. To minimize the IPC overhead, we use shared memory among processes to avoid extra memory copy. We pre-register shared memory to be used for training, and load/store updated objects when necessary during training.

3.4 Evaluation

We evaluate DGS over several real-world graphs and compare it with state-of-the-art sampling algorithms. Overall, our results show that:

- DGS accelerates the training process by 1.25×-4.01×, and reduces the communication cost by up to 28.3%. The benefits DGS brings up increase with cluster size.
- DGS achieves almost the same final accuracy as full-batch training and without accuracy loss.
- DGS demonstrates its ability to explore complex GNN models against other sampling algorithms in DGT.

Experimental Setup: We evaluate DGS upon a GPU cluster with 4 physical servers, each equipped with 2 Tesla V100, 40 CPU cores (2.4GHz Intel Xeon Gold 5115), 128 GB RAM, and 2 Mellanox ConnectX5 NICs. The servers are interconnected via 4 Mellanox SN2100 switches running Onyx 3.7.1134 operating system. We virtualize an 8-node cluster by separating two docker containers at each server, where each container is equipped with one dedicated GPU, 20 CPU cores, 64GB RAM, and 10Gbps virtual Ethernet interface[1]. All servers run 64-bit Ubuntu 18.04 with CUDA library v11.0, DGL v0.6.1, and PyTorch v1.10.1.

Baseline: We choose several state-of-the-art sampling algorithms for comparison. Random sampling [66] performs random selection upon neighbors, which is the default sampling algorithm adopted in DGL. FastGCN [43] interprets graph convolutions as integral transforms and uses Monte Carlo approaches for importance sampling. LADIES [199] adopts a layer-wise sampling algorithm which considers the previously sampled nodes for calculating layer-dependent sampling probability. ClusterGCN [45] exploits the graph clustering structure and allows each worker to independently train GNN model upon its assigned subgraph.

[1]We use SR-IOV to separate the resource of physical NIC. [54] shows that it achieves nearly the same performance as the non-virtualized environments.

	Reddit	**Ogbn-products**	**Amazon**	**Ogbn-papers**
Nodes	232.9K	2.45M	1.60M	111.1M
Edges	114.6M	61.86M	132.2M	1.616B
Features	602	100	200	128
Classes	41	47	107	172
Avg. Deg	984.0	50.5	82.7	29.1

Table 3.1: Graph datasets used in our evaluation.

Note that except Random, all other works are open-sourced for single machine only[2][3][4]. Therefore, we modify the three algorithms to process distributedly based on the example code of DGL[5]. Such modification only helps process the algorithm in a distributed manner and has no effect on the final accuracy. As ClusterGCN does not require feature exchange between workers and hence always achieves linear throughput, we only compare with it in terms of accuracy (§3.4.2).

Datasets: Table 3.1 lists four real-world graph datasets that we used in our evaluation, including Reddit [66], a dense online discussion forum dataset, Ogbn-products [71], an undirected product co-purchase graph, Amazon [185], a multi-label amazon co-purchasing network, and Ogbn-papers [71], a billion-edge directed citation graph. We run multi-class classification tasks on Reddit, Ogbn-products, and Ogbn-papers, and binary classification tasks on Amazon.

Models & Metrics: We evaluate DGS with three representative models: GCN (Graph Convolutional Network) [91], GraphSAGE [66], and GAT (Graph Attention Network) [149]. We use the default model hyper-parameters as [66], *i.e.*, 2 layers and 16 hidden dimension per layer.

Parameters Setup. We set the batch size to 1024 and the sampling strategy with fanout {64, 64} (adopted in [199]) in all experiments. For DGS-related parameters, we set the node

[2]`https://github.com/acbull/LADIES/blob/master/pytorch_ladies.py#L95`

[3]`https://github.com/acbull/LADIES/blob/master/pytorch_ladies.py#L127`

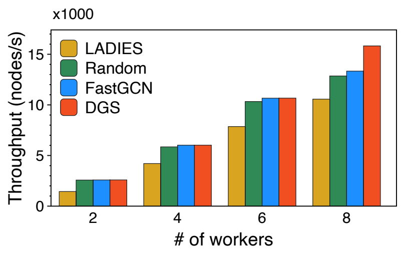
[4]`https://github.com/dmlc/dgl/tree/master/examples/pytorch/cluster_gcn`

[5]`https://github.com/dmlc/dgl/blob/master/examples/pytorch/graphsage`

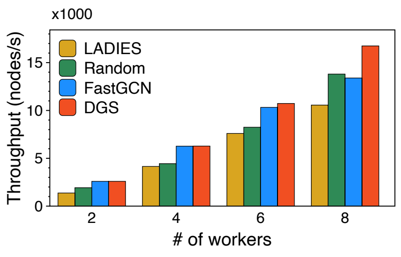
sampling size N_n at each layer the same to {64, 64}, the node sampling deviation ϵ to 0.1, the feature sampling rate N_f to 0.85, the sampling update period T_{ex} to 30, the explanation epoch $\mathcal{T}_E$ to 50, and the loss threshold $\mathcal{L}$ to 0.01 by default. The above settings ensure that the explanation module converges in our experiments.

3.4.1 Overall Performance

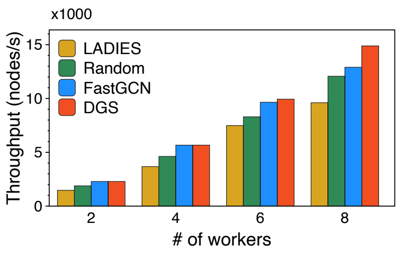
Due to the limited memory in each node (64 GB), we are not able to run large graphs like Ogbn-papers in small-scale clusters. Hence, we first present the scalability of each sampling algorithm over other 3 datasets. Then, we show the training performance over Ogbn-papers in an 8-node cluster. Besides, we also show the communication cost per epoch.



(a) GraphSAGE

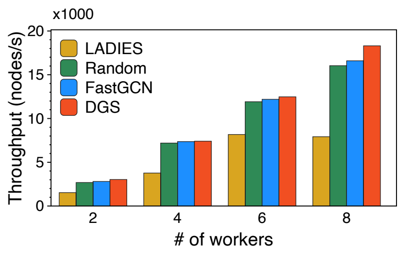


(b) GCN

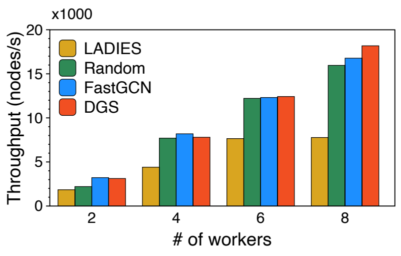


(c) GAT

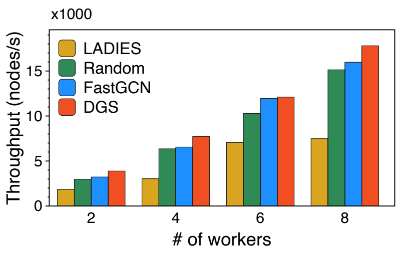
Figure 3.5: Throughput comparison with existing sampling-based methods in 3 GNN models over Reddit.



(a) GraphSAGE



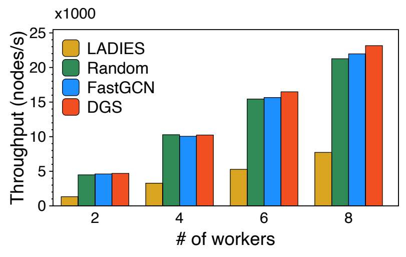
(b) GCN



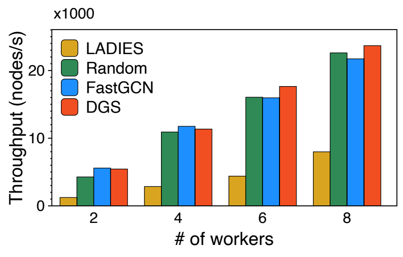
(c) GAT

Figure 3.6: Throughput comparison with existing sampling-based methods in 3 GNN models over Ogbn-products.

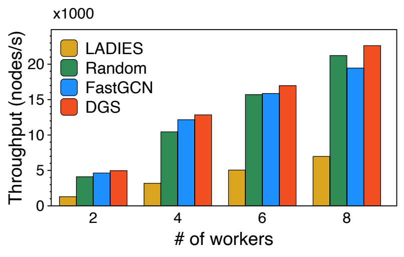
Scalability. The scalability results are shown in Figure 3.5, Figure 3.6, and Figure 3.7. We find that DGS outperforms all baselines in all settings with the speedup of 1.10×-3.24×, and the benefit increases with the cluster size. In general, LADIES performs worst among



(a) GraphSAGE



(b) GCN



(c) GAT

Figure 3.7: Throughput comparison with existing sampling-based methods in 3 GNN models over Amazon.

the three baselines due to the frequent probability synchronization of LADIES, as stated in §3.1.4. Specifically, LADIES performs as much as 3.24× worse than DGS over Amazon, as the number of synchronizations increases with large graphs. Random and FastGCN do not perform such synchronization during training: Random just samples nodes uniformly at random; FastGCN pre-computes and synchronizes the probability tensor before training, and then remains the value unchanged throughout training. Hence, the two methods perform better and achieve comparable performance in all experiments. However, we still see that they perform up to 1.25× worse performance than DGS because of their communication-inefficiency: without the locality awareness of neighbor nodes and adopting coarse-grained node-level sampling only, the two methods bring up extra and redundant communication, which degrades the global training throughput. For DGS, though it needs to synchronize the probability tensor when necessary, its update period is much coarser: one-time update every $T_{ex} = 30$ iterations versus the layer-size× updates per iteration in LADIES. In evaluating performance improvements across different GNN models, we observe that DGS yields higher efficiency in GraphSAGE and GCN than in GAT. This is because: 1) GAT model is more computation-intensive; 2) the improvement of communication is marginal when in small clusters, leaving limited opportunities for DGS to optimize training. Overall, with the communication-efficient mechanisms adopted in DGS, specifically, both node- and feature-sampling, and the coarse-grained synchronization mechanisms, DGS gains more benefit for training than all other algorithms. We expect that DGS can benefit the training process with more workers involved in.

Ogbn-papers. The training performance is shown in Table 3.2. Overall, DGS achieves 1.07×-4.02× speedup in 3 GNN models over Ogbn-papers. We find that DGS outperforms

	LADIES	Random	FastGCN	DGS	Speedup
GraphSAGE	10.43	31.42	36.72	41.57	1.13-3.99×
GCN	10.46	30.71	39.19	42.00	1.07-4.02×
GAT	9.580	28.38	34.76	38.09	1.09-3.98×

Table 3.2: Overall throughput (Knodes/s) when training over Ogbn-papers.

Method	Reddit	Ogbn-products	Amazon	Ogbn-papers
Sampling-based	11.030	15.168	24.896	8.302
NS-only	9.312	12.962	20.390	7.057
FS-only	10.026	14.119	22.194	7.809
DGS	**8.377**	**11.445**	**17.843**	**6.236**

Table 3.3: Communication cost (GB) per epoch. NS-only represents DGS adopts node-sampling only. FS-only represents DGS adopts feature-sampling only. Note that the communication cost of Ogbn-papers is small because in fact only a subset of Ogbn-papers is involved in training.

all baselines with the speedup of 1.07×-4.02×. DGS reduces more redundant communication costs than other algorithms with the help of the communication-efficient mechanisms. We expect that DGS can benefit training over even larger graphs.

Communication Cost. We also present the communication cost during training. We log the counter of the NIC to obtain the communication cost/epoch result when training GraphSAGE over three datasets in an 8-node cluster. We also deep-dive the node-sampling (NS-only) and feature-sampling (FS-only) mechanisms.

The results are shown in Table 3.3. DGS reduces the communication cost by 24.05%, 24.55%, 28.33%, and 24.89% for Reddit, Ogbn-products, Amazon, and Ogbn-papers, respectively, which demonstrates the communication efficiency of DGS. We also show the communication cost with NS-only and FS-only. As expected, the result shows that node-sampling is more useful for communication reduction, as node-sampling may eliminate the whole node feature transmission while feature-sampling only reduces a subset (15%) of it.

Method	Reddit				Ogbn-products				Amazon				Ogbn-papers
Full-Batch	**95.17**				**70.08**				**62.62**				**43.64**
Random	93.97				69.07				61.16				41.24
Random+RF	93.69				68.02				58.06				39.41
FastGCN	95.00				68.30				61.28				41.41
FastGCN+RF	94.91				67.73				58.76				39.04
LADIES	95.05				69.37				62.28				41.62
LADIES+RF	94.70				68.77				59.24				40.86
ClusterGCN	92.49				67.86				62.35				26.19
ClusterGCN+RF	92.12				67.48				57.37				24.72
DGS													
# of workers	2	4	6	8	2	4	6	8	2	4	6	8	8
$T_{ex} = 30$	94.57	94.80	95.02	95.03	69.88	69.23	69.51	**69.77**	62.36	62.33	62.38	62.35	**41.86**
$T_{ex} = 60$	94.36	95.09	**95.09**	95.05	69.01	69.02	69.03	69.02	62.41	62.43	62.42	**62.44**	40.72
$T_{ex} = 90$	94.80	95.06	95.00	94.95	68.98	68.95	69.03	68.98	62.30	62.28	62.20	62.34	40.77

Table 3.4: Comparison of test accuracy (%) on 4 datasets, where DGS under different numbers of partitions are shown. **The accuracies of full-batch and the highest accuracy of DGS in terms of # of workers are in bold.**

3.4.2 Final Accuracy

Next we present the final accuracy. We compare DGS with baselines, including ClusterGCN, in terms of final accuracy when training GraphSAGE over 4 datasets. We add each baseline with a random feature sampling rate 0.85, noted as +RF, to show the performance of our feature-level sampling. For DGS, we show the results by varying the number of partitions[6] and T_{ex} as the two parameters may affect the final accuracy in DGT. We also add the full-batch training (mini-batch without sampling) accuracies as our target accuracies.

The results are shown in Table 3.4. The full-batch accuracies are: 95.17% in Reddit, 70.08% in Ogbn-products [7], 62.62% in Amazon, and 43.64% in Ogbn-papers. As expected, LADIES achieves the highest accuracies among baselines in almost all datasets due to its adaptive layer-wise importance sampling. FastGCN has lower accuracies than LADIES. ClusterGCN behaves differently in different datasets: in Amazon it achieves as much as 62.35% comparable accuracy as LADIES, but 92.49% and 26.19% low accuracy in Reddit and Ogbn-papers. It is because ClusterGCN depends heavily on the graph structure and the partition strategy may not be optimal. The accuracies of Random are the worst as it

[6]For Ogbn-papers, we only train it over 8-node due to memory constraint.

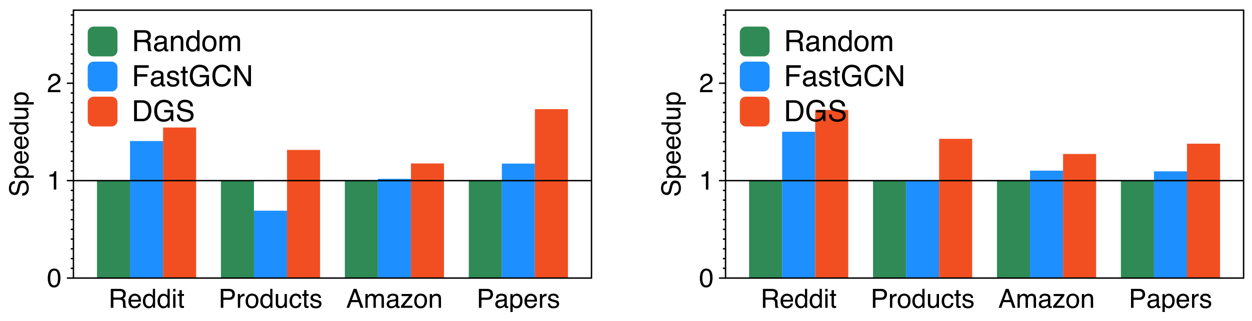
[7]The accuracy is different from the result reported in [1] because we use 2-layer GraphSAGE and they use 3-layer.

does not recognize the importance of each node. Besides, all baselines with RF face accuracy loss, because the random feature sampling cannot identify the important features during training and hence may sample the important ones and degrade the accuracy. DGS achieves similar accuracies as full-batch. As the table shows, DGS has no preference regarding the number of workers. This is because DGS adopts METIS partition which shows great ability in capturing graph community information regardless of the number of partitions. Besides, we find that DGS achieves the highest accuracy in most cases when T_{ex} is moderately 60, rather than what we expected $T_{ex} = 30$. We assume it is because the frequency of strategy may also affect the final accuracy, and a frequent update may adversely prevent the model from converging. Moreover, with infrequent updates upon sampling ($T_{ex} = 90$), DGS achieves lower accuracy compared to others, but still outperforms most baselines.

3.4.3 Impact of GNN Parameters

Next we present the effect of several GNN parameters, *i.e.*, partitioning strategy, feature size, and number of layers, to demonstrate the ability of DGS to explore complex GNN models in DGT. By default, we conduct all experiments when training GraphSAGE over Reddit in an 8-node cluster.

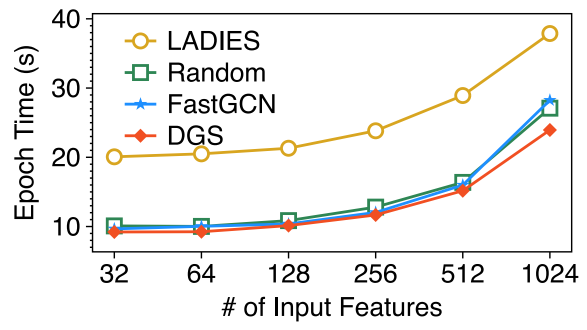
We first evaluate the effect of partitioning strategy. We adopt chunk-based strategy [198] and random strategy over 4 datasets and compare DGS with baselines. Note that we omit LADIES here because it performs far worse than others.



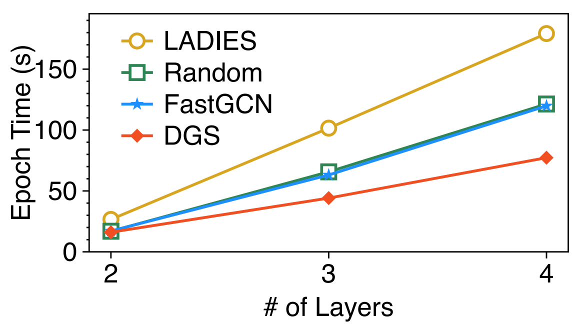
(a) Chunk-based partition (b) Random partition

Figure 3.8: Impact of Partitioning Strategy.

Impact of Partitioning Strategy. Figure 3.8 shows the throughput results (normalized by the throughputs of Random sampling). We see that DGS always achieves the best among all, with the average speedup of 1.44× and 1.45× in chunk-based and random settings, respectively. The results demonstrate that DGS is able to accelerate DGT via reducing communication, regardless of the choice of partition strategies.



(a) DGS's benefit increases as feature size increases.



(b) DGS's benefit increases as the number of layer increases.

Figure 3.9: Impact of feature size and number of layers.

Impact of Input Features. Next we evaluate the effect of input feature size. We vary the input features of Reddit from 32 to 1024-dimension and compare DGS with baselines. We use *epoch time* as the metric.

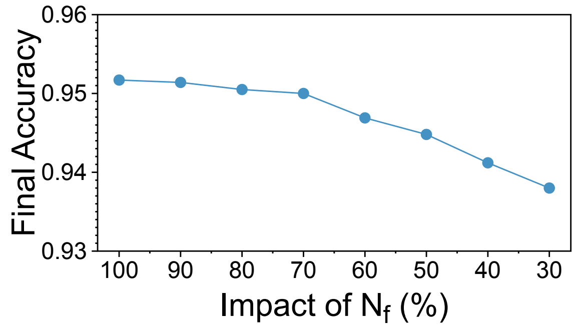
Figure 3.9a shows the throughput results. We see that DGS outperforms baselines in all settings, with the speedup ranging from 2.2% to 54.89%. Especially when feature size reaches 1024, DGS achieves 11.6%-51.03%× throughput speedup. We remind the readers that the communication cost is proportional to the size of input features, hence the communication ratio increases and shifts the training bottleneck from computation to communication. Without the awareness of nodes' locality in Random, FastGCN, and LADIES, the extra communication introduced by "unimportant" nodes from remote workers can result in significant communication overhead. DGS, however, takes into account the locality of nodes and reduces communication leveraging both node- and feature-level sampling.

Impact of Layers. We also evaluate the effect of the number of GNN layers. We create variants of GraphSAGE with 2, 3, and 4 layers and reuse the sampling strategy at each layer, *i.e.*, sample 64 nodes per layer. We use epoch time as the metric.

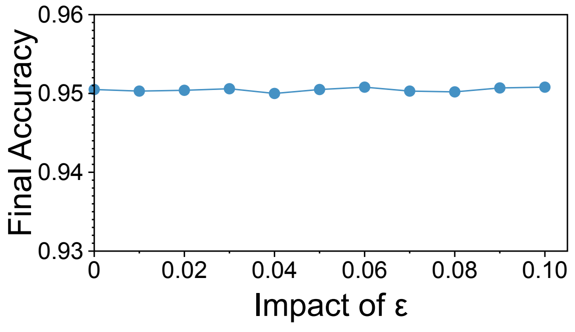
Figure 3.9b shows the throughput results. Compared to the increase of input features, the increase of layers affects GNN training more severely. It is because the increase of layers affects not only the communication volume among workers for more hop neighbor nodes' features, but also the computation workload of GNN, which ultimately limit the exploration for more complex but powerful GNN models. While DGS only optimizes the communication part, it achieves the lowest epoch time in all settings, with the speedup of 35.5% -56.8% in 4-layer experiments. Specifically, the epoch time of DGS increases gracefully compared to other baselines, indicating that DGS is capable of facilitating the exploration of deeper GNN model.

3.4.4 Sensitivity Analysis

We evaluate parameters of DGS, N_f and ϵ in terms of accuracy. We vary the value of N_f from 100% to 30%, the value of ϵ from 0 to 1/10, in GraphSAGE training over Reddit.



(a) Using a lower N_f may hurt the final accuracy.



(b) Using a varied ϵ has little effect on the final accuracy.

Figure 3.10: Sensitivity analysis of DGS parameters.

Impact of feature sampling rate N_f on accuracy. Figure 3.10a shows the results of N_f. We find that N_f has no effect in accuracy from 100% to 70%, but starts to degrade the accuracy starting from 60% (from 95.17% to 94.69% and lower). Such phenomenon echos our motivation results in §3.1.4: the graph data is sparse and has various redundant information (30% or more information can be reduced without hurting accuracy).

Impact of node sampling deviation ϵ on throughput. Figure 3.10b shows the results of ϵ. We find that in general, ϵ has no effect on accuracy from 0 to 1/10. This is because METIS partition results in a minimal relationship of local subgraph with remote nodes.

Hence the bias which may occur in DGS that prefer local nodes to remote nodes has no effect upon the final accuracy.

3.4.5 Resource Utilization of Online Explanation

Finally, we measure several resource utilizations of DGS when training with and without online explanation (OE) with GraphSAGE over Reddit. Note that we set T_{ex} to a high value, *i.e.*, 2000, such that the explanation process will not be triggered during the epoch for the w/o explanation case.

Method	**CPU Util.**	**GPU Util.**	**Memory**
with OE	1380.2%	3337MB	20.4GB
without OE	1218.1%	3267MB	18.4GB

Table 3.5: Resource Utilization of Online Explanation.

As shown in Table 3.5, the CPU utilization and memory consumption with OE are 162.1% and 10.9% higher than those without OE, while the GPU utilization almost remains the same. This is because the online explanation processes entirely in CPU and does not consume GPU resources.

3.5 Related Work

GNN Framework. Various works have emerged to improve GNN training. For full-graph training, NeuGraph [116] and ROC [78] process full-graph embeddings synchronization layer-by-layer. As they do not use sampling during training, they may suffer from significant device memory constraints when performing large graph training. For mini-batch training, various frameworks [58, 60, 107, 158, 186, 196, 197] have been extensively used either in academia or industry. Note that DGS can benefit all mini-batch training frameworks by reducing the communication cost.

GNN Training Optimization. Several GNN training optimizations have emerged in recent years. GNNAdvisor [164] explores GNN input properties and proposes optimizations such as workload management and GPU memory customizations. Marius [120] optimizes the data movement during training with partition caching and buffer-aware data orderings. CAGNET [147] proposes distributed-memory parallel GNN training algorithms and reduces the communication costs by dividing the feature vectors into small sub-vectors. BNS-GCN [151] explores partition-parallelism and applies a random boundary sampling strategy for distributed GNN training. However, it applies sampling randomly at node-level, but DGS applies sampling with explanation guidance at fine-grained feature-level. Dorylus [146] brings serverless techniques to GNN training. It breaks down a single training iteration into 4 stages and pipelines each partition based on the workflow of each stage. The above optimizations are orthogonal to DGS.

3.6 Conclusion

This paper proposes DGS, a communication-efficient graph sampling framework for distributed GNN training. Its key idea is to reduce network communication cost by sampling neighborhood information based on the locality of the neighbor nodes in the cluster, and sampling data at the levels of not only graph nodes but also node features. DGS constructs an explanation graph that preserves the relationship between the local graph and remote nodes, and leverages the recently-proposed model explanation technique to design an online explanation scheme that interprets the importance of nodes and features. Our evaluation results show that DGS outperforms existing state-of-the-art sampling algorithms by up to 1.25×, and reduces the communication cost by up to 28.3% while preserving the final accuracy.

CHAPTER 4

G3: A SCALABLE AND EFFICIENT FULL-GRAPH GNN TRAINING SYSTEM

Graph-structured data are natural representations of many real-world applications such as social networks and knowledge graphs. Recent works extend deep neural networks (DNNs) to capture structural information in graphs [66, 91, 149, 173]. This new family of DNNs, known as graph neural networks (GNNs), achieves state-of-the-art performance in machine learning tasks such as node classification [170, 176] and recommendation systems [180].

The reason behind GNN's high expressiveness is that GNNs learn from the relationships between data samples while traditional DNNs are trained over individual samples with no structural information. Figure 4.1 shows GNN computation process that includes *neighborhood aggregation* operations and standard NN operations: in each GNN layer, one node's new embedding is calculated by first aggregating its neighbors' embeddings from the previous layer, and then applying NN operations. This computation process repeats when a node travels through each GNN layer, capturing information from the multi-hop neighborhood of the node.

Training GNNs over billion-edge graphs is time-consuming because of the neighborhood aggregation operation. The common practice to scale out DNN training over large-scale input data is data parallelism [93]. Data parallelism splits data into multiple independent partitions, which can be trained on different workers in parallel. However, data parallelism can no longer be applied to GNN training directly because the neighborhood aggregation operation takes neighboring data samples that may reside on a remote worker. Therefore, when data parallelism splits an input graph into partitions, cross-partition neighboring data samples create large and interleaved computation dependencies among these partitions, making the partitions dependent on each other in data-parallel training. The dependency pattern, which is determined by input graph structure rather than the GNN model itself, complicates the worker synchronization scheme and poses system challenges to efficiently train GNN models in parallel.

Due to the large and interleaved dependencies among partitions in data parallel training, existing GNN systems are difficult to scale to a large number of workers. A widely adopted approach in current GNN systems involves replicating out-of-partition neighboring node data to a worker in order to train GNN over the graph partition independently [60,196], causing duplicate computation and communication systematically. Some systems such as DGL [158], P^3 [60], and BGL [113] use sampling-based training methods to mitigate the overhead by sampling only a small part of the graph during each training iteration. However, sampling operations can generate biased results [43,45] and the duplicate work still exists and grows exponentially to the number of GNN layers [78].

Recently, full-graph training (*i.e.*, no sampling used) [122,129,151,152] has been a popular topic in the machine learning research community for its better performance. While GNN systems including ROC [78] are able to support full-graph training, similar to the methods above, their evaluations still show limited scalability due to imbalanced and duplicate workload, especially when training in large clusters or with deep GNN models [32,99]. Moreover, they are not built for large input graphs because the workers needs to load the entire graph into GPU memory for processing, which is not possible when processing billion-edge graphs.

In this paper, we present G3, a distributed system that can efficiently support full-graph training of GNNs at scale on billion-edge graphs and without accuracy compromise. G3 proposes *GNN hybrid parallelism* to eliminate duplicate work and effectively manage large and complex computation dependencies between workers (§4.2). Furthermore, G3 uses hybrid parallelism to scale out training by dividing the process at the per-node level and sharing intermediate results in a pipelined fashion among peers.

However, GNN hybrid parallelism is not free, as it comes with higher system overhead due to potential stragglers during layer-wise synchronization and more frequent data sharing. Therefore, to maintain a maximum degree of parallelism, G3 distributes balanced computation and communication workload across workers by dividing the input graph with a locality-aware iterative graph partitioning algorithm (§4.3). G3 also overlaps communication with computation using a multi-level pipeline scheduling algorithm (§4.4) that implements both inter- and intra- layer pipelines with an adaptive bin packing strategy. In contrast to pipeline parallelism used in prior works [60, 124, 146] which comes at the cost of accuracy loss, G3's method does not compromise model accuracy.

We evaluate G3 on large datasets with representative GNN models and compare it with state-of-the-art GNN systems. Our results show that G3 can achieve up to 2.24× speedup in a 16-node cluster, and better final accuracy (∼6% higher) over prior works. Besides, we demonstrate G3's ability in exploring complex GNN models and high-dimensional graph datasets: it can achieve up to 25.9× speedup with a 4-layer deep GNN model and 1.94× when training over graphs with 256-dimensional input features.

Overall, this paper makes the following contributions:

- G3 introduces GNN Hybrid parallelism to handle large and complex computation dependencies, enabling scalable GNN training on large graphs without accuracy loss.
- G3 accelerates the training process by balancing workloads across workers with locality-aware iterative graph partitioning, and overlapping communication with computation using multi-level pipeline scheduling.
- We implement a G3 prototype and conduct comprehensive experiments over large graphs, showing that G3 achieves as much as 2.24× speedup in a 16-node cluster, and better final accuracy over prior works.

4.1 Background and Motivation

We first review the computation process of GNN training and then demonstrate the scalability issues in distributed GNN training of prior work through analysis and experiments.

4.1.1 Graph Neural Networks (GNNs)

GNNs are a family of neural networks that learn node representations from graph-structured data (*i.e.*, nodes and links) [66, 91, 149, 173]. The key idea is to aggregate information from neighbors following the graph structure, and perform embedding transformations layer-by-layer.

GNN Computation in One Layer. Figure 4.1 shows the process of GNN computation on the blue node in GNN layer-k, which consists of two stages: neighbor aggregation stage and NN operations stage. In the aggregation stage, layer-k aggregates the node's neighboring nodes' embeddings calculated from the preceding layer (if layer-1, the embeddings are the corresponding nodes' input features). Afterwards, layer-k applies standard DNN operations such as matrix multiplication in Graph Convolution Network (GCN) [91], to the aggregation result, and finally outputs the layer-k embedding for the blue node. All other nodes in the graph process the same above computation in layer-k with their own neighbors.

Formally, the above computation process of GNN layer-k is expressed as follows:

$$\begin{aligned} a_v^{(k)} &= \mathit{Aggregate}^{(k)}(\{h_u^{(k-1)} \mid u \in \mathcal{N}(v)\}) \\ h_v^{(k)} &= \mathit{Update}^{(k)}(h_v^{(k-1)}, a_v^{(k)}) \end{aligned}$$

where $h_v^{(k)}$ denotes the node embedding of node v in the GNN layer-k, and $\mathcal{N}(v)$ denotes neighbors of v. For each node v in layer-k, *Aggregate* first outputs $a_v^{(k)}$ by gathering the embeddings of its neighbors $h_{u\in\mathcal{N}(v)}^{(k-1)}$ with an accumulation function such as mean or sum, then *Update* computes the node's new embedding from its previous embedding $h_v^{(k-1)}$ and the aggregation result $a_v^{(k)}$.

Forward/Backward Propagation. For an L-layer GNN model, the above computation iterates from layer-1 to L among all nodes in the forward propagation. Note that after the layer-L computation, the embeddings $h_v^{(L)}$ capture information for all neighbors within L hops of v, and can be used for downstream tasks such as node classification and link prediction. Next, all nodes' $h^{(L)}$ with their labels are fed to a loss function to generate the loss. With the loss, gradients are then calculated from layer-L to 1 following the chain rule in the backward propagation. Finally, the optimizer updates the GNN model with the gradients attached to it, ending the GNN training for one iteration.

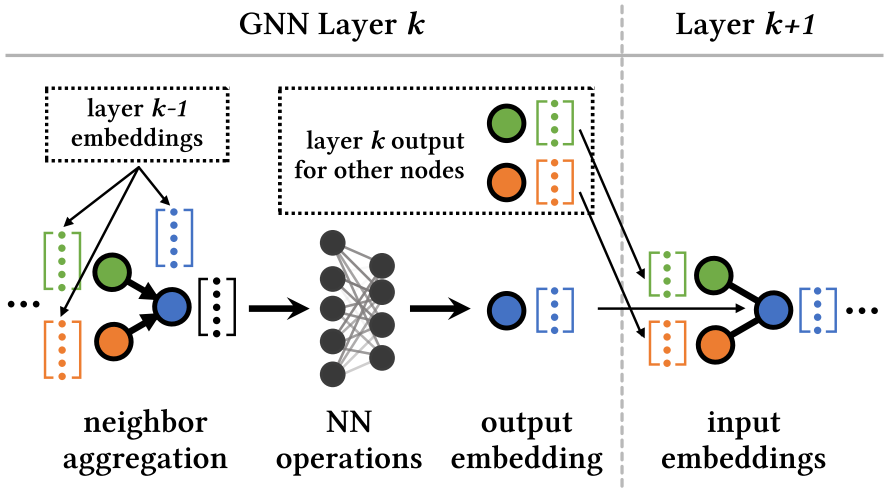


Figure 4.1: Computation of one node in layer k by first aggregating its neighbors' embeddings from previous layer $k-1$, then applying NN operations.

As shown in Figure 4.1, GNN's propagation between consecutive layers is not only over the same data sample but also across different samples due to the neighborhood aggregation operations. These cross-data-sample propagation paths create computation dependencies between data samples.

4.1.2 DNN Parallelization Methods

Data, model, and pipeline parallelism are the most common parallelization methods used in distributed DNN training [63,75,101,156,157,163]. However, we cannot directly apply these methods to GNN training due to its unique aggregation operations (§4.1.1). Here we briefly introduce how they can be adopted and implemented in GNN training, as well as their disadvantages that are specific to the GNN training process.

Data Parallelism. Data parallel training is widely implemented in current GNN systems [78,116,196,197] as large graphs may not be able to fit in a single GPU memory and hence must be partitioned and processed in a distributed manner. Implementing data parallelism in GNN needs to partition the input graph. However, these graph partitions cannot be independently processed, due to the computation dependencies between these partitions created by GNN's neighbor aggregation operations. Existing systems adopt methods such as cross-partition neighbor replication to satisfy these computation dependencies. However, as we show in §4.1.3, these methods introduce system overhead and lead to scalability limitations.

Model Parallelism. In DNN training, model parallelism partitions a model across workers. The worker that holds the input layer of the model is fed with the training data, followed by intermediate results being transmitted sequentially across workers. As the training workflow is still serialized between workers, the advantage is not speeding up the training process, but reducing the memory footprint of a huge model. However, since GNNs typically use small models which commonly have only 2 to 5 layers, model parallelism is not used to accelerate training in previous GNN systems.

Pipeline Parallelism. In pipeline parallelism, the DNN model is partitioned and distributed across workers as in model parallelism. The training data is also split into batches, which are then streamed into the first worker. In the pipeline, every worker computes output activations or gradients for a batch and immediately propagates them to the downstream worker, which hence keeps workers busy and improves workers' utilizations. While pipeline parallelism assumes that batches are entirely independent to process [124], this assumption no longer holds in GNN training. When a worker processes a batch, it needs to re-build a subgraph from the batch, which needs to include multi-hop out-of-batch neighbors for neighborhood aggregation operation in GNN, causing computation overhead due to duplicate work across workers. A study [52] shows that adopting pipeline parallelism in GNN training leads to significant efficiency degradation: training time increases by 5$\times$ in a 4-node cluster compared to the single GPU result.

4.1.3 Training GNNs on Large Graphs

Training GNNs over real-world graphs with tens of millions of nodes and billions of edges poses challenges in hardware capacity and overall efficiency. We put existing methods into two categories and summarize their techniques as follows.

Sampling-based GNN Training. Some GNN training methods use neighbor sampling strategies that down-sample nodes' neighbors in the neighborhood aggregation operation to reduce computation cost [43, 66, 153, 180, 185], and hence speed up the training process. However, as observed in previous works [73, 78], these techniques often ignore a large fraction of neighbors and thus may suffer from accuracy loss. For example, a classical GNN model, GraphSAGE [66], samples and aggregates only a small number (10 to 25) of neighbors for each node. Moreover, the duplicate work still exists and grows exponentially to the number of GNN layers, making neighbor sampling less efficient when training deep GNN models [32, 99]. A recent work [159] also reports that the sampling step is bounded by the I/O throughput of the storage and becomes the bottleneck limiting GPU utilization.

Besides, some sampling-based GNN training methods are used together with graph partitioning when the input graph is too large to fit into a single device. In this case, the result may suffer from both accuracy loss and scalability limitation as well [158].

Full-graph GNN Training. Efficient full-graph GNN training is especially challenging on large graphs as it is both computation-intensive and memory-intensive. Therefore, to improve overall efficiency, many existing GNN systems [78, 116, 196, 197] leverage data parallelism by partitioning and distributing the input graph to multiple workers that collaboratively train the same GNN model. However, the cross-partition computation dependencies in data parallel training make workers dependent on each other. These computation dependencies, determined by the input graph structure, are usually large and complex in real-world graphs. To satisfy the dependencies, two types of schemes are proposed in previous works:

- **Layer-wise communication barriers**. NeuGraph [116] implements layer-wise communication barriers for global synchronization of intermediate results. This communication introduces extra overhead and can be straggled by unbalanced workloads.

- **Cross-partition neighbor replication**. The method of loading all out-of-partition neighboring node features for each worker's assigned partition, as used in DGL [196], ROC [78], and other systems, leads to duplicate computation and communication. This is because the same data may be loaded and computed by multiple workers. Additionally, as GNN training aggregates node embeddings from an L-hop neighborhood, the amount of duplicate work increases exponentially with the number of GNN layers L.

Our experiment confirms the analysis. Figure 4.2 shows the speedup of training throughput for ROC (results from [78]) and DGL on the Reddit dataset [66]. When the number of

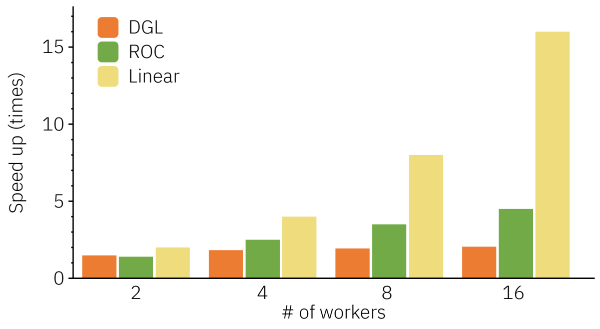


Figure 4.2: Scalability of existing GNN systems without neighbor sampling on Reddit. Numbers are reported based on each system's throughput on 1-node. Linear represents the ideal speedup ratio.

workers scales from 2 to 16, the performance of DGL and ROC only increases as much as 1.4× and 3.2×, far lower than the ideal 8× speedup ratio, as represented by Linear.

Full-graph GNN training is actively researched in today's machine learning research community [129, 151, 159]. However, for the reasons described above, current GNN systems fall short of supporting full-graph training when high accuracy and efficiency are desired at the same time.

4.2 GNN Hybrid Parallelism

Existing systems are less efficient in training GNNs at scale due to the complex computation dependencies created by neighbor aggregation, and the naive adoption of existing DNN parallelization methods results in significant overhead.

We propose *GNN Hybrid parallelism*, a novel parallelization strategy that avoids duplicate work by sharing intermediate results peer-to-peer between GNN layers.

4.2.1 GNN Hybrid Parallelism Workflow

G3 uses GNN hybrid parallelism to enable both data parallelism and model parallelism, while also pipelines the inter- and intra-layer training process. Figure 4.3 depicts a running example on an 8-node graph with 3 workers.

To begin with, the input graph is divided into smaller partitions for training, known

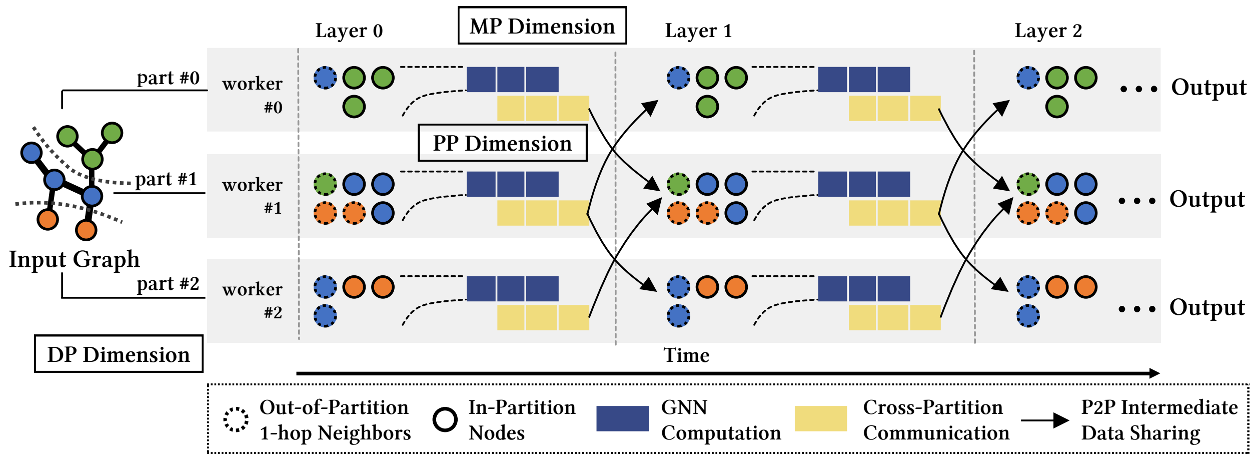


Figure 4.3: A running example of hybrid-parallel training workflow on an 8-node graph with 3 workers. The workflow incorporates three parallel dimensions (data, model, and pipeline). The detailed explanation on the example is in §4.2.1.

as the **data-parallel dimension**. In the given example, an 8-node graph is split into three parts, each marked with a different color. After partitioning, each partition is assigned to a separate worker, and edges connecting the partitions become computation dependencies between the workers.

The training process continues by having each worker train its own GNN model with the assigned partition. Once a worker completes computation on one layer, the output is not only used as input for the next layer computation on that worker, but also sent peer-to-peer to other workers that have dependencies on it (as demonstrated by the cross-worker data-sharing arrows in the given example). This creates the **model-parallel dimension**.

In each layer, nodes are divided into smaller groups called "bins" (depicted as colored rectangles in Figure 4.3) and processed one at a time by a worker, creating a pipeline opportunity for overlapping computation and communication: once a bin's computation is finished, the worker can send its output to other workers and start processing the next bin. These pipelines exist for both inter- and intra-worker tasks, forming the **pipeline-parallel dimension**.

The backward pass is similar to the forward pass, with the difference being that the inputs are gradients rather than node embeddings. Once an epoch is completed, the GNN model parameters are aggregated and averaged using the AllReduce method across all workers.

Benefits in GNN Hybrid Parallelism. GNN hybrid parallelism enables the system to scale with the growth of the graph size and model size for the reasons listed below.

First, Hybrid parallelism handles interleaved computation dependencies in GNN training without the need for cross-partition replication, as the *model-parallel dimension* shares intermediate results peer-to-peer, reducing system overhead.

Second, Hybrid parallelism creates opportunities to overlap computation and communication in inter- and intra-layer training by packing graph nodes into bins and processing them in pipelines.

4.2.2 System Challenges

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

Challenge 1: Balanced Workload Distribution. Data parallelism is a technique that divides a workload into small partitions to process them simultaneously. In order to prevent stragglers in parallel processing, it is important to have a balanced workload in each partition. In GNN hybrid parallelism, communication across partitions becomes significant and it is important to balance the communication workload as well (§4.3.1). Previous works, such as DGL [196] and NeutronStar [159], use off-the-shelf graph partitioning algorithms that only balance computation workload but do not consider the added complexity of communication in GNN training. In GNN training, simply minimizing the global edge-cut on the input graph, like in graph processing tasks, cannot balance the communication workload over each partition (discuss in §4.3.1). To address this issue, we formulate the cost factors and propose a locality-aware iterative graph partitioning algorithm (§4.3) that efficiently balances workloads across workers.

Challenge 2: Efficient Pipeline Scheduling. In GNN hybrid parallelism, workers communicate with each other and share intermediate results to satisfy the GNN computation dependencies across workers. To construct efficient pipelines globally that overlap computation and communication, the order of computation is important as it affects the blocking time of each worker due to cross-worker dependencies. Existing methods, *e.g.*, HGL [64] and JasmineGraph [87], use simple FIFO strategies when processing graph nodes, which does not fully utilize the opportunity to overlap communication and computation. G3 tackles this problem with a multi-level pipeline scheduling algorithm (§4.4) which proactively schedules for both inter- and intra-layer training pipelines to maximize the chance of overlapping, with a bin packing mechanism to assign computation priority and adapt to memory and model size.

4.2.3 Comparison with Recent Works

We note that similar ideas of GNN hybrid parallelism have been explored in some recent works [60, 151]. However, existing solutions fail to fully exploit the opportunities to reach the maximum degree of parallelism. They did not identify the challenges of high system synchronization overhead raised by the idea of GNN hybrid parallelism. More specifically, existing solutions (1) did not balance cross-partition communication in graph partitioning algorithms, and (2) did not explore more efficient pipelines with minimized waiting time by assigning different computation priorities.

G3 fully explores these challenges that are unique in distributed GNN training, presents comprehensive optimization and implementation to address the system challenges, and evaluates the solutions with both end-to-end and microbenchmark experiments.

Moreover, in the pipeline-parallel dimension, nodes are grouped into smaller groups called *bins* and processed in pipelines. This allows G3 to adapt bin sizes to the GPU memory capacity and enables G3 to handle larger graphs, unlike full-graph training systems like NeutronStar [159] and BNS-GCN [151] which may encounter GPU memory limitations with large graphs.

4.3 Balanced Work Partitioning

In distributed GNN training, achieving a balanced workload among partitions requires more than just equal graph nodes and global min edge-cut. In this section, we formulate the cost factors in distributed GNN training workload and propose a locality-aware iterative graph partitioning algorithm that efficiently balances these factors across workers.

4.3.1 Cost Factors in Distributed GNN Training

Notation	Description
$G = (V, E)$	Input graph with its node and edge set
$P(G, k) = \{V_1, ..., V_k\}$	k-partition result of graph G V_i represents the nodes in partition i
$E(V_i)$	Edges in G that destine to nodes in V_i
V_i^{remote}	Out-of-partition neighbors of partition i
d_v	Degree of node v from G
$N(v)$	Neighbors of node v from G

Table 4.1: Notations and factors affecting GNN workload over a partition in Hybrid parallelism.

When assigning a graph partition to a worker, several graph-related factors affect the worker's training workload. We summarize several factors which determine the workload of partition V_i as follows:

- Number of nodes $|V_i|$, which directly contributes to the GNN computation cost.
- Number of edges $|E(V_i)|$, which affects the GNN computation cost due to neighborhood aggregation.
- Number of out-of-partition neighboring nodes $|V_i^{remote}|$, which determines the amount of data to be transferred out to other workers after each layer's computation. This is essentially the peer-to-peer communication cost in the model parallel dimension of Hybrid parallelism.

Chunk-based graph partitioning method, used in NeuGraph [116], ROC [78] and NeutronStar [159], partitions the graph into chunks that contain nodes with consecutive IDs. While it balances $|V_i|$, the *actual* amount of workload may be highly unbalanced because the method does not consider the total number of edge cuts, resulting in unbalanced $|E(V_i)|$ as well as large and unbalanced $|V_i^{remote}|$.

METIS algorithm [88], used by DGL [196] and BNS-GCN [151], can find a partition decision with minimum edge-cut. However, minimizing edge-cut does not balance $|V_i^{remote}|$ for each partition i. Hence, the communication workload across workers still varies.

4.3.2 Locality-aware Iterative Partitioning

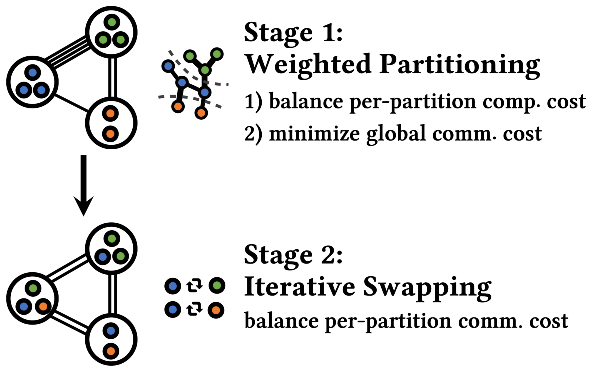

Stage 1:
Weighted Partitioning
1) balance per-partition comp. cost
2) minimize global comm. cost
Stage 2:
Iterative Swapping
balance per-partition comm. cost

Figure 4.4: Two stages of partitioning algorithm

The partitioning algorithm has two stages as depicted in Figure 4.4: (1) the weighted partitioning stage that balances per-partition computation cost and minimizes global communication costs. (2) the iterative re-partitioning stage that balances per-partition communication costs.

First, in the weighted graph partitioning stage, the algorithm generates a graph partitioning decision with a balanced computation cost among partitions while minimizing the global communication volume. The global communication volume is defined as the sum of each partition's communication volume, which is determined by the number of each partition's remote nodes. We formalize our optimization objectives as follows:

$$\text{minimize } T_{max\_nn} = \max_{V_p \in P} |V_p|, \tag{4.1}$$

$$\text{minimize } T_{max\_aggr} = \max_{V_p \in P} \sum_{v \in V_p} d_v, \tag{4.2}$$

$$\text{minimize } T_{total\_comm} = \sum_{V_p \in P} |V_p^{remote}|, \tag{4.3}$$

where $P = \{V_1, V_2, ...\}$ denotes the partitioning decision and V_i contains all nodes v in partition i. Equation 4.1 and Equation 4.2 essentially balance computation cost among partitions while Equation 4.3 minimizes global communication cost.

The objectives above formulate a multi-constraint graph partitioning problem [89]: a vector of weights is assigned to each node, and the goal is to produce a partitioning such that the partitioning satisfies a balanced constraint associated with each weight, while attempting to minimize the global communication volume (or edge-cut).

In our formulation, the balanced sum of node feature size (Equation 4.1) and the balanced sum of node degree (Equation 4.2) in each partition are two constraints while we aim at minimizing the total communication volume (Equation 4.3).

Note that even though the global communication volume is minimized by the algorithm above, the communication cost of each partition cannot be balanced under the multi-constraint graph partitioning problem setting, as it is a result of the actual partitioning and hence cannot be pre-defined as a constraint.

Second, in the iterative re-partitioning stage, the algorithm swaps nodes between partitions to balance the communication volume in each partition. We formalize our optimization objective as follows:

$$\text{minimize } T_{max\_comm.} = \max_{V_p \in P} |V_p^{remote}| \tag{4.4}$$

Algorithm 2: Locality-aware Iterative Partition

```
Data:
(1) Input Graph: $G$
(2) Number of Workers: $n$
(3) Multi-constraint Partition: $P^{Multi}(G, n) = \{V_1, .., V_n\}$
Result: Final Partition $P = \{V_1, .., V_n\}$
1  begin
2      $P \longleftarrow P^{Multi}(G, n)$
3      while true do
4          $V_{max} \longleftarrow \text{argmax}_{V_p \in P} |V_p^{remote}|$
5          $V_{min} \longleftarrow \text{argmin}_{V_p \in P} |V_p^{remote}|$
6          $ratio \longleftarrow |V_{min}^{remote}| / |V_{max}^{remote}|$
7          if ratio is converged then
8              return $P$
9          $v_{max} \longleftarrow \text{argmax}_{v \in V_{max}} |v^{remote}|$
10         $v_{min} \longleftarrow \text{argmin}_{v \in V_{min}} |N(v) - V_{max}|$
11         $V_{max} \longleftarrow V_{max} - \{v_{max}\} + \{v_{min}\}$
12         $V_{min} \longleftarrow V_{min} - \{v_{min}\} + \{v_{max}\}$
```

The algorithm is shown in Algorithm 2. In each iteration, we first find two partitions that have the highest and lowest number of remote nodes, denoted as V_{max} and V_{min}. Then we consider the locality in each node by picking a node v_{max} in V_{max} which has the highest number of remote neighbors (*i.e.*, out-of-partition nodes in its neighborhood) and a node v_{min} in V_{min} which has the lowest number of remote neighbors if moved out of V_{min}, and then we swap these two nodes. The above process repeats until the difference between $|V_{min}^{remote}|$ and $|V_{max}^{remote}|$ converges below a threshold γ (we use 0.5% in our experiment). We also stop the process if cyclic swapping.

Fast Neighbor Tracking Algorithm. Finding every partition's out-of-partition neighbors V_i^{remote} can be time-consuming (line 4–5) because the search of all nodes in V_i takes $O(|V|)$ time. We present an algorithm that enables incremental updates to track each node's out-of-partition neighbors. The algorithm reduces the time complexity to $O(d_v)$, where v is the node moving in or out.

Making incremental updates in V_i^{remote} is not straightforward. This is because moving a node out of V_i does not necessarily remove all its remote neighbors from V_i^{remote}, as those neighbors could be remote neighbors to other nodes in V_i. To solve this problem, we maintain the in-partition degrees d_v^{in} for every node $v \in V_i^{remote}$. For example, when a node v moves out from V_i, the in-partition degrees of v's remote neighbors, v^{remote}, will be decreased by 1, and the node will be removed from V_i^{remote} if its d_v^{in} reaches 0. The technique is used when implementing node swapping (line 12–13) between the two affected node sets, rather than triggering a global recount. Moreover, as the search procedures (line 4–5 and line 10–11) are independent without dependencies, they are parallelized on multiple CPU cores with shared memory.

4.4 Multi-level Pipeline Scheduling

In GNN Hybrid parallelism, workers share intermediate results to meet computation dependencies. To achieve efficient cross-worker pipelines to overlap computation and communication, the computation sequence is crucial as it impacts worker waiting time. G3 proposes a multi-level pipeline scheduling algorithm that exploits opportunities for host-GPU communication, GPU computation, and network communication via inter- and intra-layer pipelines.

4.4.1 Bin Packing Mechanism

The multi-level pipeline scheduling algorithm utilizes a bin packing mechanism: In each layer, nodes are divided into smaller groups called *bins* and processed one at a time by a worker. Once a bin's computation is finished in one layer, the worker can send its output to other workers and start processing the next layer.

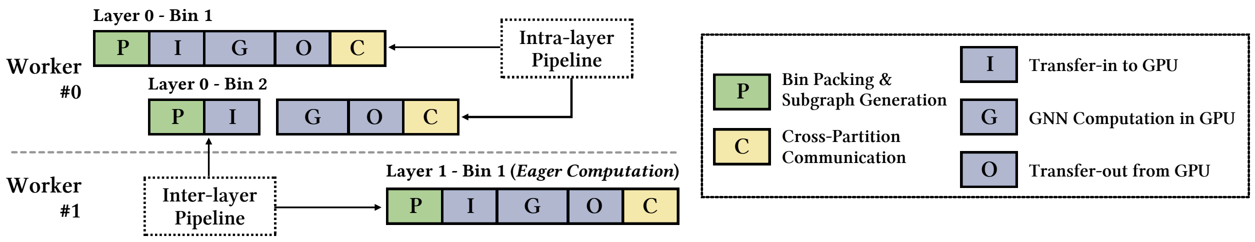


Figure 4.5: An example of the intra-layer pipeline and inter-layer pipeline with two bins originated from worker #0.

The lifespan of a bin has 5 stages in the forward pass, as shown in Figure 4.5. First, the scheduling algorithm decides on the allocation of nodes and generates a subgraph containing these nodes and their neighboring nodes (even if initially they are not in the bin). Next, the subgraph and its node embeddings will be transferred from host memory to GPU memory and used for GNN computation. Finally, the output will be transferred out from GPU and shared with remote workers when required.

The backward pass is similar to the forward pass, except that the inputs are gradients instead of node embeddings. We reuse the bin packing decision in the forward pass as the backward pass of a bin is performed in exactly the reverse direction of the forward. The scheduling decision made in the forward is a good indication of how the backward should be scheduled.

Analysis and Comparison. Utilizing the bin packing mechanism in GNN training allows for fine-grained division of the training process at the per-node level, providing opportunities for overlapping computation and communication. Furthermore, it helps avoid loading the entire graph onto the GPU, which can prevent GPU memory issues when the input graph is large (§4.6.2), as seen in other GNN systems [151, 159].

We note that HGL [64] and JasmineGraph [87] also utilize bin packing, but they pack bins at the subgraph level and process them in a FIFO order without any priority setting. In contrast, G3 packs bins at the node level and adapts to GPU memory capacity to improve utilization while avoiding running out of GPU memory. G3 also proactively schedules inter- and intra-layer pipelines based on computation priority (§4.4.2).

4.4.2 Pipeline Strategies

In GNN Hybrid parallelism, cross-partition dependencies are handled by sharing intermediate results via peer-to-peer communication. Specifically, a node's embedding is sent to the workers where its remote neighbors are located. To optimize efficiency, the communication of a bin is overlapped with the computation in the next layer, either within the same worker (intra-layer pipeline) or on a remote worker (inter-layer pipeline), as shown in Figure 4.5.

(1) Inter-Layer Pipeline

We propose two designs that maximize overlapping opportunities in inter-layer pipelines.

Node Computation Priority. Since the total communication cost for a layer is fixed and distributed evenly among workers with our partitioning algorithm (in §4.3), we aim to begin data transmission as early as possible. Therefore, our scheduling algorithm (described in §4.4.3) prioritizes computation for nodes that have more remote neighbors outside of their partition, as these nodes generate more communication workload.

Eager Computation. From the perspective of the receiving worker, as soon as the worker completes computation for the current layer, it can begin GNN computation on nodes that are ready for the next layer. A node is considered ready when its remote neighbors' embeddings have all been received from other workers.

When the memory footprint of ready nodes exceeds a threshold, eager computation packs them together and begins processing. The threshold is set to half the size of the GPU memory. This threshold is consistent with the bin size in the intra-layer pipeline (§4.4.2) to create smooth pipelines.

The above optimization opportunity exists because our node priority setting ensures that other workers will begin transmitting output embeddings as soon as possible. This way, the receiving worker can begin processing some of the nodes in the next layer using the partial results received from other workers.

(2) Intra-layer Pipeline

Computing with GPUs requires loading data into GPU memory. Because GPU memory is relatively small compared to host memory, even when the input graph is partitioned across multiple workers, individual graph partitions may still be larger than the GPU memory of a worker. Therefore, as described in the bin packing mechanism, a layer's computation might be split into multiple bins and processed in the GPU sequentially (Figure 4.5).

In order to construct an effective pipeline that overlaps GPU computation and host-GPU communication, we utilize at most half the size of GPU memory for each bin so that one bin can be processed while another is loading in or out of the GPU. Additionally, the algorithm adapts bin sizes based on the memory requirements of each model layer to optimize efficiency. Since different models have different memory footprints, our algorithm adaptively adjusts bin sizes based on the model structure.

4.4.3 Scheduling Algorithm

We now introduce the multi-level pipeline scheduling algorithm that realizes the bin packing mechanism and pipeline strategies outlined above.

Step 1. Sorting Nodes. The worker uses BFS to traverse its assigned graph partition from any node in every connected component, sorting the nodes in the order of their visit. Then, the worker separates the nodes with remote neighbors from the rest and prioritizes them by placing them at the front of the list, creating the priority list $\mathcal{L}$. This step corresponds to the node computation priority setting in inter-layer pipelines.

Step 2. Calculating Bin Size Limit. We estimate the GPU memory required for one node in a GNN layer by using the layer's input and output embedding size. The bin size limit is then adaptively calculated for each layer by dividing half of the GPU memory size by the per-node memory usage. This ensures that each bin can be processed while another is loading in or out of the GPU without exceeding the GPU's memory limit. This step decides the bin size limit for intra-layer pipelines.

Step 3. Index-based Bin Packing. We iterate through the priority list $\mathcal{L}$ sequentially (index-based), adding ready nodes and their neighbors into a bin. Once a bin reaches its limit, we move on to a new bin and continue adding until all nodes in $\mathcal{L}$ are processed. This step schedules bins for intra-layer pipelines on each worker and realizes eager computation.

Index-based partitioning preserves locality when adjacent nodes are stored close to each other as [78, 198] shows, and our BFS-based sorting in the first step creates such locality, which increases the chance of packing nodes and their neighbors into the same bin and helps accelerate neighborhood aggregation in GNN training.

Analysis. The scheduling algorithm's **time complexity** is primarily determined by the BFS used for the computation priority of nodes, which takes $O(|E| + |V|)$, the same as initializing the graph data. In practice, GNN training begins on the GPU after the first bin is generated on the CPU. While the remaining bins are generated on the CPU, their costs do not affect overall performance as they are hidden under the GPU computation.

Proving the **correctness** of the algorithm is straightforward, as it only alters the computation order for the nodes without affecting the computation itself in GNN training. Our experiments on time-to-accuracy (§4.6.3) also confirm the correctness of the algorithm and our implementation.

4.5 Implementation

In this section, we give an architecture overview of G3 (§4.5.1), and present the implementation details of the G3 prototype (§4.5.2).

4.5.1 Overview

The workflow of G3 is shown in Figure 4.6. G3 takes a large graph as input and goes through two stages (*i.e.*, locality-aware iterative graph partitioning and multi-level pipeline scheduling). In particular, the first stage partitions the input graph iteratively in order to balance the workload according to the cost model. Then, the second stage trains the GNN

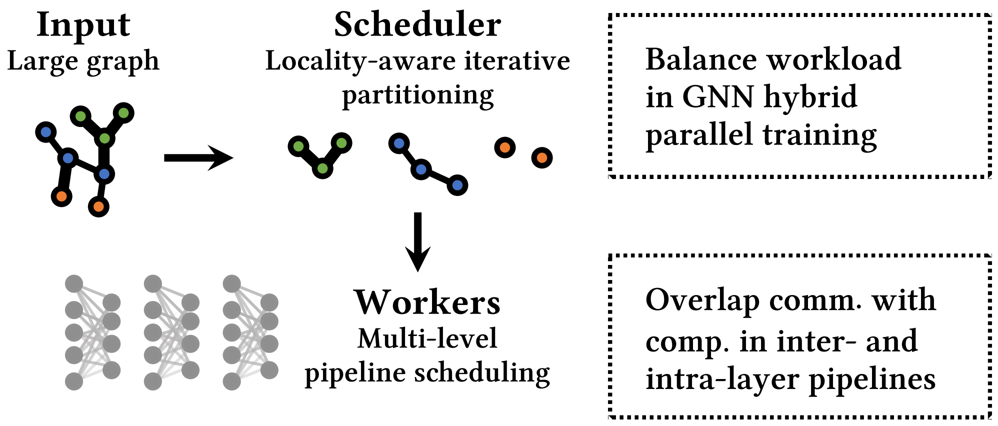

Input
Large graph
Scheduler
Locality-aware iterative partitioning
Balance workload in GNN hybrid parallel training
Workers
Multi-level pipeline scheduling
Overlap comm. with comp. in inter- and intra-layer pipelines

Figure 4.6: Overview of G3 System

using inter- and intra-layer pipeline scheduling to overlap communication with computation.

The graph partitioning stage is processed offline in the **scheduler**. For the online scheduling stage, the architecture of G3 takes advantage of the scheduler (*i.e.*, global and local task scheduler) to synchronize the training process. Specifically, each **worker** in the scheduling stage runs a local task scheduler to schedule its local task and an executor process for training tasks, while the worker with rank 0 runs a global scheduler that coordinates the model synchronizations across the cluster.

4.5.2 System Implementation

We implement G3 purely in Python. The functionalities in G3 include graph partition, task execution, and communication handlers between nodes in the graph. We build G3 on top of DGL for GNN operations and PyTorch for parameter synchronization, respectively.

Scheduler. The scheduler process takes charge of graph partitioning and online model coordination. Before training, the scheduler generates each partition's nodeID lists as described in Algorithm 2. The lists are then passed to DGL partition APIs to partition the graph into corresponding subgraphs, which contain all nodes listed and the edges destined for them. As described in §4.3.2, to accelerate the partition process, G3 parallelizes it through multiple CPU cores. During training, the global scheduler runs a coordination thread for layer-wise parameter synchronization. Note that G3 wraps and synchronizes each layer's parameters separately. Therefore, the coordination thread is responsible for monitoring the layer-wise BP programs in all workers, and notifying all workers to simultaneously update the corresponding layer parameters once all workers have finished the computation.

Workers. The worker process takes charge of P2P data sharing, online data-loading, and task execution. The worker first analyzes its subgraph and sets up send&recv hash tables for P2P data sharing. The hash tables contain dst/src workers with respect to their nodeIDs, which indicate the P2P inter-layer transmission paths. As each worker processes upon its dedicated partition throughout training, such tables remain unchanged and reside permanently in host memory for fast lookup. For the data-loading part, the worker achieves fast online bin packing by leveraging unused CPU resources (the same way as partition does) to judge ready nodes, and spawns data-loading threads for each bin to wrap the bin's nodes and their incident edges into `DGLBlock` format and then pushed into `TaskQ` for computation. The main process consumes tasks sequentially, and then splits the output result into pieces and spawns threads to send them out, referring to the generated P2P hash tables.

4.6 Evaluation

In this section, we evaluate G3 with comprehensive testbed experiments. We first show the scalability of G3 by comparing it with mini-batch systems (§4.6.1), and its throughput performance compared to other full-graph systems (§4.6.2). Then we show G3's fast convergence speed and better accuracy compared to all baselines (§4.6.3). We also report the GPU utilization of each system (§4.6.4). Furthermore, we show the potential of G3 by exploring more complex GNN models and graphs in terms of layers and features (§4.6.5–§4.6.6). Finally, we demonstrate the effectiveness of G3's designs by comparing them with the state-of-the-art strategies (§4.6.7).

In summary, our key results reveal that:

- G3 achieves higher throughput than all baselines by up to 2.24$\times$ in a 16-node cluster, and as much as 5.5$\times$ better GPU utilization.

- G3 achieves up to 2.3× faster convergence speed and better final accuracy (∼6% higher) compared to baselines.
- G3 is able to explore more complex GNN models and graph datasets. It achieves as much as 25.9× speedup with a 4-layer GNN model, and 1.94× speedup when training over graphs with 256-dimensional features compared to DGL.
- The microbenchmarks reveal that balanced partition and multi-level pipeline improve the training performance by up to 11.66×, and the benefits increase when scaling.

Experiment setup. We evaluate G3 using 8 physical servers (each with 2 RTX 3090 GPUs, 80 CPU cores (2.1GHz Intel Xeon Gold 5218R), 256GB RAM, and 2 Mellanox ConnectX5 NICs), and 4 Mellanox SN2100 switches. We divide one physical server into two docker containers, each with a 3090 GPU, 40 CPU cores, 128GB RAM, and a 10Gbps virtual Ethernet interface[1] to get a 16-node testbed. All nodes run 64-bit Ubuntu 18.04 with CUDA v11.1, DGL v0.6.1, and PyTorch v1.10.1.

	Ogbn-products	**Amazon**	**Ogbn-papers**	**Twitter-2010**
Nodes	2.45M	1.60M	111.1M	41.65M
Edges	61.86M	132.2M	1.616B	2.405B
Features	100	200	128	128
Classes	47	107	172	100
Avg. Deg	50.5	82.7	29.1	57.7

Table 4.2: Graph datasets used in our evaluation.

Datasets. Table 4.2 lists four graph datasets that we used in our evaluation, including three popular GNN datasets: Ogbn-products [71], Amazon [185], and Ogbn-papers [71], and one graph dataset Twitter-2010 [36,37]. We generate random features for Twitter-2010 ensuring that the ratio of labeled nodes remains consistent with what we observed in the OGB datasets, and only use the throughput results when training over it.

GNN models & Metrics. We use three representative GNN models, GraphSAGE [66], GCN [91], and GAT [149] for evaluation. By default, we adopt standard 2-layer GNNs

[1]We use SR-IOV to separate the resource of physical NIC. [54] shows that it achieves nearly the same performance as the non-virtualized environments.

with a hidden dimension of 16 in all experiments. For GAT, we use 8 attention heads. We test the performance of G3 on node classification tasks (*e.g.*, predicting the category of a product in the Ogbn-products dataset). We use *throughput* as our evaluation metric, which is the sum of all workers' average throughput.

Baseline. We compare G3 with two kinds of GNN training systems, *i.e.*, mini-batch system and full-graph system, for different evaluation purposes.

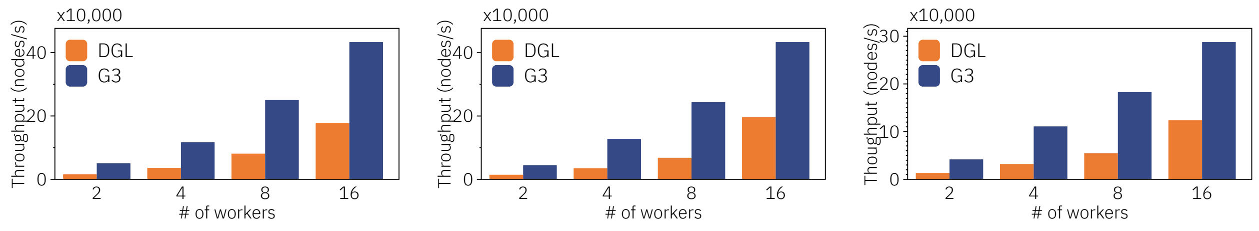
For mini-batch systems, we compare G3 with DGL [158, 196], a representative deep learning library for graphs, in terms of scalability and accuracy. We use DGL with PyTorch backend and conduct distributed training experiments. We use METIS, the default partition algorithm of DGL, in all DGL evaluations. We also compare with ClusterGCN [45], a method that does not synchronize features among workers during training, in terms of time-to-accuracy. Note that we do not compare with it upon throughput, as it has no feature communication throughout training and hence always achieves linear scalability, but at the cost of significant accuracy loss (see §4.6.3).

For full-graph systems, we compare G3 with BNS-GCN [151] and NeutronStar [159] in terms of throughput, accuracy, and GPU utilization. For system configurations, we maintain the default partition strategies for each system, *i.e.*, using METIS for BNS-GCN and a chunk-based approach for NeutronStar. We set the sampling ratio to 1 to ensure full-graph training in BNS-GCN. We implement GraphSAGE aggregators based on the existing GCN implementation in NeutronStar. We excluded these systems from our scalability experiments as they are not specifically designed for scalability. As previously reported in their original papers (Figure 5 in[151], Figure 12 in [159]), these systems show diminishing returns when increasing the number of workers. This is primarily due to large system overhead from cross-partition replication, as well as unbalanced workloads and underutilized overlapping opportunities, as discussed in §4.1.3. Additionally, these systems load the entire graph directly onto the GPU for parallel processing, making it infeasible to train GNNs on billion-edge graphs due to GPU memory constraints (Figure 4.11), especially when the cluster size is small.

Training pattern. In our experiments, G3, BNS-GCN, and NeutronStar perform full-graph training, while DGL performs sampling-based mini-batch training. For mini-batch training, we set the batch size of DGL to 1000 in all experiments. For a fair comparison upon throughput, we set our initial bin size (§4.4.2) equal to the batch size when comparing G3 with DGL. For neighborhood sampling in DGL, we adopt the sampling strategy (25, 10) [66] in all experiments. For full-graph training, we set G3's initial bin size to different values according to the input graph. For small graphs including Ogbn-products and Amazon, the initial bin size is set to 5000 to achieve high initial throughput like other full-graph systems, as they parallelize training by directly loading the whole graph into GPUs. When training over large graphs including Ogbn-papers and Twitter, we set G3's initial bin size to 1000 to avoid GPU memory explosion.

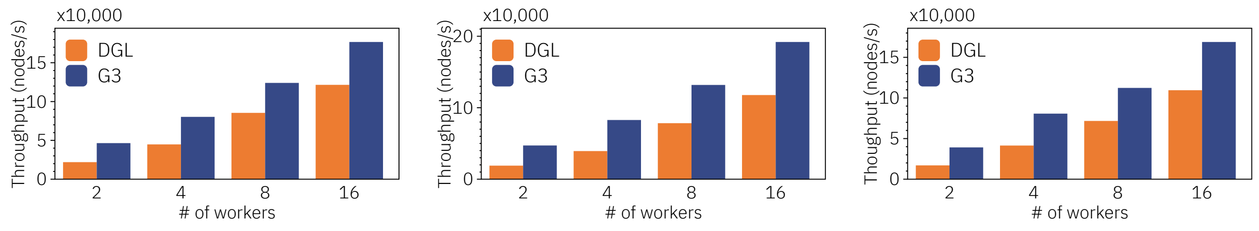
4.6.1 Scalability

We demonstrate G3's scalability by comparing with DGL over four datasets in a cluster whose size ranges from 2 to 16. We train three models on each system to obtain the global throughput. The results are shown in Figure 4.7, Figure 4.8, Figure 4.9, and Figure 4.10.



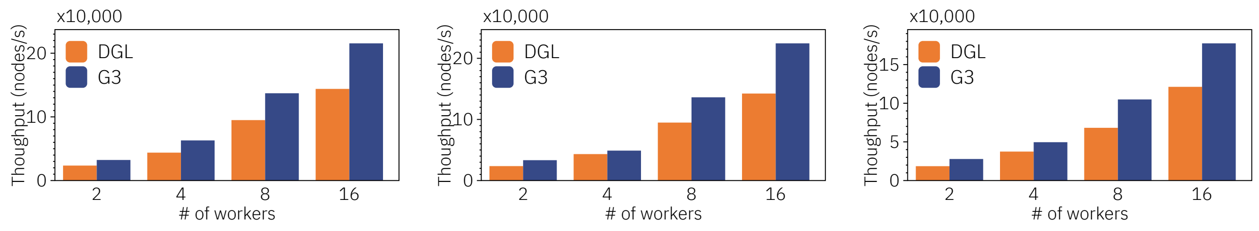
(a) GraphSAGE (b) GCN (c) GAT

Figure 4.7: Training 3 GNN models over Ogbn-products. G3 is able to gain up to 3.66× higher throughput over DGL.



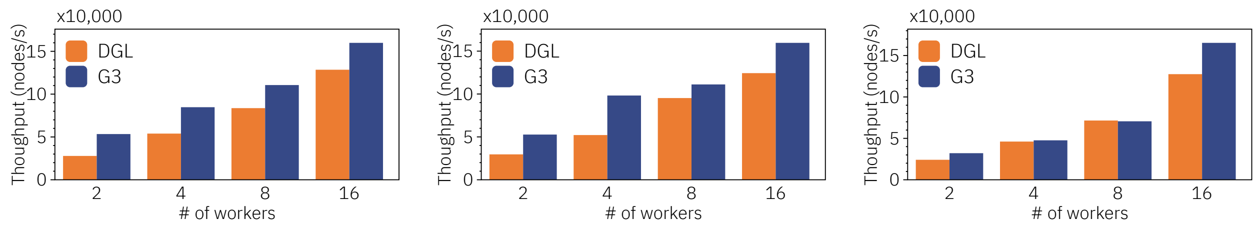
(a) GraphSAGE (b) GCN (c) GAT

Figure 4.8: Training 3 GNN models over Amazon. G3 is able to gain up to 2.48× higher throughput over DGL.



(a) GraphSAGE (b) GCN (c) GAT

Figure 4.9: Training 3 GNN models over Ogbn-papers. G3 is able to gain up to 1.47× higher throughput over DGL.



(a) GraphSAGE (b) GCN (c) GAT

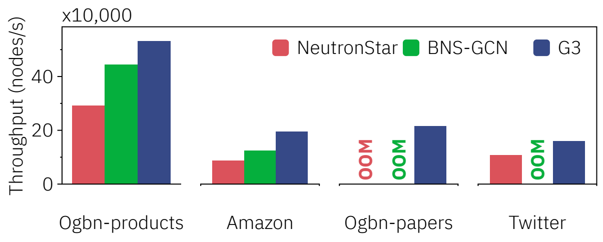
Figure 4.10: Training 3 GNN models over Twitter-2010. G3 is able to gain up to 1.91× higher throughput over DGL.

Different Systems. DGL shows poor throughput performance in all models with increasing cluster size. For example, when training over Ogbn-papers with the cluster size from 8 to 16, we only observe as much as 1.32× speed up compared to the double cluster size. This corresponds with the discussion in §4.1 that even with sampling strategies, the data parallelism scheme adopted by DGL suffers from duplicate computation and communication among workers. Compared to DGL, G3 achieves better scalability improvement almost in all cases. For example, G3 speeds up training by 1.64× and 1.74× on average over Ogbn-papers and Twitter-2010 when the cluster scales from 8 to 16, which is higher than those of DGL, *i.e.*, 1.32× and 1.44×. We remind the readers that training in large graphs is more challenging as their dependencies are more complex and would exaggerate the scalability issue. Therefore the results above indicate better scalability of G3 than DGL. Overall, G3 outperforms DGL by 2.04×, 1.51×, 1.32×, and 1.13× in a 16-node cluster when training over four datasets, respectively.

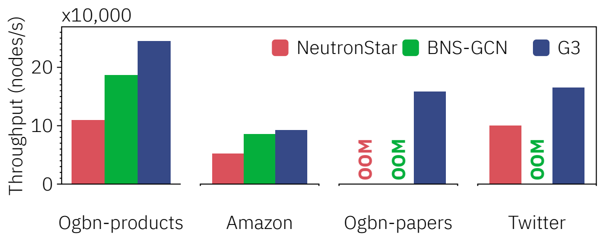
Different GNN models. The training performance varies significantly across different GNN models. In a 16-node cluster, while G3 achieves higher performance improvement (1.24–2.45× speedup) with GraphSAGE and GCN models, it only speeds up GAT training process by 1.25–1.46×. In specific, G3 and DGL achieves almost the same throughput when the cluster size is small (from 2 to 8). This is because: 1) GAT model is more computation-intensive; 2) the improvement of communication is marginal when in small clusters. Hence, there is little room for G3 to optimize training. Moreover, GAT training involves extra intermediate attention tensors generation during forward propagation. The extra generated tensors increase the data volume transmitted between CPU and GPU, which therefore slows down the training process.

4.6.2 Comparison with Other Full-graph Systems

We compare with other full-graph systems, *i.e.*, BNS-GCN and NeutronStar, in terms of training throughput over four datasets in a 16-node cluster. We train GraphSAGE and GAT models in each system and obtain the global throughput.



(a) GraphSAGE



(b) GAT

Figure 4.11: Full-graph training over 4 datasets. G3 is able to gain 1.08-2.24× higher throughput over baselines.

The results are shown in Figure 4.11. G3 achieves 1.08-2.24× speedups over two baselines in all settings. NeutronStar performs worst in all settings because it adopts the chunk-based partition strategy that requires a significant communication workload. BNS-GCN performs better because METIS partition minimizes global edge-cut and thus incurs a low global communication workload. However, such strategy cannot balance communication for each partition, as described in §4.3.1, and hence may have straggler issues. Besides, both systems require loading every subgraph in each assigned worker for full-graph training. Such manner has two issues: 1) It fails to exploit the overlapping opportunities between computation and communication; 2) It results in an out-of-memory error when training over large graphs, as the size of the whole subgraph can easily exhaust the GPU memory in each worker. For G3, the balanced partition strategy it adopts achieves both low global communication and balanced communication for each partition, resulting in a balanced subgraph distribution for training. Moreover, the multi-level pipelining achieves: 1) fine-grained bin-level overlapping of communication with computation; 2) training over large graphs without out-of-memory error as it supports swapping bins between CPU and GPU to relieve resource constraints. Though such memory swapping comes with a throughput slowdown, we believe it is necessary for the tradeoff between memory constraint and training efficiency. Overall, G3 outperforms all baselines, and the mean speedups over BNS-GCN and NeutronStar are 1.38×/1.20× and 1.84×/1.89× for GraphSAGE/GAT training, respectively. G3 speeds up less with GAT because of the higher computation workload in GAT and extra memory swapping overhead, as illustrated in §4.6.1.

4.6.3 Time-to-Accuracy

In these experiments, we compare G3 with baselines in terms of time-to-accuracy. We conduct experiments using G3, BNS-GCN, and NeutronStar over Ogbn-products and Amazon, and using G3, DGL, and ClusterGCN over Ogbn-papers. We train GraphSAGE in all experiments in a 16-node cluster, and record the test accuracy during training.

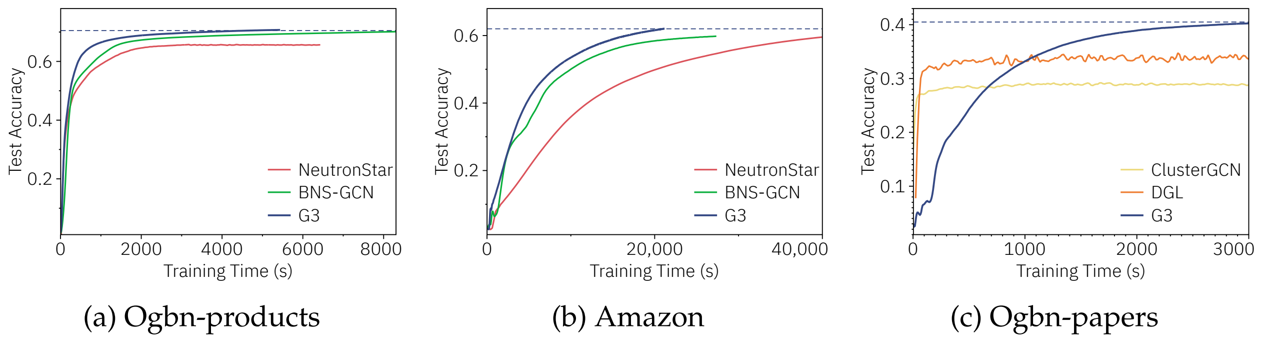


Figure 4.12: Time-to-accuracy results over three datasets. G3 achieves 1.8-2.3× faster convergence speed than full-graph baselines, and 6%-10.7% higher final accuracy than mini-batch baselines.

The results are shown in Figure 4.12. The dashed lines are the approximate final accuracies that G3 achieves in each dataset, *i.e.*, 70.5% for Ogbn-products, 62% for Amazon, and 40.5% for Ogbn-papers, respectively.

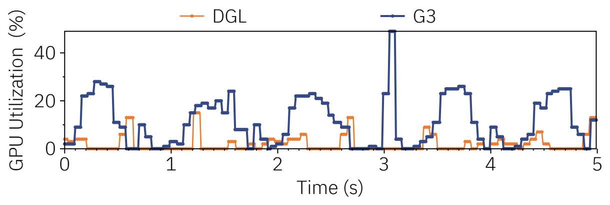
As Figure 4.12a and Figure 4.12b for small graphs show, three full-graph training systems all converge to similar final accuracy. But G3 converges fastest among the three, *i.e.*, 1.8-2.3×, thanks to its higher training throughput, which can be attributed to the balanced partition and multi-level pipelining.

As Figure 4.12c for the large graph shows, initially, ClusterGCN converges the fastest among the three because it has no feature synchronization among workers, and hence has the minimal epoch time (∼3.2 seconds). However, due to the lack of neighbor information from remote hosts during training, ClusterGCN can only converge to as much as 29.8%. Afterward, we see no accuracy improvement with more training epochs. DGL converges faster but can only converge to low final accuracy, *i.e.*, ∼6% lower than the value G3 achieves. The faster convergence speed that DGL achieves is due to the property of mini-batch training. Such paradigm updates the model during every iteration, which is a higher update frequency than that of full-graph training that G3 adopts (*e.g.*, 75 vs. 1 in Ogbn-papers). As a result, the frequent update manner converges faster, but at the cost of significant final accuracy loss due to the sampling in training (§4.1.3). For G3, it initiates the slowest among all due to the property of full-graph training. But then it converges rapidly and finally achieves the best accuracy thanks to its high efficiency and full-graph training. Overall, G3 achieves the best final accuracy than DGL and ClusterGCN. The higher accuracy of G3 can be attributed to the full-graph training adopted by G3 than the mini-batch training adopted by DGL [78].

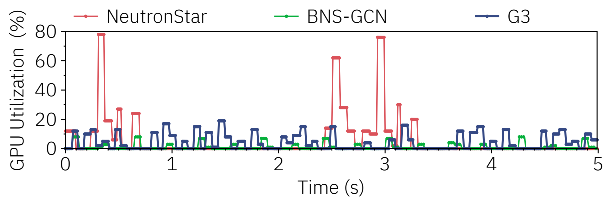
4.6.4 GPU Utilization

We report the GPU utilization of G3 compared with baselines when training GraphSAGE over Ogbn-products in a 16-node cluster. We measure the GPU utilization every 10 milliseconds. We use different initial bin size settings for G3 when compared to mini-batch and full-graph systems, as specified in §4.6.1 and §4.6.2. The results in a five-second window are shown in Figure 4.13.

As Figure 4.13a shows, we find that G3 achieves both higher peak GPU utilization (49% vs. 15%) and higher average GPU utilization (11.2% vs. 2%). Note that the small utilization value is due to the sparse computations in GNN models that fail to leverage GPU efficiency [60]. DGL achieves lower GPU utilization because its sampling becomes



(a) GPU utilization compared with mini-batch systems. The peak/average GPU utilizations are 49%/11.2% and 15%/2% for G3 and DGL, respectively.



(b) GPU utilization compared with other full-graph systems. The peak/average GPU utilizations are 19%/5.4%, 78%/4.8%, and 8%/1.1% for G3, NeutronStar, and BNS-GCN, respectively.

Figure 4.13: G3 achieves 5.5×/2.9× higher average GPU utilization than mini-batch/full-graph baselines.

the bottleneck and limits the GPU utilization, as elaborated in §4.1.3. For G3, however, it remains high GPU utilization thanks to the balanced workload across workers and the multi-level pipelining to exploit overlap opportunities.

As seen in Figure 4.13b, NeutronStar has a higher peak GPU utilization compared to G3 and BNS-GCN, but also experiences the most GPU idle time. This is because NeutronStar loads the entire graph to GPU memory before processing, which leads to high GPU usage but more idle time (76.6% of the time) waiting for communication. Additionally, NeutronStar uses libtorch instead of DGL to implement GNN operators, which may also contribute to the difference in peak GPU utilization. BNS-GCN performs the worst peak and average GPU utilization due to the extra sampling steps during training. Overall, G3 has the best average GPU utilization among all systems, with 2.9× higher utilization on average than BNS-GCN and NeutronStar. As a result, G3 experiences almost no GPU idle time during the time span. This is due to the balanced workload that eliminates straggler issues and the multi-level pipelining that maximizes the overlap of communication and computation.

We also observe that with a smaller initial bin size setting, G3 achieves better GPU utilization. This is because, with a finer-grained bin packing, G3 can achieve better pipelining, but at the cost of more CPU-GPU memory swapping. We leave the optimal bin size configuration that balances the tradeoff between the fine-grained pipelining and minimal memory swapping between CPU and GPU as future work.

4.6.5 Impact of Layers

In this experiment, we compare G3 with DGL when training GraphSAGE with different numbers of layers over four datasets in a 16-node cluster. We create three variants for each GNN model with 2, 3, and 4 layers. The sampling strategies for DGL are (25, 10), (25, 15, 10), and (25, 20, 15, 10) for 2-, 3-, and 4-layer models, respectively. We report the throughput speedup of G3 against DGL.

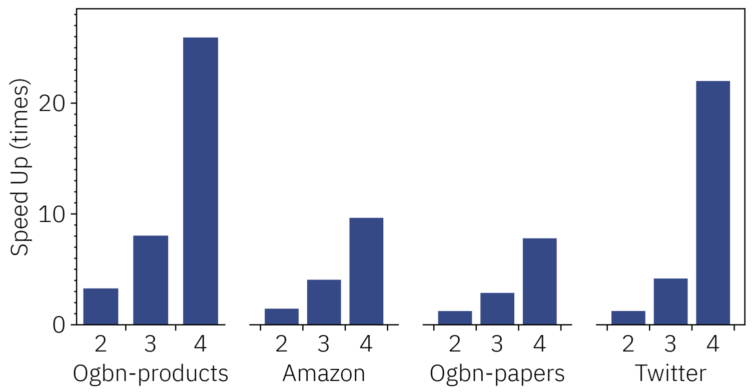


Figure 4.14: The throughput advantage of G3 over DGL increases with more GNN layers.

Figure 4.14 shows the results. We observe that the advantage of G3 over DGL increases drastically with the number of layers when training over all datasets. Specifically, G3 outperforms DGL by up to 25.9× in the 4-layer model. As elaborated in §4.1.3, DGL replicates all out-of-partition nodes for each mini-batch, whose size grows exponentially with the number of layers. Therefore, the efficiency of DGL degrades drastically with the increasing number of layers. By comparison, G3 alleviates duplicate computation and communication due to its Hybrid parallelism design and thus shows a greater advantage over DGL with deeper GNNs. Specifically, G3 accelerates the training process over the four datasets by 25.9×, 9.6×, 7.8×, and 22.0×, respectively. The significant improvement of G3 than DGL demonstrates the potential of G3 to support exploration for deeper and more complex GNN models.

4.6.6 Impact of Input Features

In this experiment, we compare G3 and DGL using Amazon dataset with varying input feature dimensions in a 16-node cluster. We train GraphSAGE on Amazon and vary the input feature dimensions from 16 to 256.

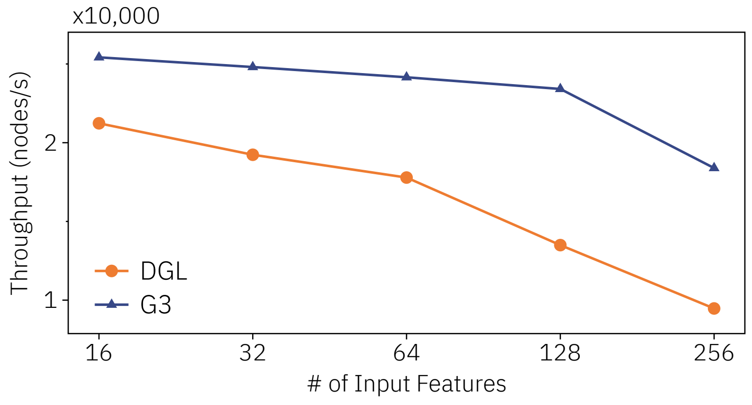


Figure 4.15: G3 achieves better throughput performance when the number of features increases.

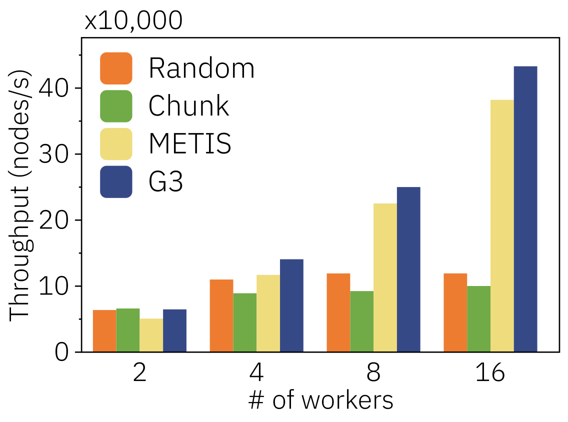
Figure 4.15 shows that G3 outperforms DGL in all dimensions and has a more graceful decrease in throughput as the feature dimension increases. DGL's heavy degradation in performance with increased feature dimension (*e.g.*, G3 outperforms DGL by 1.94× in a 256-dimensional graph) is due to the increased redundant communication, while G3 eliminates this redundant communication and uses multi-level pipeline scheduling to overlap computation and communication. We observe a slight decrease in G3's throughput when the input feature dimension increases from 128 to 256, possibly due to the large communication volume not being fully overlapped. Overall, the results demonstrate G3's versatility over a wider range of graph datasets.

4.6.7 Microbenchmarks

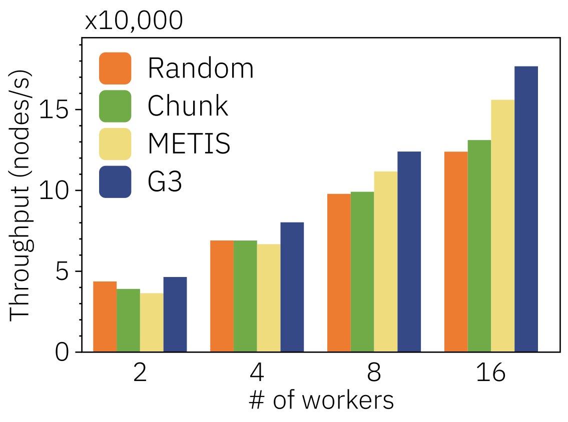
Balanced Work Partitioning. We evaluate the effectiveness of G3's locality-aware iterative partitioning by replacing it in G3 with the following partitioning methods.

- **Random**. Nodes are assigned to partitions at random.
- **Chunk-based** [198]. Nodes are split into contiguous chunks, with each chunk representing a partition.
- **METIS** [88]. A widely used graph partitioning library that generates partitions with minimum edge-cuts.

- **G3** with locality-aware iterative graph partitioning.



(a) Ogbn-products



(b) Amazon

Figure 4.16: Effectiveness of balanced partitioning.

Figure 4.16 shows the training throughput using the above algorithms when the cluster scales out. Algorithms such as random and chunk-based partitioning perform poorly as they do not take into account all edges in the graphs. METIS and balanced partitioning, which aim to minimize the number of edges across subgraphs, perform better. Balanced partitioning, in particular, by balancing communication and computation workload, improves the straggler issue and is important for multi-level pipeline scheduling (as discussed in §4.2.2). Overall, balanced partitioning leads to a maximum 4.3× improvement in speed compared to other partition algorithms.

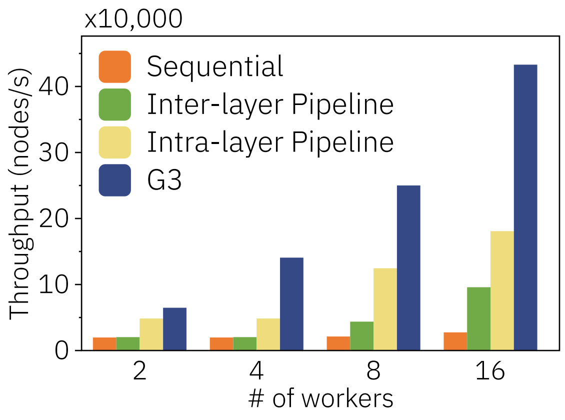
Multi-level Pipeline Scheduling. We evaluate the effectiveness of multi-level pipeline scheduling by performing experiments on the following variants of G3.

- **Sequential**. G3 with no inter- or intra-layer pipeline mechanism. All workers process computation and communication sequentially during training, which is the same manner as [64, 87]'s bin-packing performs.

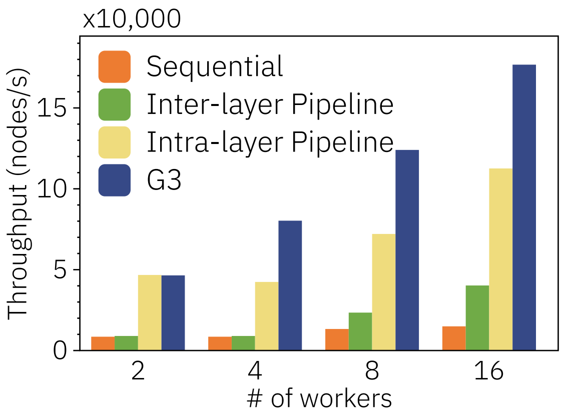
- **Inter-layer**. G3 that pipelines only across layers (§4.4.2). Each worker employs the inter-layer pipeline but naively packs nodes with consecutive node IDs into bins.

- **Intra-layer**. G3 that pipelines only within the same layer (§4.4.2). Each worker waits for all required intermediate results and employs the intra-layer pipeline with adaptive bin packing.

- **G3** with the multi-level pipeline scheduling.



(a) Ogbn-products



(b) Amazon

Figure 4.17: Effectiveness of multi-level pipeline.

Figure 4.17 shows the training throughput of various G3 variants. The sequential method has the lowest throughput. Intra-layer and inter-layer methods both improve through partial pipelining, with the intra-layer performing better, but this advantage decreases as the cluster size increases. This is because (1) the initial bin size of 11 allows for more opportunities for intra-layer pipelining, and (2) the inter-layer pipeline takes advantage of more network resources with a larger cluster size. The multi-level pipeline scheduling in G3 provides the best scalability and can achieve a maximum 11.66× improvement in performance speed.

4.7 Related Work

GNN Frameworks. DGL [158, 196] supports distributed GNN training and is widely used in both academia and industry. ROC [78] proposes online learning-based graph partitioning for a balanced workload and uses dynamic programming for efficient memory management. NeuGraph [116] bridges graph processing with DNN and support efficient multi-GPU training. However, these frameworks suffer from either layer-wise communication barriers or cross-partition neighbor replications, as elaborated in §4.1.3. PyG [58], AliGraph [197], Euler [34], and AGL [186] adopts mini-batch training with sampling, resulting in sub-optimal final accuracy [78].

GNN Training Optimization. GNNAdvisor [164] presents optimizations such as workload management and GPU memory customizations. DGCL [40] proposes a communication planning algorithm to optimize communication during training. PaGraph [107] proposes a caching policy that reduces data movement between CPU and GPU for the frequently visited nodes. These works assume that graphs are stored in one machine. BNS-GCN [151] advocates full-graph training and proposes a simple yet effective sampling method to relieve the communication and memory overhead. PipeGCN [152] defers the communication to the next iteration's computation, which introduces staleness and results in a lower theoretical convergence speed. Sancus [129] is also aware of embedding staleness and uses historical embeddings with cache to avoid communications adaptively. SALIENT [85] optimizes mini-batch training with a fast sampling approach, shared-memory parallelization, and pipelining of batch transfer. Dorylus [146] adopts pipeline parallelism [75, 124] to maximize the resource utilization in the serverless scenario. The above works may still potentially compromise their final accuracies. SAR [122] is a CPU-only GNN training system that proposes a distributed sequential rematerialization scheme for efficient memory usage. It does not support GPU training.

4.8 Conclusion

This paper tackles the scalability challenge in distributed GNN training. We propose GNN Hybrid parallelism in G3 to scale out GNN training with carefully scheduled peer-to-peer intermediate data sharing, enabling scalable GNN training on large graphs. G3 accelerates the training process by balancing workload across workers with locality-aware iterative partitioning, and overlaps communication with computation using a multi-level pipeline scheduling algorithm. We evaluate G3 with extensive experiments on large graphs, and the results demonstrate up to 2.24× improvement in training throughput and better final accuracy compared to previous systems in a 16-node cluster.

CHAPTER 5

HERMOD: A NEAR-OPTIMAL COFLOW SCHEDULER FOR LLM TRAINING

Large language models (LLMs) have demonstrated exceptional performance in tasks such as machine translation, code generation, and conversational agents [39, 55, 65, 126]. However, training LLMs is highly demanding due to their complex architectures and massive data requirements. To overcome the memory and computational limits of individual accelerators, LLM training is distributed across tens of thousands of accelerators, utilizing parallelization strategies to enable efficient coordination and scaling across the cluster [61, 81, 132].

Parallelism in LLM training involves diverse interleaved communication patterns: `AllReduce`, `ReduceScatter`, and `AllGather` for data parallelism (DP) and tensor/sequence parallelism (TP/SP) [80, 130, 136, 145]; `AlltoAll` for expert parallelism (EP) [100, 108]; and `send/recv` for pipeline parallelism (PP) [75, 124]. From a network perspective, these communication patterns can be represented as distinct coflows [41], which are managed by optimized coflow schedulers.

Coflow schedulers optimize job completion time (JCT) by reordering communication tasks to address data dependencies inherent in training, minimize computation blockages, and enhance the overlap between communication and computation. For instance, ByteScheduler [130] improves DP efficiency by overlapping DP coflows across layers, while Lina [100] lowers the cost of EP coflows by prioritizing them over DP coflows during the backward pass. These tailored strategies for DP and EP have improved the overall training performance efficiently.

Unfortunately, existing approaches [100, 130] suffer from several limitations that hinder optimal scheduling. First, these methods restrict their scheduling strategies to end-hosts, neglecting the entire cluster, which leads to contention within the network core—an issue worsened by the adoption of fine-grained load-balancing mechanisms [61, 81, 132].

Additionally, they fail to account for PP coflows, which are prevalent in both dense and sparse models and exhibit sequential dependencies, where each PP stage must await the completion of the preceding coflow. Preliminary simulations showcase that incorporating PP coflow scheduling could improve distributed LLM training performance by up to 1.29× (§5.1.3). Lastly, these strategies fail to address flow contentions within coflows, resulting in inefficient communication when the coflows are internally imbalanced.

Meanwhile, naively combining existing strategies poses significant challenges, particularly in the presence of widely adopted optimizations like ZeRO-2 and 1F1B [125, 136]. For example, integrating Lina and ByteScheduler can lead to the persistent deprioritization of DP coflows throughout training. However, this approach can be suboptimal, as DP coflows become critical during the forward pass of the next iteration, where they are required to `AllGather` layer parameters [136]. Furthermore, pipeline schedules such as 1F1B introduce an additional microbatch dimension, further complicating scheduling decisions and necessitating a more holistic, cluster-wide scheduling approach.

This leads us to a central question: *Can we design a coflow scheduler that optimally orchestrates all types of coflows at cluster level throughout the LLM training process?*

To address this question, we systematically analyze the LLM training DAG with coflows (§5.2) and identify three cases where coflow overlaps occur. A key insight emerges from this analysis: coflows can be uniquely characterized by three critical model factors—microbatch ID, coflow type, and layer ID—which collectively determine their priorities for optimal scheduling. These factors are application-specific and remain constant throughout training, making them easy to convey to the underlying transport layer and switches.

Guided by this insight, we propose Hermod, a near-optimal coflow scheduler that schedules all types of coflows optimally at cluster level throughout the LLM training process. Hermod consists of two key components: 1) a model factor-driven inter-coflow priority scheduling mechanism with strict prioritization, and 2) an optimal matching-based intra-coflow scheduling strategy tailored to each coflow type. The prioritization is determined by strictly adhering to the LLM training DAG and quantized according to the above three model factors. We rigorously analyze the potential conflicts among these factors and resolve them to develop a holistic priority strategy for coflows throughout training. For intra-coflow scheduling, the goal is to minimize coflow completion time (CCT) without degrading overall JCT. We propose strategies for each coflow type, ensuring that flows in the highest-priority coflow are transmitted at maximum rates. While this approach is straightforward for balanced DP and PP coflows, managing imbalanced EP coflows is more challenging due to varying flow sizes. We leverage framework-level metadata [145] with precise flow size information within EP coflows, and design a matching-based strategy for EP coflows, leading to optimal performance in LLM training [33, 35].

We have implemented a fully functional prototype of Hermod and evaluated it using both dense and sparse LLM models through small-scale testbed experiments and large-scale simulations. The results demonstrate that Hermod significantly improves JCT over existing schedulers for all LLM models. Specifically, in the testbed experiments, Hermod improves end-to-end training speed by up to 1.44× in a 32-GPU cluster and 1.38× with 100Gbps network bandwidth, respectively. At larger cluster scales, the benefits of Hermod are further amplified, yielding speedups of up to 1.50× in a 8192-GPU cluster and 2.07× with a 400Gbps network.

In summary, this paper makes the following contributions:

- We conduct a systematic analysis of the LLM training DAG with coflows, identifying three cases of coflow overlaps and three model factors that collectively determine coflow priority (§5.2).
- We propose a model factor-driven inter-coflow scheduling mechanism with strict prioritization, along with an optimal matching-based intra-coflow scheduling strategy tailored to each coflow type to enhance global communication performance (§5.3).
- We evaluate Hermod on both dense and sparse LLM models, demonstrating that Hermod significantly improves JCT compared to existing schedulers in both testbed and simulation experiments (§5.5).

5.1 Background: LLM Parallelism and Coflow Scheduling

Large-scale architectures and datasets in LLM training demand robust parallelisms and efficient communication mechanisms to maximize efficiency. We review these parallelisms and examine existing communication scheduling methods, while presenting our motivation.

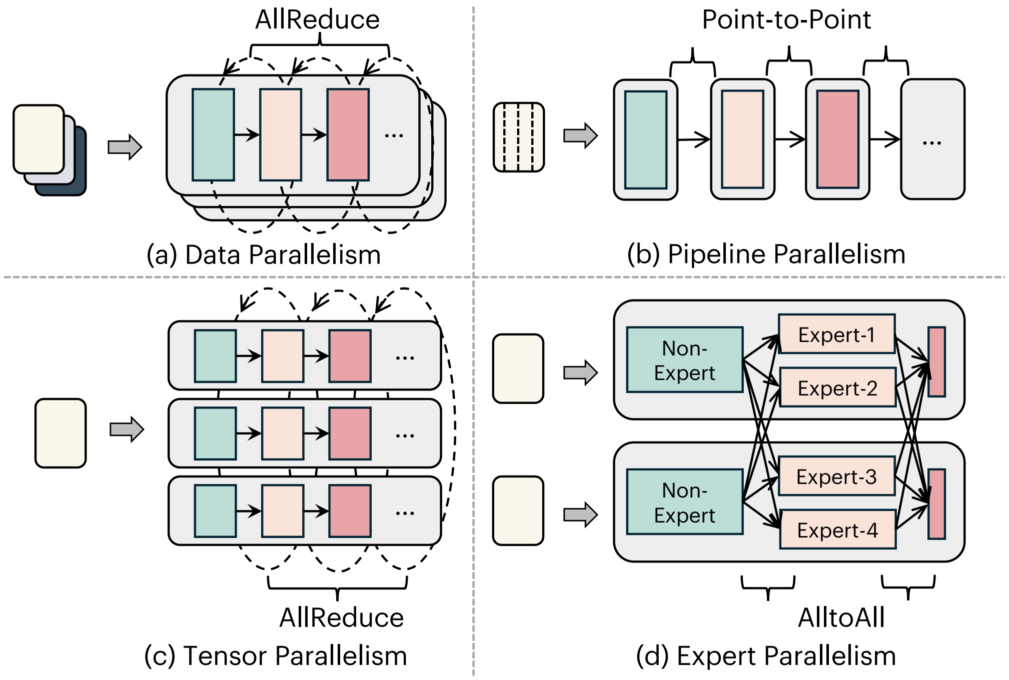


Figure 5.1: Different parallelisms and their major communication patterns in LLM training.

5.1.1 LLM Parallelism

Data parallelism (DP). DP distributes the dataset across devices, replicating both the model and optimizer states, as illustrated in Figure 5.1(a). Devices independently compute forward and backward passes, synchronizing via `AllReduce` operations [77, 130, 187]. ZeRO [136] improves memory efficiency by fine-grained sharding states and decomposing `AllReduce` with `ReduceScatter` and `AllGather`. ZeRO-2 is commonly adopted in LLM training for efficient memory management without introducing extra communication overhead [81, 195]. Ring/Tree-AllReduce [18, 80, 141] and hierarchical AllReduce [156] are widely adopted to optimize the synchronization process.

Pipeline parallelism (PP). As Figure 5.1(b) shows, PP splits the model into stages, reducing memory by storing only portions of the model per device [75,124,125]. Micro-batches are processed sequentially across stages, with intermediate activations exchanged via `P2P` communication. Scheduling methods like 1F1B and interleaved 1F1B mitigate pipeline bubbles [124, 125].

Tensor and sequence parallelism (TP & SP). TP divides layer operations among devices, requiring `AllReduce` to aggregate results [145] (Figure 5.1(c)). SP extends TP by further partitioning operators like LayerNorm and Dropout along the sequence dimension, reducing memory overhead [92]. TP and SP together optimize communication by replacing `AllReduce` with `ReduceScatter` and `AllGather`. In this paper, we use TP to refer to both parallelisms.

Expert parallelism (EP). EP enables Mixture-of-Experts (MoE) by distributing expert computations across devices [57,144]. As shown in Figure 5.1(d), expert layers use `AllToAll` communication to dynamically route tokens to the appropriate experts, with flow sizes determined by runtime token distribution [79, 100, 105, 108, 175].

Modern LLM training paradigms hybridly employ various parallelisms for efficient training [108, 125]. For example, PTD-P [125] is widely adopted to balance computation across GPUs while minimizing pipeline and communication overhead. While TP is often limited to a single node due to its high communication cost, DP, PP, and EP are typically deployed across multiple nodes [161, 175].

5.1.2 Coflows in Paralleled LLM Training

Coflow is a classical network abstraction that represents a group of flows sharing a common completion goal [46]. It provides an application-level, holistic view of communication to help avoid stragglers and gain benefits through strategic scheduling and placement [188]. This abstraction has been proven expressive in multiple communication patterns between successive computation stages of data-parallel applications, *e.g.*, MapReduce [51], and Spark [183], etc.

Coflows in LLM training. As LLM training is typically represented as a kind of data-parallel application [125, 130], we summarize the key characteristics of its communication through the coflow perspective in Table 5.1:

- **DP and TP/SP coflows** use ring- or tree-AllReduce, where flow information—including

Coflow Characteristics		**DP**	**PP**	**TP & SP**	**EP**
Coflow Type		Ring-based, Tree-based	Point-to-Point	Ring-based, Tree-based	All-to-All
On-Arrival Knowledge	Flow Number	N_{dp}	1	N_{tp}	$N_{ep}(N_{ep}-1)$
	Flow Endpoints	Known	Known	Known	Known
	Flow Size	Deterministic	Deterministic	Deterministic	On-demand
	Flow Distribution	Balanced	Balanced	Balanced	Imbalanced
Coflow Scope		Inter-rack	Inter-rack	Intra-node	Intra-rack

Table 5.1: Coflows in different parallelisms.

number, endpoints, size, and distribution—is pre-determined during compilation [31], enabling proactive scheduling and resource allocation.

- **PP coflows** involve sequential `P2P` flows between pairs of pipeline stages. Although containing only one flow per endpoint pair, these communications are still treated as coflows [46]. Flow information is known ahead of time, during compilation [31].

- **EP coflows** use `AllToAll` operators with predetermined flow numbers and endpoints, while flow sizes are dynamically calculated by the gate network based on token routing (§5.1.1). Token distribution across experts causes imbalanced flow distribution, requiring runtime scheduling adjustments for efficiency [100, 108].

The coflow scope aligns with hardware topology, varying in network bandwidth to match the communication volume of each parallelism. DP and PP coflows span racks, TP/SP are intra-node, and EP occur within a single rack [108, 175].

Prevalence of cluster-wide coflow overlap. Based on the above coflow characteristics, we next examine the prevalence of cluster-wide coflow overlap in LLM training to highlight the necessity for optimized coflow scheduling strategies. We conduct a measurement study on a production-level NVIDIA DGX SuperPOD platform [20], equipped with 128 H-series GPUs (Hopper architecture) connected via 128 400Gbps ConnectX-7 NICs in a rail-optimized topology. All servers are synchronized using Precision-Time Protocol (PTP) [143] for accurate timestamp recording, and the `overlap-grad-reduce` training parameter is enabled for faster training [22]. By measuring the start and end times of each communication operator, we calculate the overlap duration among them.

Our analysis focuses on inter-node coflows, the primary bottleneck due to the limited 100Gbps RDMA bandwidth compared to the 400GBps NVLink bandwidth within

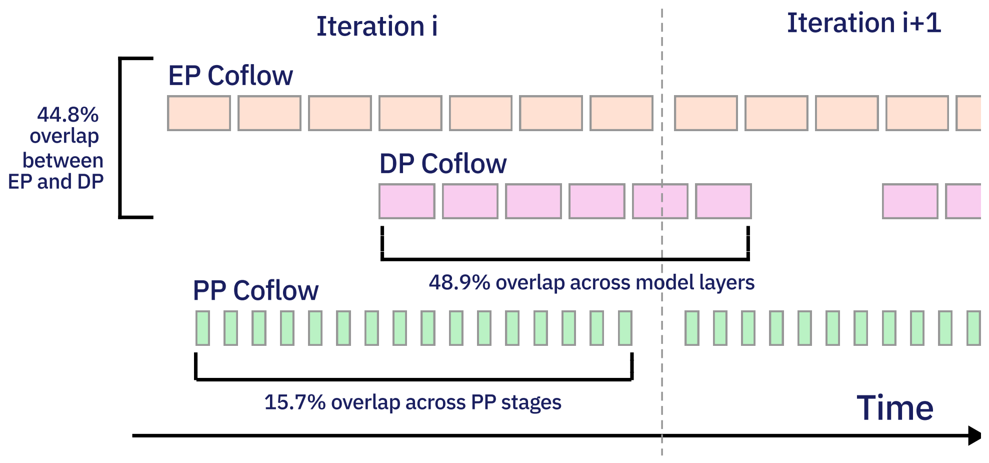

Iteration i
Iteration i+1
EP Coflow
44.8% overlap between EP and DP
DP Coflow
48.9% overlap across model layers
PP Coflow
15.7% overlap across PP stages
Time

Figure 5.2: [Production measurement] Cluster-wide coflow measurement using Mistral-8×7B reveals significant overlap between coflows.

nodes [161]. We measure overlap for DP, PP, and EP coflows. The results, shown in Figure 5.2, reveal significant cluster-level coflow overlap[1]. Specifically, EP and DP coflows overlap for 44.8% of the iteration time, DP coflows across different layers for 48.9%, and PP coflows across different stages for 15.7%. These frequent overlaps lead to contention and degraded communication performance.

5.1.3 Existing Works on Coflow Scheduling

Several works have been proposed to manage coflow overlaps [67,77,100,130,187]. Some methods [67,77,130,187] optimize DP by reordering DNN layer transmissions to enhance communication-computation overlap. For example, ByteScheduler [130] prioritizes DP coflows from front layers and uses tensor partitioning to improve overlap. Other approaches [100] focus on EP, prioritizing communication in expert layers during backward propagation to minimize `AlltoAll` time costs.

Despite their effectiveness, these approaches have several limitations in handling coflow scheduling and addressing contentions across and within coflows:

Limitation #1: Lack of cluster-wide scheduling. Existing approaches [77,100,130] primarily focus on scheduling at the endhosts, rather than managing training traffic across the entire cluster. These methods enable coflow preemption only at the point of submis-

[1]Two coflows are considered overlapped when their flows occur concurrently at the same endpoints.

sion to the communication library (*e.g.*, NCCL [7]), while the underlying transport layer remains unaware of these decisions. As a result, the lack of coordination between the intermediate scheduler and the transport layer can lead to suboptimal performance due to contention at core network components, such as aggregate and core switches.

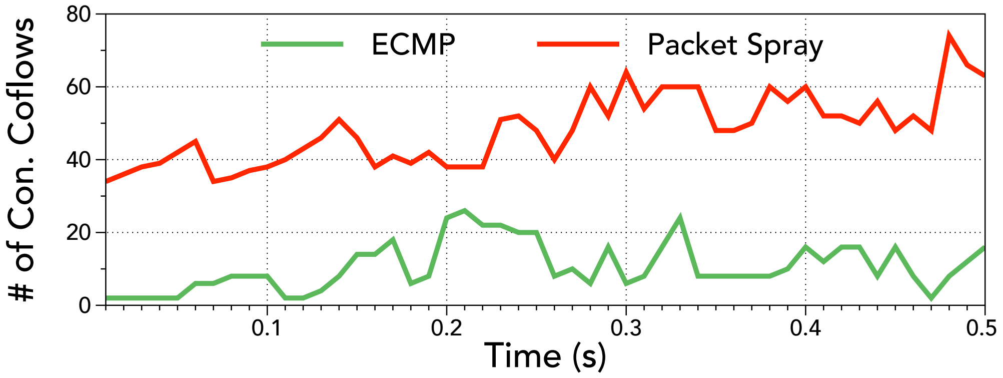


Figure 5.3: [Simulation] Number of concurrent coflows at a core switch with different load balancing mechanisms.

This contention is further intensified in state-of-the-art LLM training clusters, where fine-grained load balancing mechanisms [38, 61, 81, 132] increase the probability of flow intersections at switches. Although we lack permission to capture logs from production switches, we simulate this scenario using the simulator from [162]. We train Mixtral-8×7B [79] in a 1024-GPU cluster using two load balancing mechanisms commonly adopted in recent AI clusters: ECMP [61, 81] and packet spray [38]. By measuring coflow numbers in switch queues every 10µs, we observe that fine-grained packet spray increases concurrent coflows by an average factor of 7.4, as shown in Figure 5.3. The transport layer's lack of awareness prevents it from prioritizing high-priority coflows at switches, leading to suboptimal performance in LLM training.

Limitation #2: Incomplete consideration of all coflow types. Existing approaches [77, 100, 130] overlook certain coflow types, particularly PP coflows, which are crucial as they typically sit on the critical path of training and can block subsequent stages [75, 124].

We reuse the previous simulation configuration and assess the impact of different parallelism strategies with naive scheduling at the cluster level. The results in Table 5.2 indicate that prioritizing DP coflows degrades training performance, whereas naive prioritization of PP and EP coflows improves iteration times by 1.29× and 1.51×, respec-

Configuration	Baseline	DP-first	PP-first	EP-first
Training Speedup	-	0.81×	1.29×	1.51×

Table 5.2: [Simulation] The potential benefits of scheduling with different prioritization strategies. DP-, PP-, and EP-first are heuristic-based approaches that prioritize DP, PP, and EP traffic, respectively.

tively. Notably, EP coflows are specific to sparse MoE models, while PP coflows occur in both sparse and dense model training. These findings highlight the need for optimized scheduling across all coflow types—EP, DP, and PP—at the cluster-wide level to improve LLM training efficiency.

As LLM models scale, PP becomes increasingly vital due to its lower communication costs compared to TP. Experiments in [125] show that while TP degree caps at 8 due to bandwidth limitations [161], PP degree scales with cluster size, leading to greater communication contention. Addressing these challenges is crucial to maintain training efficiency.

Limitation #3: Unaware of flow contentions within coflows. Existing approaches [77, 100, 130] focus solely on scheduling communication operators (*i.e.*, coflows) without accounting for flow contentions within each coflow. This limitation can lead to suboptimal performance, as internal flow contentions can affect overall job completion time [33, 49]. In EP coflows, for example, imbalanced flow distribution among experts can exacerbate contentions, leading to increased training completion times. We elaborate on these contentions in §5.3.2. These approaches work primarily as intermediate schedulers, failing to incorporate fine-grained, flow-level resource allocation strategies from traditional coflow scheduling [33, 47, 49].

5.1.4 Motivation and Challenge

The above limitations motivate us of a coflow scheduler that can optimally, if possible, consider all coflow types, and be implemented at the cluster-wide level. Specifically, the coflow scheduler should be:

- **Comprehensive:** It should account for all coflow types, including DP, PP, and EP, to optimize training performance across different parallelism strategies.

- **Fine-grained:** It should address flow contentions within coflows and optimize flow-level resource allocation to minimize JCT.
- **Cluster-wise deployable:** It should be aware of coflow overlap throughout training and be deployable across the entire cluster, managing coflow scheduling at both endhosts and network core components to reduce contention.
- **Optimal:** The scheduler should aim to optimally minimize the completion time of the entire training.

Challenge: Non-trivial efforts in designing a comprehensive policy. To design a comprehensive coflow scheduler which optimally manages all coflow types, a naive approach is to combine existing strategies for each coflow type for a comprehensive policy. Unfortunately, this approach can lead to inefficiencies, especially with the prevalent optimizations such ZeRO-2 and 1F1B [81, 125]. For instance, combining ByteScheduler and Lina results in EP coflows always prioritize over DP coflows, while DP coflows are prioritized according to their associated layers. This could lead to suboptimal performance. When crossing iterations, DP coflows are on the critical path and be needed for `AllGather` operations to gather layer parameters in ZeRO-2 [136]. This results in higher priority of DP coflows over others. Besides, the significance between EP coflows of the front layers and DP coflows of the latter layers is left uninvestigated. Furthermore, pipeline scheduling strategies such as 1F1B introduce an additional microbatch dimension to cluster scheduling, further complicating scheduling decisions—an aspect not yet addressed by existing schedulers.

In summary, existing approaches either require complex combinations to be adaptive with prevalent optimizations or neglect some coflow types with potential performance degradation. To address these challenges, a systematic analysis of coflow interactions regarding the LLM training directed acyclic graph (DAG) is in its urgent need.

5.2 LLM Training DAG with Coflows

This section critically revisits the LLM training DAG with a focus on coflows, analyzing the overlap among EP, PP, and DP coflows at different training phases. By identifying inefficiencies and proposing a more efficient scheduling policy, we aim to enhance resource utilization and improve overall training performance.

5.2.1 Problem Statement

We consider the common industry scenario where a dedicated cluster runs a single LLM training job [81]. The network topology, worker placement, and parallelization strategy are predefined before execution, resulting in a fixed LLM training DAG throughout training. This DAG determines coflow dependencies, encompassing all necessary data movements. Formally, it translates into a dependency graph where each coflow represents a set of flows with predefined characteristics, as detailed in Table 5.1.

The objective is to define a cluster-wide optimal scheduling policy that efficiently allocates network resources among coflows to minimize job completion time (JCT). Specifically, the policy should:

- Adhering to the LLM training DAG: It should strictly respect coflow dependencies as determined by the DAG, incorporating conventional mechanisms like ZeRO-2 [136], 1F1B, and its variants [125].
- Minimize global JCT: The policy should minimize JCT of completing the entire coflows. That says, it should avoid coflow overlaps as much as possible, while maximizing communication-computation overlap.
- Cluster-wise deployable: It should be practically deployable across the entire cluster.

5.2.2 Systematic Analysis of LLM Training DAG

We depict a conventional LLM training DAG in Figure 5.4, which trains an MoE model consisting of multiple MoE layers with ZeRO-2 and 1F1B optimizations. The training process follows an iterative manner, with phases of forward propagation (FP) and backward propagation (BP) in each iteration. Within each iteration, EP coflows occur twice per MoE layer group for token dispatching and gathering. PP coflows then facilitate intermediate

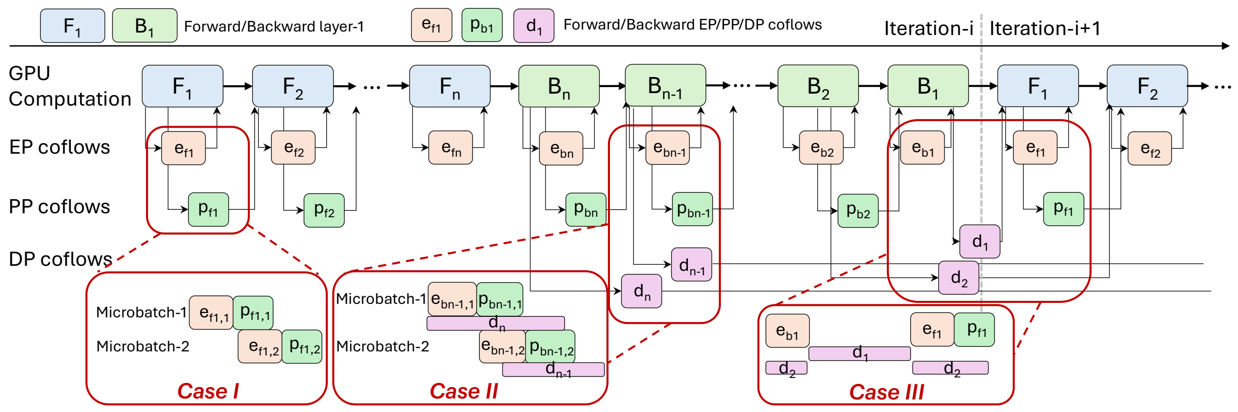


Figure 5.4: Computation and coflows in LLM training.

result transmission between layers[2]. These operations are repeated across layers and microbatches. During BP in the current iteration and FP in the next, DP coflows execute as `ReduceScatter` and `AllGather` operations for gradient synchronization and parameter updates[3] [81, 130, 136, 187].

From the above DAG, we identify in total three cases where coflows may overlap, as highlighted by the red rectangles in Figure 5.4:

- *Case I: EP and PP coflows among different microbatches.* After processing microbatch-1, the worker invokes the asynchronous operator `isend` to transmit intermediate results to the next PP stage, generating PP coflows. Simultaneously, it processes microbatch-2, performing attention and gate computations before generating EP coflows for token dispatching. Our profiling results in §5.1.2 show average attention and gate computation times of 2.4ms and 0.6ms, respectively. Given that PP coflows complete in ∼4ms on average, there is potential for PP-EP coflow overlap, even at the end host (as we have identified in §5.1.2). This overlap becomes more likely at the cluster scale, where PP coflows compete for bandwidth with other coflows, prolonging their completion while attention and gate computation times remain constant.

- *Case II: EP, PP, and DP coflows among different microbatches during backward propagation.*

[2]For ease of presentation, we configure each layer to represent a pipeline stage, thus showing PP coflows between layers. In practice, PP coflows occur between stages, which may span multiple layers, depending on the PP degree.

[3]We use the same layer-wise description as in [130, 187], whereas in PyTorch, DP coflows are grouped into buckets spanning multiple layers while maintaining a sequential execution order [127].

Building on *Case I*, DP coflows emerge during backward propagation, overlapping with BP computation. In Megatron-LM, DP coflows are invoked only after the last microbatch completes its gradient calculation for the layer and initiates gradient synchronization. The concurrent nature of DP coflows with computation allows DP to be transmitted simultaneously with other coflows [100].

- *Case III: EP, PP, and DP coflows among different layers during cross-iteration phases.* In this case, DP coflows from latter layers may overlap with EP and PP coflows from front layers during the next iteration's forward propagation. This scenario becomes particularly common when enforcing priority scheduling with tensor partition[4] [130], as the figure depicted. Unlike *Case II*, DP coflows in this case are critical, as they gather the parameters required to initiate forward propagation in the next iteration [136].

We make a key observation based on the identified cases: cluster-wide coflows throughout training can be uniquely represented by three model factors: microbatch ID (MID), coflow type (CType), and layer ID (LID). Specifically, MID and CType collectively identify coflows in *Case I* and *Case II*, while CType and LID are for coflows in *Case III*.

Notably, these model factors showcase the following characteristics, making them well-suited for optimal coflow scheduling:

- *Pre-determined.* The factors are pre-determined in applications and remain constant throughout training.
- *Simple.* The factors are simple integers that can be easily transformed and transmitted to transport and switches.

5.3 Hermod

In this section, we present the design of Hermod. We leverage the three model factors and propose a near-optimal inter-coflow priority policy for scheduling all coflow types

[4]At the time of writing, Megatron-LM [145] does not yet support priority scheduling. We think this may be partly due to the use of clip-by-norm [133] in LLM training, which can create barriers between iterations. However, clip-by-value [133] could eliminate these barriers algorithmically.

(§5.3.1). Building on this strict priority, we further introduce an optimal intra-coflow scheduling policy to avoid fine-grained flow contentions (§5.3.2).

5.3.1 Model factor-driven Inter-Coflow Priority Scheduling

Our goal is to design an optimal inter-coflow scheduling policy to address the identified three cases leveraging three model factors. The primary goal of the coflow scheduling is to minimize the JCT of the training DAG in Figure 5.4, *i.e.*, reducing the time spent on the critical path (the longest path in DAG where computation/communication *must* be executed) as much as possible and hiding the communication behind computation as much as possible [67, 130].

Model factor-driven priority assignment. We make a key observation based on the above goal: by considering *how many on-critical-path computations can be blocked by the coflows* and *when the transmitted data will be consumed*, we can define a *near-optimal* priority scheduling policy for all cases. Again, we use the example in Figure 5.4 for illustration.

- For *Case I*, we must address the contention between $p_{f_{1,1}}$ and $e_{f_{1,2}}$ coflows. We observe that allocating more network resources to $e_{f_{1,2}}$ while delaying $p_{f_{1,1}}$ may accelerate the progress of microbatch-2 in layer-1. However, this progress is stalled until microbatch-1 completes in layer-2, which is delayed due to $p_{f_{1,1}}$, ultimately increasing the overall model processing time. The above phenomenon is due to the sequential nature of microbatch processing, where earlier microbatches block more computations on the critical path. Hence, it is essential to prioritize the coflows of earlier microbatches over the latter ones. This prioritization can be generalized across all coflows within the same iteration.

- For *Case II*, while the priority between EP and PP coflows is determined by the microbatch ID, their relationship with DP coflows must also be considered. Since EP and PP coflows are consumed in the subsequent layer, whereas DP coflows are used in the next iteration, EP and PP coflows should be prioritized over DP coflows. This minimizes blocking time while maximizing the overlap of DP coflows with computation.

- For *Case III*, we analyze the priority between EP/PP coflows from the next iteration and

DP coflows from the previous iteration. Given that data from former layers is consumed sooner than that from latter layers, regardless of coflow type, coflows from earlier layers should be prioritized.

	Case I	***Case II***	***Case III***
Priority factor	MicrobatchID (Lower is better)	Coflow type (EP & PP > DP)	LayerID (Lower is better)

Table 5.3: Priority factors used to assign priority for three cases. The content in the parentheses indicates priority order.

Formally, the above priority decisions are quantified into three model factors, establishing a strict priority order among coflows, as listed in Table 5.3. Specifically, MID prioritizes earlier microbatches to prevent critical path delays. CType assigns higher priority to EP and PP coflows to maximize their overlap with DP coflows, with EP and PP coflows further ordered by MID. LID prioritizes coflows from former layers to enhance overlap opportunities for latter layers.

Conflicts	MID+CType	MID+LID	CType+LID
Priority factor	/	MID*	/

Table 5.4: Potential conflicts for different combinations of factors and the corresponding priority factor. / indicates impossible scenario in training, and MID$*$ means that MID is prioritized mostly, unless in certain rare cases (*e.g.*, DP coflows involve).

Handling factor conflicts. While comprehensive the above analysis is, an unresolved question remains: how should coflow priorities be determined when the three model factors conflict? To address this, we rigorously examine all potential conflicts using domain knowledge of LLM training and summarize the resolution in Table 5.4. Specifically,

- *Conflict between MID and CType*: A potential conflict, such as a DP coflow from an earlier microbatch versus an EP coflow from a later microbatch, does not arise during training. As stated in §5.2.2, DP coflows are invoked only after the last microbatch has completed its computation. Consequently, DP coflows always carry the highest MID value, inherently avoiding this conflict.

- *Conflict between MID and LID*: A scenario such as a layer-2 coflow from microbatch-1 versus a layer-1 coflow from microbatch-2 can occur in two representative cases. The first case involves EP and PP coflows with different MIDs and LIDs coexisting during FP. Here, we prioritize MID over LID to *minimize the on-critical-path computations' blockages*, ensuring that earlier microbatches are processed with minimal delays. The second case arises during cross-iteration when a layer-1 EP/PP coflow from microbatch-2 overlaps with a layer-2 DP coflow. Since DP coflows always hold the highest MID (*e.g.*, microbatch-4), prioritizing them prevents blocking microbatch-1 in the subsequent iteration (as DP coflows correspond to "microbatch-1" in the next iteration [136]). However, this scenario is exceedingly rare, as DP coflows typically complete before microbatch-2 begins processing in the next iteration (our measurements indicate that each stage persists for approximately $\sim$100+ ms). Consequently, the general resolution for MID+LID conflicts is: if DP coflows are present, LID takes precedence over MID; otherwise, MID is prioritized. Given the rarity of DP-involved conflicts, the default prioritization strategy remains MID over LID.

- *Conflict between CType and LID*: A case such as a PP coflow from layer-2 versus a DP coflow from layer-1 is infeasible during training. During backward propagation, DP coflow from layer-1 is never generated before PP coflow from layer-2, as DP coflows are always the final coflows produced for a layer. During cross-iteration, the DP coflow from layer-1 *must* be generated before the PP coflow from layer-2 due to dependencies on the corresponding layer in the next iteration. Thus, this conflict does not occur.

- *Conflict among all three factors*: Such a scenario is impossible during training, as demonstrated by the combined analyses above.

In summary, the priority assignment for the three factors follows the order of MID > CType > LID. Notably, the above priority assignment with the three model factors *ultimately* establishes *a strict order* among all coflows in the cluster throughout training.

Priority scheduling in interleaved 1F1B. While our prior analysis focuses on the conventional DAG structure of LLM training with the widely adopted 1F1B pipeline schedule [124, 125], our priority scheduling policy can be generalized to other DAG variants, such as interleaved 1F1B [125]. In contrast to the conventional method, interleaved 1F1B distributes model layers across workers in an interleaved manner to minimize pipeline stage bubble time, leading to interleaved transmissions of PP coflows. Our systematic analysis indicates that under this scheduling variant, PP coflows are no longer on the critical path and can be efficiently overlapped with computation, whereas the prioritization properties of other coflows remain unchanged. Accordingly, only the priority order in *Case II* in Table 5.3 requires adjustment: EP > PP > DP (where EP coflows remain on the critical path, PP coflows are consumed within the same iteration, and DP coflows are cross-iteration). All other priority assignments in Table 5.3 remain unchanged. Fortunately, the order of the three factors remain unchanged, as the conflict cases are common. Furthermore, the overall priority scheduling policy remains intact, as the nature of conflict cases is consistent across scheduling variants. We verify the efficiency of Hermod with interleaved 1F1B in our evaluation (§5.5.2).

5.3.2 Matching-based Optimal Intra-Coflow Scheduling

Given the strict priority of coflows, the next step is to allocate network resources optimally among flows within each coflow to eliminate contention and minimize coflow completion time (CCT).

Line-rate Transmission is Desirable

Following previous coflow scheduling studies [33, 47, 49], the key to network resource allocation within each coflow is designing optimal rate allocation strategy. In this work, we systematically analyze the optimal rate allocation strategies tailored to different coflow types. Specifically, we examine DP and PP coflows, which exhibit balanced flow distributions, and EP coflows, which are characterized by imbalanced distributions, as summarized in Table 5.1.

Balanced coflows. As Figure 5.5 shows, balanced coflows exhibit fixed optimal rate allocation strategies that minimize their CCTs, with each flow transmitting at line rate regardless of the specific strategy employed. For example, in ring-based DP coflows, the optimal strategy is to form a ring of flows, each transmitting at line-rate to achieve a bandwidth-optimal solution [128]. Similarly, PP coflows benefit from line-rate transmis-

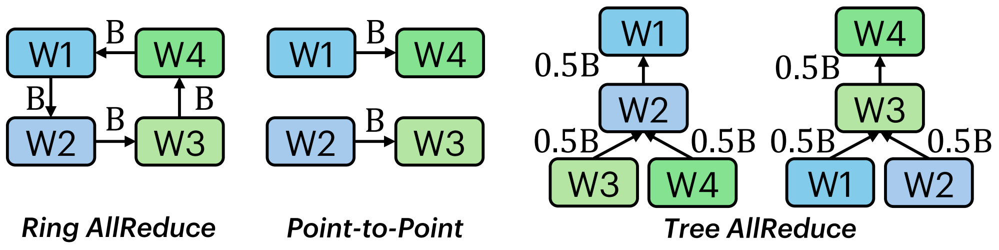


Figure 5.5: Line-rate transmission is desirable for coflows with balanced flow sizes, *e.g.*, DP & PP coflows.

sion, as each coflow consists of a single flow. Tree-based DP coflows differ slightly; since they require two transmission trees, the available bandwidth is evenly divided, with each tree allocated half of the line-rate bandwidth [18].

Imbalanced Coflows. For coflows with imbalanced flow sizes, *i.e.*, EP coflows, the optimal solution to eliminate contentions among flows for minimal CCT is to ensure that the largest flow always transmits at line-rate, as performed by previous works [33, 47, 49]. This can be achieved through two approaches: (i) the *bandwidth-sharing* strategy, which allocates bandwidth in proportion to flow size [47, 49], and (ii) the *line-rate transmission* strategy, which avoids explicit rate allocation and instead prioritizes completing all flows as soon as possible while adhering to the coflow order [33].

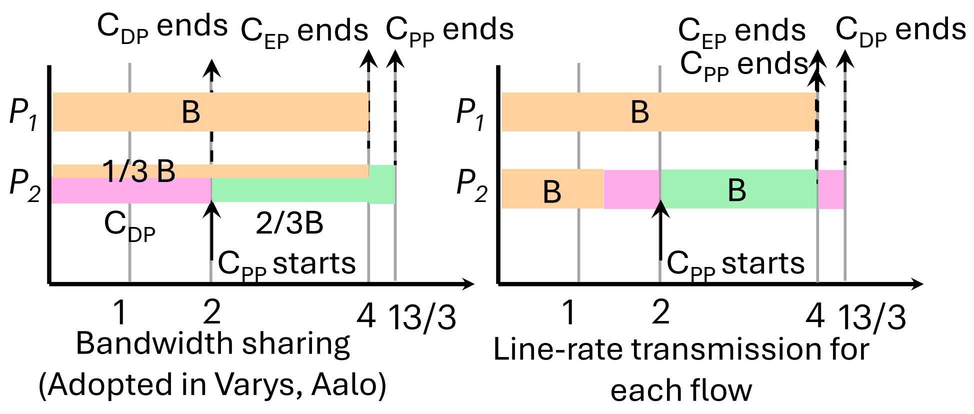


Figure 5.6: Line-rate transmission is desirable for coflows with imbalanced flow sizes to avoid degrading the global JCT.

In the context of LLM training, we find the line-rate transmission strategy to be more suitable. As shown in Figure 5.6, consider a scenario where EP (highest priority) and DP (lowest priority) coflows start concurrently, and a PP coflow (second priority) begins at

$t = 2$. This scenario is common during backward propagation when a worker starts EP coflows alongside DP coflows, and its neighbor begins transmitting intermediate results for the next microbatch. Under bandwidth sharing, all coflows complete at $\frac{13}{3}$, but the PP coflow finishes later, potentially delaying the global JCT due to its placement on the critical path and its higher priority over DP coflows. In contrast, with line-rate transmission, the PP coflow finishes simultaneously with the EP coflow, with DP coflows transmitting underneath, which prevents any delay in global JCT.

Theoretical support of line-rate transmission. Line-rate transmission is desirable over bandwidth sharing in LLM training context because allocating bandwidth proportionally does not improve the completion time of the highest-priority coflow [33] and can instead negatively impact global JCT, as our example illustrates. In contrast, for an *ordered* coflow, it is better to allocate as much bandwidth as possible to the highest-priority coflow, ensuring all its flows are transmitted quickly and minimizing flow completion times. This principle has been theoretically proved by prior studies: Sincronia demonstrates that the optimal strategy for coflows with strict priority does not require explicit rate allocation and achieves a solution within $2\times$ of the optimal. Similarly, pFabric [35] asserts that, once flow priorities are determined (as in our scenario), the primary role of rate control [167] is to enforce these priorities while preventing network congestion.

Matching, Not Rate Allocation

Building on the above analysis, we conclude that the optimal strategy for transmitting imbalanced EP coflows is to transmit all flows at line rate while avoiding network congestion. Consequently, we propose a matching-based approach for EP coflows to schedule all flows without incuring incast. The basic idea is to divide the transmission phase into multiple sub-phases, with each sub-phase forming a matching within the EP group and transmitting a portion of the flows accordingly.

Challenge and opportunity. A key challenge arises: how can we determine the flow sizes of EP coflows a priori to maximize the matching scope within the EP group at each sub-phase? The flow sizes for each sender-receiver pair are dynamic and can only be known after the gate network computation (Table 5.1). Without this prior knowledge, efficiently assigning flows to sub-phases while preventing incast remains a significant challenge.

Fortunately, we observe that while the flow sizes are dynamic, the number of tokens to be dispatched can be obtained from the training framework's metadata [19]. Specifically, to facilitate subsequent `AlltoAll` communication, frameworks such as Megatron-LM preallocate appropriately sized receiving buffers on GPUs for each sender-receiver pair. Although each worker requires only a subset of the transmission matrix—corresponding to its own send and receive volumes—the framework simplifies the process by performing a full `AllGather` operation across the EP group. The overhead of this operation is negligible compared to iteration time, *i.e.*, ∼0.4ms vs. ∼5s based on our logs (§5.1.2). While we observed this metadata in Megatron-LM [19], a widely used LLM training framework, this practice is necessary in other frameworks as well, as GPU buffer reservation is required prior to transmission. In summary, by leveraging this transmission matrix, we can transform flow sizes from unknown to known and design a matching strategy to transmit all flows in the EP coflows without incurring incast.

Algorithm 3: Finding Matching Sequences and Weights

```
Input: Transmission matrix $T \in \mathbb{R}^{n \times n}$
Output: Matching sequences $M_1, M_2, \ldots, M_k$ and corresponding weights
        $w_1, w_2, \ldots, w_k$
1  $matchings \leftarrow \{\}$;
2  $weights \leftarrow \{\}$;
3  $T \leftarrow \text{array}(T)$;
4  while $\exists T_{ij} > 0$ do
5      $row\_ind, col\_ind \leftarrow \text{linear\_sum\_assignment}(-T)$ ;        // Hungarian
        algorithm
6      $M \leftarrow \text{zeros\_like}(T)$;
7      for $i \in row\_ind$ and $j \in col\_ind$ do
8          $M[i, j] \leftarrow 1$;
9      $matching\_values \leftarrow T[row\_ind, col\_ind]$;
10     $matching\_weight \leftarrow \min(\{T_{ij} \mid T_{ij} > 0\})$;
11     $weights \leftarrow weights \cup \{matching\_weight\}$;
12     $matchings \leftarrow matchings \cup \{M\}$;
13     $T \leftarrow \max(T - matching\_weight \times M, 0)$;
14 return $matchings, weights$;
```

Matching algorithm. We present our solution for finding matching sequences and weights for EP coflows in Algorithm 3. Our approach leverages the classical Hungarian algorithm [94] to iteratively find the maximum matching in the residual transmission matrix, greedily selecting as many flows as possible for transmission (Line 5). Specifically, the algorithm iteratively computes the maximum matching, subtracts the matched flows from the transmission matrix, and repeats this process until all flows are matched (Line 5-13). The result is a sequence of matching flows and their corresponding weights. Using this sequence, the EP coflow transmission phase is divided into multiple sub-phases, with each sub-phase involving transmitting flow portions proportional to the associated weights by the workers in each matching. Since the input matrix is identical across all workers in the EP group, the derived matching sequences and weights are also identical, eliminating the need for synchronization among workers. Furthermore, by greedily selecting the maximum matching in the residual matrix, the algorithm guarantees both the optimal CCT for EP coflows and line-rate transmission for the highest-priority coflows.

5.4 Implementation

We have implemented a fully functional Hermod prototype in C++ and Python, integrating it with Megatron-LM, PyTorch, and NCCL. The model factor-driven inter-coflow scheduling is enforced at both endhosts and switches, while the matching-based intra-coflow scheduling is handled at endhosts in `LibTorch`. Specifically, on the endhost side, we deploy Hermod scheduling policy on all devices as a single-threaded process. Each scheduler instance maintains a priority queue for coflows, with priorities determined by three factors obtained from Megatron-LM. To enforce scheduling at switches, we implemented a new priority-aware networking layer to replace the default IB verbs. This layer allows for configuring traffic priorities for collective operations at runtime, with the same efficiency as IB verbs. At the switch side, we deploy policies to manage traffic with different priorities, which are marked by DSCP values in the IP headers. To simplify encoding, we compress the priority value from the three factors into a 6-bit DSCP: the top three bits for microbatchID, one bit for coflow type, and two bits for layerID, following the strategy outlined in §5.3.1. Additionally, we implemented an asynchronous `AlltoAll` operator using our matching-based algorithm in PyTorch to enforce the proposed transmission logic for EP coflows.

5.5 Evaluation

In this section, we evaluate Hermod's performance by answering the following questions:

- **How does Hermod perform in end-to-end LLM training?** Testbed experiments (§5.5.2) over three representative LLM models show that Hermod improves the training speed by up to 1.44× over baselines in a 32-GPU cluster, and 1.38× speedup with varying network bandwidths.
- **How does Hermod perform in complex interleaved 1F1B scenario and in terms of its components?** Testbed experiments (§5.5.2) show that Hermod improves training speed by 1.18× in interleaved 1F1B scenario. Additionally, its inter- and intra-coflow scheduling contributes to 1.24× and ∼1.11× speedup, respectively.
- **How does Hermod scale with increasing cluster size?** Simulation experiments (§5.5.3) show that Hermod scales effectively for large-scale model training, outperforming baselines by up to 1.50× in a 8192-GPU cluster, and 2.07× speedup with varying network bandwidths.

5.5.1 Methodology

Our evaluation comprises both small-scale testbed experiments for moderate-size models and large-scale simulations for large-size models.

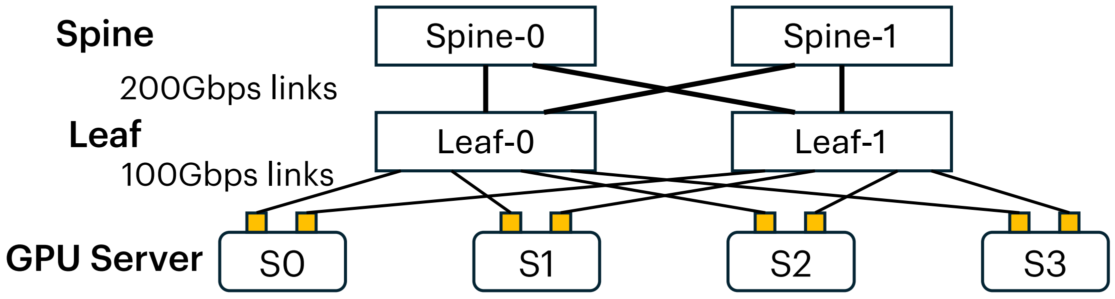


Figure 5.7: Testbed topology.

Testbed setup. The testbed depicted in Figure 5.7 consists of four A100 servers, each equipped with eight NVIDIA A100 GPUs, 96 CPU cores (3.0GHz Intel Xeon Gold 6248R), 503GB RAM, and two ConnectX-6 100Gbps NICs. The servers are connected via four NVIDIA SN3700 Ethernet switches, forming a full-bisection rail-optimized network [14]. Within each server, the GPUs are interconnected by NVLink. All servers run Ubuntu 22.04 with CUDA v12.4, PyTorch v2.3.1, and NCCL v2.20.5.

Simulation setup. Our simulator is built on top of FlexFlow's simulator [15] and the packet-level simulator htsim [16], following a similar methodology as [162]. We extend the FlexFlow simulator to generate DAGs that describe both computation and communication tasks across all coflow types. The DAGs are then fed into htsim to simulate packet communication between GPUs. By default, we simulate a 128-server cluster, with each server hosting eight GPUs, connected via a full-bisection fat-tree network with 100Gbps links and a 1μs link propagation delay.

Configuration	Models	# of layers	# of experts	PP degree	TP degree	EP degree	Sequence length	Micro-batch size
Testbed	Qwen-MoE-12L	12	64	4	1	8	4096	4
	Mixtral-8×7B	32	8	2	1	8	4096	4
	GPT-3 13B	40	/	4	4	/	2048	4
Simulation	Qwen-MoE-24L	24	64	8	8	32	4096	8
	Mixtral-8×22B	32	8	8	8	8	4096	8
	GPT-3 175B	96	/	8	8	/	2048	8

Table 5.5: Models and default parallelization strategies used in our testbed and simulation.

Models and parallelization strategies. We list both the models and the default parallelization strategies used in our testbed and simulation in Table 5.5. Specifically, we use three representative model types: two sparse MoE models—Qwen1.5-MoE [175] and Mixtral-MoE [79]—and a dense model, GPT-3 [39]. These models are used with moderate sizes that fit within our testbed cluster, and we scale up the model sizes for large-scale cluster simulations. We vary the DP degree to assess system scalability.

Baselines. We compare our system with the vanilla Megatron-LM [145] and the state-of-the-art communication scheduler Lina [100]. As Lina is not open-sourced, we implemented its key design in Megatron-LM. We do not compare with conventional coflow schedulers, as they are not designed for LLM training, nor with ByteScheduler [130], as its open-source code [12] is outdated and incompatible with the latest versions of NCCL and PyTorch.

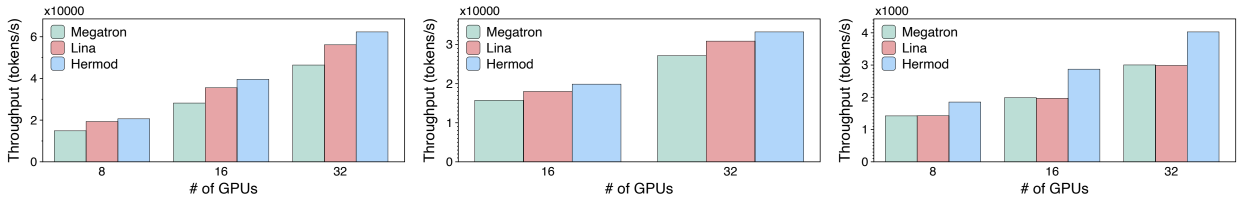
Metrics. We use # of tokens processed per second and iteration time as metrics to evaluate training performance.

5.5.2 Testbed Experiments

We conduct testbed experiments to evaluate Hermod's performance in end-to-end LLM training and to analyze the effectiveness of its individual components.

End-to-end performance

Scalability. We begin by evaluating the scalability of Hermod. We use the base parallelization strategies of (t,p,d,e) as (1,2,1,4), (1,2,1,8), and (4,2,1,1) for Qwen-MoE-12L, Mixtral-8×7B, and GPT-3 13B, respectively, and scale the cluster size from 8 to 32 GPUs by increasing the DP degree. For Mixtral-8×7B, we scale only from 16 to 32 GPUs, as the 8-server setting encountered out-of-memory issues.

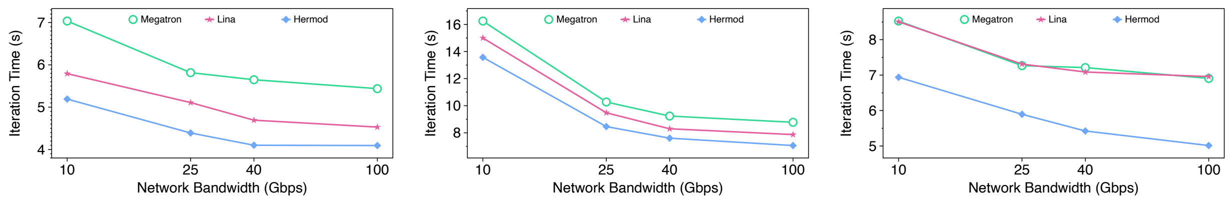


(a) Qwen-MoE-12L (b) Mixtral-8×7B (c) GPT-3 13B

Figure 5.8: [Testbed] End-to-end training throughput with cluster size scaling up.

The results are presented in Figure 5.8. Megatron exhibits poor throughput performance across all models as cluster size increases, as it does not account for the relationship between coflows. For example, when training GPT-3 13B and scaling the cluster from 8 to 16 GPUs, we observe only a 1.39× speedup, despite a 2× increase in cluster size. Lina performs better for MoE models but still underperforms for GPT-3 13B. This is because Lina focuses solely on optimizing EP coflows, which provides no benefit for dense models. In contrast, Hermod outperforms both baselines in all models, particularly with GPT-3 13B, where it achieves up to 1.44× higher throughput. The improvement is attributed to Hermod's cluster-wide scheduling across all coflow types and line-rate transmission for highest-priority coflows.

Different network bandwidths. We then evaluate the improvement Hermod can achieve compared to the baselines under typical network bandwidth settings, ranging from 10Gbps to 100Gbps. We use the parallelization strategies in Table 5.5 for each model in a 32-GPU cluster. The available bandwidth is controlled by limiting the speed of Mellanox NICs using the `mlnx_qos` command.



(a) Qwen-MoE-12L (b) Mixtral-8×7B (c) GPT-3 13B

Figure 5.9: [Testbed] End-to-end training iteration time with different network bandwidth.

The results are shown in Figure 5.9. We observe that Hermod consistently outperforms the baselines across all models under varying network bandwidths. For instance, when training Qwen-MoE-12L, both Hermod and Lina achieve higher throughput than Megatron, primarily due to the large volume of EP coflows (>80% of total communication). Hermod outperforms Lina by 1.2×, thanks to its cluster-wide prioritization of earlier microbatch coflows and PP coflows over DP coflows. When training GPT-3 13B with 100Gbps bandwidth, Hermod performs significantly better than the baselines, achieving up to 1.38× higher throughput. This improvement is attributed to the absence of heavy EP coflows, which amplifies the interference between PP and DP coflows. As a result, Hermod fully utilizes the network bandwidth through inter-coflow scheduling with strict priority and line-rate transmission, optimizing the global JCT.

Hermod Deep Dive

We conduct deep dive experiments to evaluate both the efficiency of Hermod in interleaved 1F1B scenario and the effectiveness of its components.

Interleaved 1F1B scenario. We evaluate the effectiveness of Hermod in the complex interleaved 1F1B scenario using the priority illustrated in §5.3.1 and the default parallelization strategies in the 32-GPU cluster. The results in Figure 5.10 show similar performance

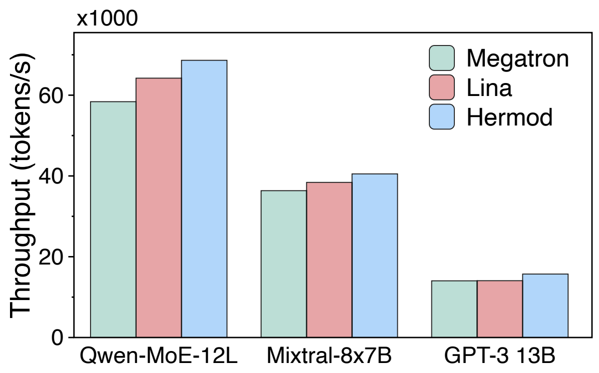


Figure 5.10: [Testbed] Effectiveness of Hermod in interleaved 1F1B scenario.

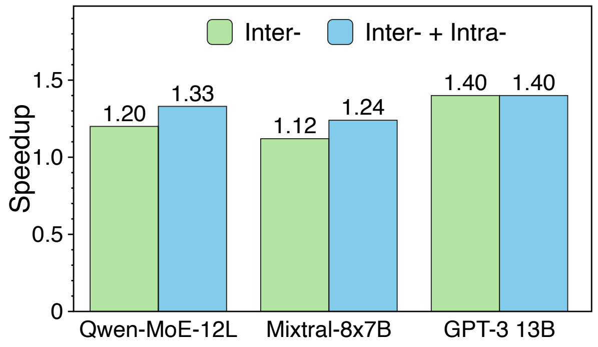


Figure 5.11: [Testbed] Effectiveness of Hermod's components.

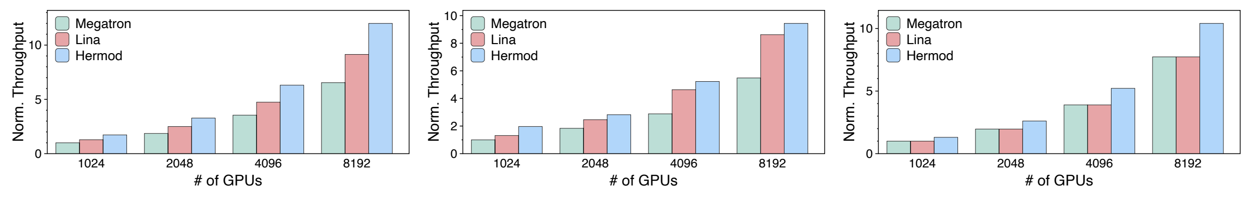
improvements as §5.5.2: Hermod outperforms Megatron and Lina by 1.18× and 1.12×, respectively, demonstrating that Hermod enhances training performance even in the intricate interleaved scenario.

Effectiveness of components. We further investigate the individual contributions of Hermod's components, specifically inter- and intra-coflow scheduling, in the 32-GPU cluster with default parallelisms as well. The results in Figure 5.11 reveal that inter-coflow scheduling is the primary contributor to performance improvement. With inter-coflow only, the speedups for training Qwen-MoE-12L, Mixtral-8×7B, and GPT-3 13B already reach 1.20×, 1.12×, and 1.40×, respectively, as it significantly reduces interference between coflows at the cluster level. Intra-coflow scheduling provides an incremental but effective improvement for MoE models, while offering no benefit to GPT-3 13B, as the coflows in dense models are balanced and transmit flows with fixed strategies (§5.3.2). With both components enabled, Hermod achieves speedups of 1.33×, 1.24×, and 1.40×.

5.5.3 Large-scale Simulation Experiments

We further conduct simulations to evaluate Hermod when training large-size models in a large-scale cluster.

Scalability. We evaluate the scalability of Hermod by scaling the cluster size from 1024 to 8192 GPUs with increasing DP degree. The results in Figure 5.12 show consistent results as the testbed experiments (§5.5.2), Megatron exhibits poor throughput performance

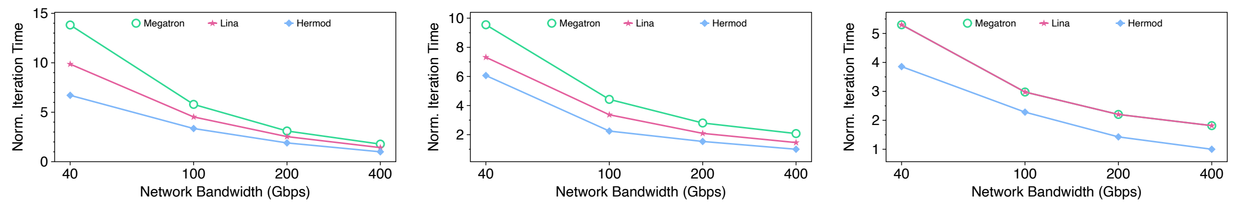


(a) Qwen-MoE-24L (b) Mixtral-8×22B (c) GPT-3 175B

Figure 5.12: [Simulation] Training speed with cluster size scaling up.

across all models as the cluster size increases, while Lina performs better than Megatron in MoE models but still struggles with GPT-3 175B. Hermod scales well with the cluster size, outperforming the baselines by up to 1.50×. When training with 8192 GPUs, Hermod achieves 83% and 31% higher throughput compared to Megatron and Lina, respectively. As the cluster size increases from 1024 to 8192 GPUs, the speedup becomes more pronounced. We anticipate that Hermod's performance improvements will be even more significant when training larger models on larger clusters.

Different network bandwidths. We then evaluate the improvement of Hermod over the baselines under different network bandwidth settings, ranging from 40Gbps to 400Gbps. The parallelization strategies listed in Table 5.5 are used for each model in the 1024-GPU cluster.



(a) Qwen-MoE-24L (b) Mixtral-8×22B (c) GPT-3 175B

Figure 5.13: [Simulation] Training speed in the cluster of 128 servers with 1024 GPUs with different network bandwidths.

The results are shown in Figure 5.13. We observe that Hermod consistently outperforms the baselines across all large models under different network bandwidths. Specifically, when bandwidth is limited, such as at 40Gbps, Hermod achieves up to 2.06× speedup over Megatron due to its priority scheduling at both endhosts and switches. When bandwidth is sufficient, such as at 400Gbps, the improvement of Hermod is less pronounced as the network is no longer the bottleneck. However, Hermod still demonstrates better performance than the baselines, achieving up to 1.81× higher throughput by minimizing the blocking caused by EP/PP coflows and overlapping DP coflows.

5.6 Discussion

Coflows with multiple paths. Similar to [33,47,49], we assume that the routing of flows within and across coflows is determined by the network layer, utilizing mechanisms such as ECMP or packet spraying. It is conceivable that training performance could be further enhanced by co-designing the routing of flows alongside coflow scheduling [194]. Developing optimal algorithms for this problem presents an interesting direction for future research.

Multi-job scheduling. As highlighted in §5.2.1, Hermod focuses on coflow scheduling for a single LLM training job within the cluster, which is a common practice in modern training clusters [81, 171, 172]. However, a cluster may also have multiple jobs running concurrently. One way to extend Hermod for this scenario is by assigning each job a different weight, which would be used to determine the priority of their respective coflows. We leave the extension of Hermod to multi-job scheduling as future work.

Failure handling. Network failures are prevalent during training and can significantly impact the performance of the training job [61,70,109,132,145,150,200]. One approach to handle network failures is to identify them using diagnosis tools [81,109] and re-schedule the affected coflows to alternative paths, though this may come at the cost of increased JCT. We leave the development of scheduling algorithms for resilient coflows in LLM training as a future research direction.

Relationship with network transport. Hermod is implemented on top of transport layer with DiffServ support [25] for cluster-wide priority scheduling. We currently implement Hermod upon RoCEv2 (elaborated in §5.4), and leave the extension to other transport protocols, such as InfiniBand and MLT [157], as future work.

5.7 Related work

In addition to the related works mentioned in §5.1.3, we also summarize the related works in the following aspects.

General coflow schedulers. General coflow schedulers [33, 46–49, 53, 188] focus on optimizing the average CCT or minimizing job tardiness (*i.e.*, enabling more jobs to meet their deadlines). While these schedulers address general coflow scheduling, they do not account for dependencies between coflows derived from LLM training, which can significantly impact end-to-end training performance.

Network transport for ML Several works focus on optimizing network transport for ML training. MLTCP [135] allows the communication phases of jobs to interleave with one another, while MLT [157] takes advantage of ML workload characteristics to implement a 3-step progressive training scheme. These approaches are orthogonal to Hermod.

Joint optimizations with scheduling for training. SYNDICATE [117] accelerates DLRM training by jointly optimizing collective operators and execution planning across heterogeneous network interconnects. TACCL [142] synthesizes communication algorithms to generate efficient collectives for multi-node topologies. These works focus on optimizing collective operations over heterogeneous networks and are orthogonal to our approach.

AI-specific Networks. Various AI-specific networks have emerged. TopoOpt [162] co-optimizes topology and parallelization strategies. TPUv4 [84] uses optical networks to improve `AlltoAll` traffic. Groq [31] develops a software-defined hardware interconnect for deterministic training chips. Meta [61] discusses customized routing schemes, optimized collectives, and network resiliency for large-scale RoCE AI networks. Alibaba-HPN [132] proposes a dual-ToR design for failure resilience based on AI workload characteristics. Megascale [81] co-designs algorithms and systems for training over 10,000 GPUs and provides network diagnosis tools. Integrating Hermod into these systems could further enhance LLM training performance.

5.8 Conclusion

We present Hermod, a near-optimal coflow scheduler that orchestrates all coflow types at the cluster level during LLM training. Hermod employs the insight that coflows are characterized by three model factors, and designs (1) model factor-driven inter-coflow scheduling to align with the LLM training DAG, and (2) matching-based intra-coflow scheduling to maximize transmission rates for high-priority coflows. Extensive evaluations validate that Hermod achieves its design goals. We believe our revisit of coflow scheduling in LLM training will inspire further LLM training optimizations.

CHAPTER 6

LEO: A HARDWARE-ACCELERATED COMMUNICATION FRAMEWORK FOR DISTRIBUTED SERVICES

In-Network Computing (*INC*) has emerged as a promising technique to mitigate the gap between high-speed network bandwidth and stagnant CPU compute capability. By offloading partial or entire computation tasks to accelerators or switches located along the datapath, INC alleviates the processing load on the CPU, reduces communication cost, and enhances overall computation throughput.

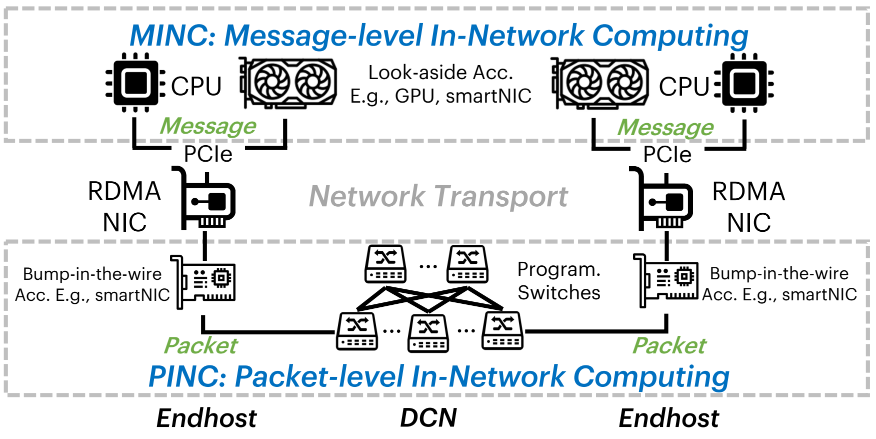

MINC: Message-level In-Network Computing
CPU
Look-aside Acc.
E.g., GPU, smartNIC
CPU
Message
PCIe
Message
PCIe
RDMA NIC
Network Transport
RDMA NIC
Bump-in-the-wire Acc. E.g., smartNIC
Program. Switches
Bump-in-the-wire Acc. E.g., smartNIC
Packet
Packet
PINC: Packet-level In-Network Computing
Endhost
DCN
Endhost

Figure 6.1: Categorization of In-Network Computing. The network stack in general can be classified into two domains: the message domain above transport, and the packet domain below transport. In this paper, the in-network computing within the message domain (*e.g.*, key-value store), is called message-level INC, while that within the packet domain (*e.g.*, packet encryption), is called packet-level INC.

INC can be categorized into two types: packet-level INC (*PINC*) and message-level INC (*MINC*), based on whether the computation occurs on packets (below the transport layer), or on messages (above the transport layer), as depicted in Figure 6.1. PINC, which facilitates computation over packets, is primarily executed by in-network devices such as bump-in-the-wire (BITW) smartNICs and programmable switches for datacenter applications such as in-network aggregation [96, 138] and network management [59]. MINC, which operates on messages, is executed by look-aside accelerators such as GPUs, FP-GAs, and smartNICs. Owing to the prosperity of applications at end-host, MINC is deployed in multiple scenarios in the datacenter, including key-value store (KVS) [97], distributed file system (DFS) [13], elastic block storage (EBS) [119], and distributed machine learning (DML) [80], etc. Despite the extensive research on PINC application development [62,174,193], the exploration in the realm of MINC has garnered less attention, given its wide applicability and potential for performance improvement.

The development of MINC applications contains two major aspects: computation and communication. For computation, the task is relatively straightforward, entailing the independent programming of hardware acceleration functions in distinct accelerators. However, communication presents a more intricate challenge, as it involves the effective organization and implementation of dataflows among devices, thereby demanding advanced system expertise from the application developers[1]. We reveal three critical problems encountered in MINC communication development as follows:

#1: Poor portability across applications and hardware. When porting MINC applications to various hardware, the issue of poor portability becomes particularly evident. Notably, there have been several MINC applications such as KVS offloaded to different hardware, *e.g.*, FPGA [97] or SoC-based smartNICs [111,168]. However, adapting application dataflows between these hardware while maintaining high communication efficiency demands substantial engineering efforts. These efforts include, but are not limited to, the reimplementation of dataflows due to different communication APIs, and the restructure of offloading strategies due to hardware performance variances, etc. Experienced engineers have estimated that such porting could take from weeks to months to complete, highlighting the significance of portability.

#2: Under-utilized resources in existing MINC systems. In MINC context, both CPU and accelerator resources are available for use, providing an inherent opportunity to optimize performance by employing these resources simultaneously. Unfortunately, current MINC systems [97, 119, 139] tend to neglect this opportunity, favoring instead to offload tasks to accelerators as much as possible. Such neglect results in the loss of computational resources in CPUs and communication resources available in CPU-NIC links.

[1]In the rest of the paper, we refer to the application developers as developers for short.

#3: Complex configuration for direct transmission. When developing MINC communication, configuring direct data transmission between devices is complicated due to its inherent complexity. This complexity stems from several intricate operations such as memory pinning and mapping, memory exposure, and address translation functions registration, etc. Such configuration is not only tedious but also error-prone for developers, especially for those lacking specialized expertise in system configuration. Furthermore, the different configuration APIs for varying hardware platforms complicate the configuration process, raising the bar for developers.

Given these problems, we argue the need for a new framework that supports portability across applications and hardware, incorporates hybrid resource utilization, and provides automatic configuration with unified APIs across devices. With such a framework, developers would be able to offload routine communication tasks to the framework and focus on the application-specific logic.

In this paper, we present LEO, a generic and efficient MINC communication framework. Specifically, LEO makes the following three notable contributions:

- **Generic abstractions for application and hardware.** To facilitate portability, LEO proposes generic abstractions for both application and hardware. Specifically, LEO introduces a *communication path* abstraction to accommodate diverse communication requirements of applications with predictable communication performance across hardware. Utilizing this abstraction, application dataflows can be conceptually represented as a sequence of paths with predictable communication performance, and hardware is abstracted into different models based on its topology and associated pre-profiled performance metrics, hence addressing **#1**.

- **Built-in multi-path optimization.** LEO integrates built-in multi-path communication optimization to efficiently utilize resources in both CPU and accelerator. We formulate the multi-path selection as a linear programming (LP) problem, which is solvable optimally using classical LP solvers [118]. To facilitate real-time path selection, we propose a heuristic algorithm that first generates a path selection priority list offline and then selects the appropriate path online based on the monitored intra-host job completion times (JCTs) of previous requests, reflecting the current path status, thereby addressing

#2.

- **Unified APIs and wrappers.** To free developers from the burden of complex configuration and facilitate easy programming, LEO exposes unified IB-style APIs along with wrappers to adapt to various vendor-specific configuration APIs for supporting automatic configuration scheme. This scheme adheres strictly to standard communication protocols, *i.e.*, PCIe DMA and RDMA, making it versatile enough to configure most device types, therefore addressing **#3**.

We have implemented a prototype of LEO and evaluated it with four representative case studies on three testbeds covering devices including FPGA-based and SoC-based smartNICs and GPU. Our experimental results confirm that LEO is generic and maintains high performance for both applications and accelerators with negligible overhead. Specifically, LEO provides 1.2-4.7× higher speedup than baselines in four case studies. Furthermore, the heuristic multi-path solution employed by LEO yields 1.18–2.17× better throughput than two greedy algorithms, and delivers comparable performance to the optimal solution while satisfying the real-time selection demand. Finally, the overhead of LEO is negligible, accounting for 10.3% of CPU utilization and 1.5µs for path selection.

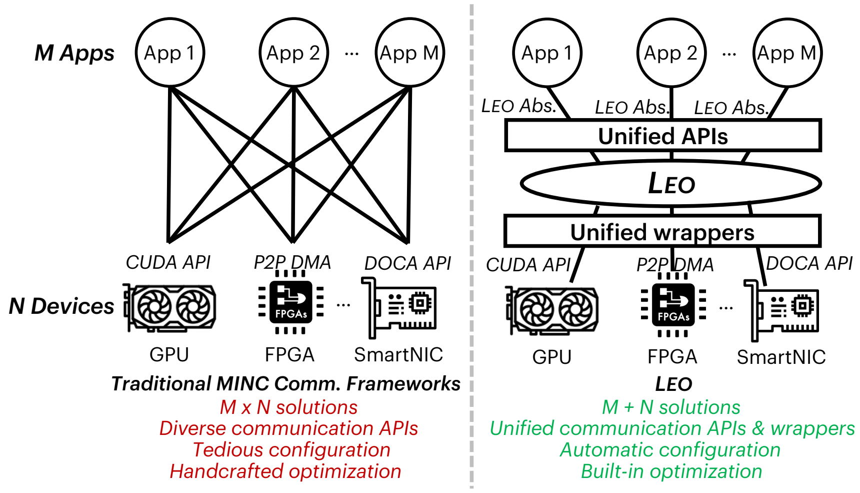


Figure 6.2: Comparison between traditional MINC communication frameworks and LEO.

Figure 6.2 summatively compares LEO against traditional MINC communication frameworks. In traditional frameworks, supporting M dataflows over N hardware types necessitates developers to handcraft M×N solutions, each reliant on vendor-specific APIs.

This process is further complicated by the need for tedious and error-prone direct data transmission configuration, as well as handcrafted communication optimization strategy. In contrast, LEO introduces expressive abstractions along with unified APIs and wrappers that simplify the programming and configuration processes for both application and hardware. This effectively reduces the development efforts to M+N solutions. Furthermore, LEO incorporates a built-in multi-path communication to utilize both CPU and accelerator.

6.1 Background and Motivation

We first provide the background of message-level in-network computing (MINC), and then discuss the problems of MINC application development.

6.1.1 Category of In-Network Computing

In-Network Computing (*INC*) is a class of hardware acceleration approaches that offloads part of or whole computation on the network datapath in order to free up CPU cycles, reduce communication overhead, and boost application performance [82, 83, 96, 97, 104, 114, 138, 139].

Figure 6.1 shows the categorization of INC. Based on whether the computation occurs on packets (below the transport layer) or on messages (above the transport layer), INC can be further categorized into packet-level INC (*PINC*) and message-level INC (*MINC*). PINC processes computation over packets by programmable devices within the network, *e.g.*, bump-in-the-wire (BITW) accelerators [59] and programmable switches [4], with each packet size restricted within Maximum Transmission Unit (MTU), *e.g.*, 1500 B. On the other hand, MINC performs computation over messages by look-aside accelerators at endhost such as GPUs, FPGAs, and smartNICs, with the message size ranging from several bytes to 10s of GB.

The differences in the size of data processing unit and the location of accelerators result in distinct supported applications and developers: PINC favors boosting those applications whose semantics can be expressed within a packet size, *e.g.*, in-network aggregation [96, 112, 138] and in-network cache [83, 114], and is maintained mostly by infrastructure maintainers; MINC, however, is prosperous in boosting applications that operate on a collection of data, including key-value stores (KVS) [97, 139], file compression and decompression in distributed file system (DFS) [13], metadata operations in elastic block storage (EBS) [119], and collective communication in distributed machine learning (DML) [80, 157], etc., thanks to its wide range size, and is typically operated by application developers. While there has been considerable attention towards PINC frameworks in recent years [62, 98, 174, 193], frameworks for developing MINC applications have not received equal focus, despite their wide applicability and the potential for substantial performance improvements.

6.1.2 Problems of MINC Application Development

The development of MINC applications necessitates efforts in both computation and communication aspects. For the computation aspect, the developers typically engage in the programming of hardware acceleration functions for each device independently. Thought time-consuming, this process is relatively easy to handle. The communication aspect, however, poses greater difficulties. It demands not only effective coordination among various devices but also the efficient implementation of dataflows for specific application, making the communication aspect more challenging to handle.

We discuss three problems of MINC application development caused by the communication aspect using examples of state-of-the-art solutions.

Poor portability across applications and hardware. Portability is of paramount importance in application development, particularly given the diversities both in upper-layer dataflows and lower-layer hardware platforms. Notably, multiple MINC applications such as KVS have been successfully offloaded to different kinds of hardware platforms including FPGA-based [97] and SoC-based smartNICs [111,168]. This process, however, obligates developers to reimplement application dataflows between devices using hardware-specific APIs and communication modules, leading to increased yet redundant development efforts. For instance, developing and testing dataflows with Register Transfer Level (RTL) language for FPGA-based hardware can consume as much as two months for proof-of-concept (PoC). If using C language over SoC-based smartNICs, the development time may be reduced, but still costs around two weeks[2]. Therefore, the task of porting M dataflows to N hardware platforms may result in an average development time of around 1.25×M×N months, posing substantial burdens for developers. A more efficient approach is to develop M distinct dataflows and N hardware solutions independently, followed by integration through an intermediate layer. This manner effectively reduces the development time to 1.25×(M+N), resulting in considerable time savings, *e.g.*, a reduction of 8.75 months in the scenario where M is 5 for the type of dataflows in KVS [168] and N is 3 for 3 kinds of hardware models. The time savings can increase with more kinds of application dataflows and hardware with varying communication APIs.

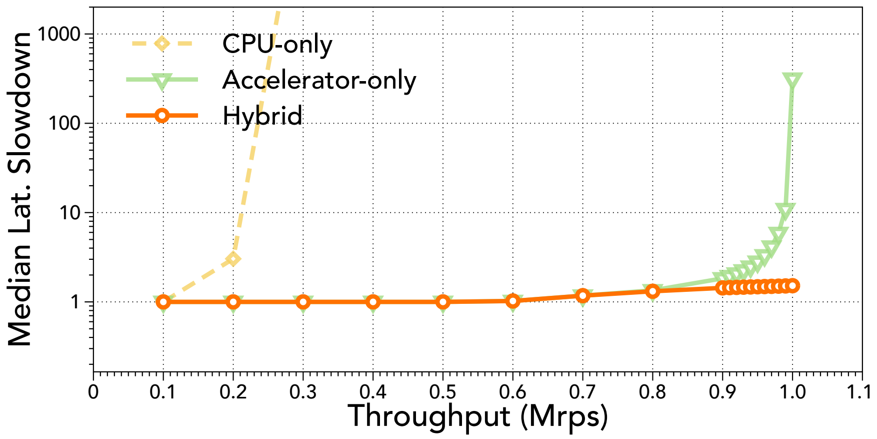


Figure 6.3: The performance improvement with hybrid dataflows optimization.

Under-utilized resources in existing MINC systems. Existing MINC systems do not maximally utilize the available resources in the hardware platform. Specifically, traditional MINC systems typically use the accelerator in the datapath for fast processing, but ignore the usage of available host CPU resources[3] [97, 119, 139]. That is, both the CPU resources at endhost and the CPU-NIC links are unused during processing, exhibiting an optimization opportunity of hybrid solutions that employ both the accelerator-only and CPU-only dataflows. To reveal the potential of this optimization, we conduct an experiment to emulate the processing of a KVS application at the server which is equipped with one CPU and one smartNIC. We reuse the throughput results reported in [139], *i.e.*,

[2]These development time were estimated by experienced FPGA and SoC engineers in a major company.

[3]iPipe [111] supports hybrid usage of host CPU and accelerator, but its approach is inefficient when network fluctuates, as revealed in §6.4.2.

23.0Mops for 16-core CPU and 71.8Mops for 24-core smartNIC. For the hybrid solution, we randomly select the dataflow for each request during processing. As shown in Figure 6.3, the throughput of the hybrid solution performs significantly better than those of two single-path settings thanks to its better utilization of the available resources. This result indicates that a middle-layer framework with hybrid dataflows optimization can demonstrate better performance improvement for MINC applications.

```
1  void init() {
2      // Init device configuration
3      device_config(pcie_addr_acc);
4      device_config(pcie_addr_nic);
5      // Make devices visible between each other
6      expose_dev_mem(acc, nic);
7  }
8
9  void expose_dev_mem(dev1, dev2) {
10     // The involved configuration
11     configure_role(dev1, dev2);
12     reg_acc_mem(dev2);
13     reg_mapping_func(dev2_drv, dev1_func);
14 }
```

Listing 6.1: Example pseudocode of communication configuration.

Complex configuration for direct transmission. Configuring direct data transmission between devices presents a significant challenge due to the complexity and variety across different device types. In MINC context, direct device-to-device transfer is essential for achieving high performance [9]. However, the facilitation of such transfer is notably difficult, particularly for developers lacking expertise in system configuration. Listing 6.1 describes pseudocode of the conventional two-fold configuration process: 1) the initialization of device configurations, such as memory pinning for accelerator and NICs (Line 3-4); and 2) the exposure of device memory across devices for direct data transmission (Line 6). Both processes demand sophisticated configuration. We set the second process as an example for illustration. This process requires firstly configuring roles for the corresponding two devices to figure out which device is responsible for activating the direct device transmission (Line 11); then using the host CPU to register the accelerator's mapped BAR memory in NIC (Line 12); and finally exposing the accelerator driver's virtual-physical address mapping function to the NIC driver (Line 13). These operations, while critical, are laborious and susceptible to errors when executed by developers, particularly for those lacking system configuration expertise. Furthermore, the configurations vary across devices due to the differences in configuration APIs, *e.g.*, DOCA for BlueField DPU and Netronome for Agilio smartNIC. Such variance raises both the study and migration costs for developers, resulting in complex device configurations.

6.2 LEO Design

We aim to design a middle-layer communication framework for MINC applications that addresses the above problems simultaneously. Specifically, this framework should achieve the following design goals:

- *Generic.* The framework should be generic to support multiple kinds of dataflows and diverse hardware platforms. This genericity enables the portability across applications and hardware, and hence reduces the development workload for developers.
- *Efficient.* The communication performance this framework provides should be as efficient as possible. Specifically, the framework should utilize the resources both in CPU and accelerator with high efficiency.
- *Easy-to-use.* The framework should be easily used by developers with simple APIs and support automatic communication configuration.

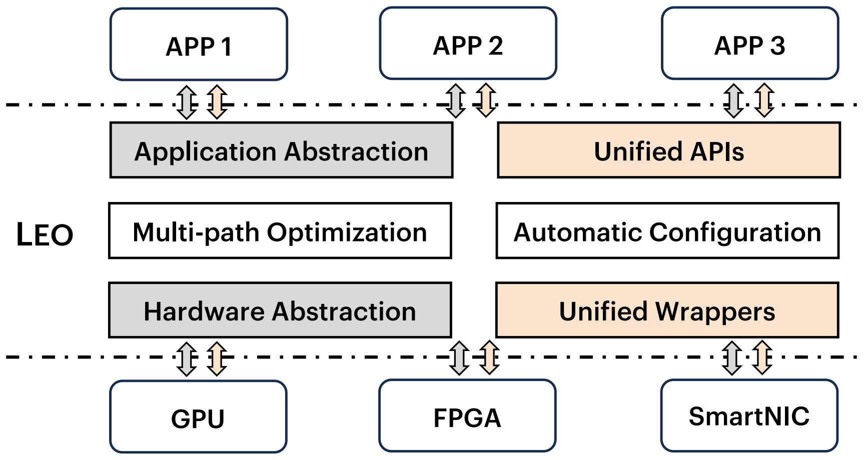
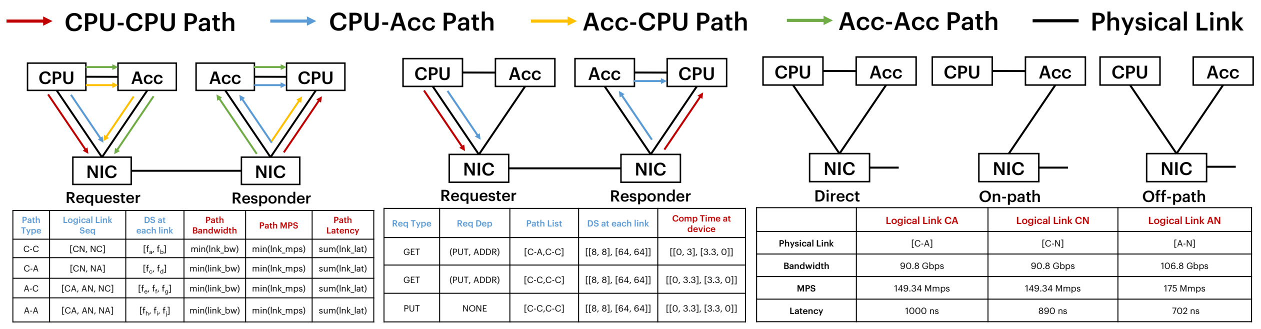


Figure 6.4: LEO overview.

Figure 6.4 shows an overview of LEO. LEO proposes the abstractions of application and hardware for describing the diversities of both sides (§6.2.1). For efficiency, LEO designs a built-in multi-path communication optimization for hybrid resource utilization (§6.2.2). For ease of programming and configuration, LEO introduces the unified APIs and wrappers to developers (§6.2.3) and facilitates automatic communication configuration (§6.3). Finally, we put the above designs together to illustrate the architecture and workflow (§6.2.4).

6.2.1 MINC Communication Abstractions

In this section, we start by introducing the abstraction of the *communication path*, which can describe diverse dataflows and estimate their performance based on the attached hardware transmission features provided by the lower layer. We then elaborate on the application and hardware abstractions.



Path Type	Logical Link Seq	DS at each link	Path Bandwidth	Path MPS	Path Latency
C-C	[CN, NC]	$[f_a, f_b]$	min(lnk_bw)	min(lnk_mps)	sum(lnk_lat)
C-A	[CN, NA]	$[f_c, f_d]$	min(link_bw)	min(lnk_mps)	sum(lnk_lat)
A-C	[CA, AN, NC]	$[f_e, f_f, f_g]$	min(link_bw)	min(lnk_mps)	sum(lnk_lat)
A-A	[CA, AN, NA]	$[f_h, f_i, f_j]$	min(link_bw)	min(lnk_mps)	sum(lnk_lat)

Req Type	Req Dep	Path List	DS at each link	Comp Time at device
GET	(PUT, ADDR)	[C-A,C-C]	[[8, 8], [64, 64]]	[[0, 3], [3.3, 0]]
GET	(PUT, ADDR)	[C-C,C-C]	[[8, 8], [64, 64]]	[[0, 3.3], [3.3, 0]]
PUT	NONE	[C-C,C-C]	[[8, 8], [64, 64]]	[[0, 3.3], [3.3, 0]]

	Logical Link CA	Logical Link CN	Logical Link AN
Physical Link	[C-A]	[C-N]	[A-N]
Bandwidth	90.8 Gbps	90.8 Gbps	106.8 Gbps
MPS	149.34 Mmps	149.34 Mmps	175 Mmps
Latency	1000 ns	890 ns	702 ns

(a) Communication path. (b) Application abstractions. (c) Hardware abstractions.

Figure 6.5: MINC communication abstractions. The blue features should be provided by developers, and the red features can be automatically profiled by LEO.

Communication path. The communication path describes the logical links traversed by the request from the requester to the responder, above the transport layer, with each path associated with specific hardware transmission features. As shown in Figure 6.5a, there are four types of communication paths in MINC communication, *i.e.*, CPU-CPU path, CPU-accelerator path, accelerator-CPU path, and accelerator-accelerator path. The features of each path include:

- *Logical link sequence*: This describes the logical link sequence traversed by the request from the requester to the responder from the perspective of developer. For example, the CPU-CPU path sequentially traverses CN (CPU-NIC) at requester and NC (NIC-CPU) at responder.

- *Data size at each link*: This feature describes the size of data traversed at each link during request processing. This value is defined by developers to be elaborated in the application abstractions.
- *Estimated path performance*: These features are determined by metrics measured on the hardware platform, as detailed in the hardware abstractions.

The benefits of the communication path abstraction are two-fold. First, it provides a uniform abstraction to describe a dataflow from the requester to the responder, and can be extended to describe multiple round trips by composing a sequence of multiple communication paths, thereby successfully describing the diverse dataflows from the application layer. Second, it conveys estimated quantitative performance metrics derived from the hardware layer to showcase its communication performance. These estimated metrics are essential for multi-path optimization to analyze the communication interference across paths (§6.2.2).

Application abstractions. Leveraging the communication path abstraction, the dataflow of MINC communication can be conceptually expressed as a sequence of multiple communication paths, with each path conveying a request for one round trip. Formally, the application abstractions are formulated as follows, where we use KVS for illustration:

- *Request type*: This represents the type of the request for processing, *e.g.*, `GET` and `PUT` in KVS.
- *Request dependency*: This describes the dependency between requests to ensure correct execution, as out-of-order cases may occur along with multi-path optimization (§6.2.2). In KVS, a `PUT` request must be executed before a `GET` request if both access data at the same address to avoid the `GET` obtaining a stale value. Dependency types include: `NONE` (no dependency), `ADDR` (same address), and `MSG` (same message).
- *Path list*: This describes the sequence of communication paths traversed for the corresponding dataflow. In KVS, a `GET` request has two dataflow types, whereas `PUT` has only one, each involving two round trips due to the network amplification effect [168].

- *Data size at each link*: This describes the predetermined size of the data structure traversed at each link during request processing. In KVS, a `GET` request firstly traverses the 8-byte key at each link and then traverses the 64-byte value on the same path. The `PUT` request operates similarly.

- *Compute time*: This lists the compute time or memory access time at each endhost device, which can be profiled before processing the application.

Hardware abstractions. The hardware abstractions can be explicitly described by the hardware model of the accelerator and the associated performance metrics, *i.e.*, bandwidth, message per second, and latency [110]. We illustrate the hardware abstractions in Figure 6.5c, showcasing the direct model of an FPGA-based hardware platform (§6.3) transmitting 64B messages over PCIe Gen3x16. The hardware model is expressed in nodes and edges, where each node represents a hardware device (CPU, accelerator, NIC), and each edge represents a physical link between devices. Note that the mapping between logical links and physical links is determined by the hardware model. We provide three representative hardware model abstractions [111, 165]: direct, on-path, and off-path, and support customized model abstractions for flexibility. The performance metrics are measured during initialization with profiling tools such as VTune [5] and Hostping [110] for intra-host communication.

6.2.2 Multi-path Optimization

LEO should support efficient multi-path optimization for hybrid resource utilization. Unlike previous multi-path works which select paths between hosts [115, 134], LEO selects paths within each host, exhibiting several unique properties: i) Only intra-host communication needs to be considered since inter-host communication is handled and transparentized by the hardware transport, *e.g.*, RDMA [166]; ii) Sufficient information is available in both the application and hardware, including communication paths for each request type, profiled compute times, and hardware models.

These properties simplify the path selection problem, making it solvable theoretically. We first formulate the path selection problem as a linear programming (LP) problem, which is solvable optimally with off-the-shelf LP solvers [118] (§6.2.2). Next, we propose a heuristic algorithm to address the time constraints of the LP solver (§6.2.2). Finally, we discuss some special cases (§6.2.2).

Symbols	Description
$P = \{p_1, p_2, ..., p_i\}$	Set of all paths
$L = \{l_1, l_2, ..., l_n\}$	Set of all intra-host links
$C = \{c_1, c_2, ..., c_o\}$	Set of all compute nodes
$R = \{r_1, r_2, ..., r_t\}$	Set of all request types
$\mathcal{L}(p_i) = [c_{i1}, l_{i1}, ..., c_{ik}]$	Sequence of nodes and links in p_i
$\mathbb{P}(r_t) = [p_{t1}, ..., p_{tn}]$	Path list of r_t
Δt_t	Time interval between r_t and r_{t+1}
$S(r_t)$	Size of r_t
$BW(l_j)$	Bandwidth capacity of l_j
$T_{tr}(l_j)$	Base traverse time of l_j
$T_c(c_j, r_t)$	Compute time of c_j for r_t
$X(p_i)_t = \{0, 1\}$	Whether select request r_t to p_i at t
$Q(l_j)_t$	Queue latency of l_j at step t
$Q(c_i)_t$	Queue latency of c_i at step t

Table 6.1: Key notations in problem formulation.

Problem Formulation

Table 6.1 lists the key notations used for problem formulation. We address the path selection problem at a fine-grained, per-request level. Specifically, we discretize the entire processing time into separate steps, with each step spanning Δt_t, representing the arrival time interval between request r_t and r_{t+1}. Our primary objective is to minimize the total intra-host JCT, which are the sum of their compute and communication time within hosts. As the compute time $T_c(c_j, r_t)$ is fixed and known in advance, our main focus is to minimize the communication cost, defined as the sum of transmission time and queue latency at each link and compute node.

While the transmission time is fixed, the queue latency varies over time and depends on the previous queuing status:

$$Q(c_i)_t = \begin{cases} \max\{Q(c_i)_{t-1} + T_c(c_i, r_t) - \Delta t_t, 0\} & \text{if } c_i \in \mathcal{L}(p_i) \\ & \&\& X(p_i)_t = 1 \\ \max\{Q(c_i)_{t-1} - \Delta t_t, 0\} & \text{otherwise} \end{cases} \tag{6.1}$$

$$Q(l_j)_t = \begin{cases} \max\{Q(l_j)_{t-1} + \frac{S(r_t)}{BW(l_j)} - \Delta t_t, 0\} & \text{if } l_j \in \mathcal{L}(p_i) \\ & \&\& X(p_i)_t = 1 \\ \max\{Q(l_j)_{t-1} - \Delta t_t, 0\} & \text{otherwise} \end{cases} \tag{6.2}$$

where $Q(c_i)_t$ and $Q(l_j)_t$ increases only if the request at time t selects p_i and traverses c_i or l_j, otherwise they decrease by Δt_t. While these equations contain non-linear $\max\{\cdot\}$ functions, we apply the big-M method to linearize them for resolution.

We denote the queue latency and traverse time of path p_i at time t as $Q(p_i)_t = \sum_{c_i \in \mathcal{L}(p_i)} Q(c_i)_t + \sum_{l_j \in \mathcal{L}(p_i)} Q(l_j)_t$, and $T(p_i, r_t)_t = \sum_{l_j \in \mathcal{L}(p_i)} T_{tr}(l_j) + \sum_{c_j \in \mathcal{L}(p_i)} T_c(c_j, r_t)$, respectively. Formally, the objective function and constraint of k-step path selection problem are:

$$\min \sum_{t=n}^{t=n+k} \sum_{p_i \in P} X(p_i)_t \cdot [Q(p_i)_t + T(p_i, r_t)_t] \tag{6.3}$$

$$\sum_{p_i \in \mathbb{P}(r_t)} X(p_i)_t = 1 \tag{6.4}$$

where (6.3) aims to minimize the total intra-host JCTs for all paths over a short horizon, *i.e.*, the next k request sequence predicted using historical request trace [178], while (6.4) imposes the path selection constraint, ensuring that each request is assigned to exactly one path.

Notably, when k=1, the LP algorithm simplifies to a greedy version that selects the path with the least JCT for each request type. This approach optimistically assumes the network can handle all queued requests. However, in practice, high request throughput can cause queues to accumulate across paths, leading to high queuing latency and degraded performance, as our results reveal (§6.4.3).

Heuristic Multi-path Selection

Though promising, the time cost of LP solver [118] is notably high, *e.g.*, more than 2.4 ms in our evaluation (§6.4.3), which is beyond the real-time requirements of µs-level applications. To address this, we propose a per-request-type heuristic algorithm for real-time path selection. We observe a trade-off between selecting a queuing path with low latency and an idle path with high latency. Adding a request to the queuing path increases the queue latency for all subsequent requests until the queue is drained, leading to an extra cost $\hat{k} \cdot [\sum_{c_i \in \mathcal{L}(p_i)} T_c(c_i, r_t) + \sum_{l_j \in \mathcal{L}(p_i)} \frac{S(r_t)}{BW(l_j)}]$ according to (6.3), where $\hat{k}$ denotes the number of requests that will experience the queue latency caused by the current request. We call it the *communication toll* of the path decision. Finally, the trade-off is transformed into a balance between the immediate JCT gain and the incurred communication toll.

Algorithm overview. Our algorithm operates in two phases: first, we calculate the traverse time and communication tolls for each request-path pair offline. Then, we estimate the path status online based on the intra-host JCTs of completed requests to select appropriate paths.

In the offline phase, we calculate the basic path traverse time $T(p_i, r_t)_t = \sum_{l_j \in \mathcal{L}(p_i)} T_{tr}(l_j) + \sum_{c_j \in \mathcal{L}(p_i)} T_c(c_j, r_t)$ and the communication toll $Q_{toll}(p_i)_t = \sum_{c_i \in \mathcal{L}(p_i)} T_c(c_i, r_t) + \sum_{l_j \in \mathcal{L}(p_i)} \frac{S(r_t)}{BW(l_j)}$ for each path p_i and request type r_t as: We use the path traverse time $T(p_i, r_t)_t$ to generate the initial path selection priority list for each request type, as the queue is initially empty in each path.

Next, in the online phase, we select the appropriate path based on both the current JCT and the communication toll. We denote $\hat{k}_{p_i}$ as the estimated number of requests that will experience queue latency caused by the current request if selecting p_i. Then, the JCTs of the subsequent $\hat{k}_{p_i}$ requests will also increase by $Q_{toll}(p_i)_t$. Therefore, the total intra-host JCT is:

$$T_{total}(p_i)_t = \hat{k}_{p_i} \cdot Q_{toll}(p_i)_t + T(p_i, r_t)_t \tag{6.5}$$

By simply comparing the $T_{total}(p_i)_t$ of all paths, we select the path with the minimal value as the final choice, thus approximating (6.3).

Estimation of $\hat{k}_{p_i}$. The key challenge of the algorithm is to estimate the value of $\hat{k}_{p_i}$ to accurately reflect the status of each path. We achieve this by monitoring the intra-host JCTs of previous requests, following a method similar to [95], and using these JCTs as indicators of the current path status. Notably, modern RNICs widely support accessible hardware timestamps at the host [24, 95]. Therefore, LEO uses multiple hardware and software timestamps to collaboratively monitor the intra-host JCT of each path.

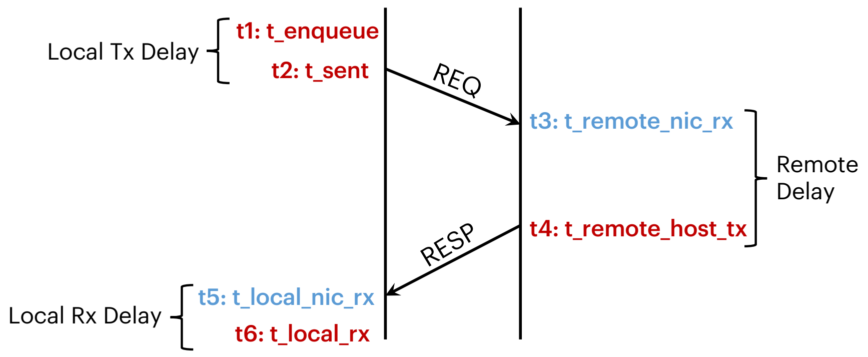


Figure 6.6: Timestamps used to measure different delays at endhosts. The hardware and software timestamps are shown in blue and red, respectively. The intra-host JCT is the sum of all delays shown in the figure.

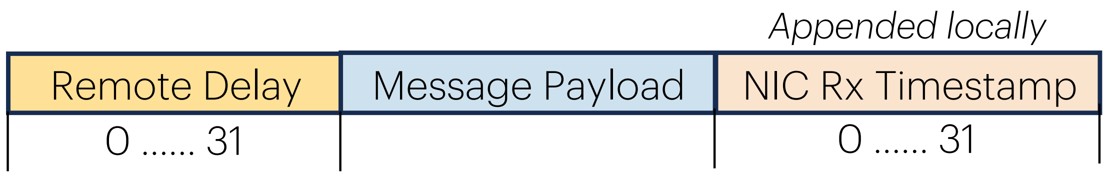


Figure 6.7: Message format of LEO.

Figure 6.6 illustrates the event sequence of a request traversing the path. t_1 and t_2 are the times when the message is enqueued to the corresponding path and to the NIC queue, respectively, recorded by LEO using the CPU. t_3 is the time when the descriptor of the request, *i.e.*, CQE, is obtained at the server side by RNIC. t_4 is the time when the message is enqueued to the NIC queue, recorded by the CPU. Note that t_4-t_3 is calculated by the server CPU and appended in the message header. t_5 and t_6 are the corresponding receive timestamps for the response at the client side and are appended locally at the client. Details of the message format is shown in Figure 6.7. Finally, the overall intra-host JCT of the path is calculated as $(t_2 - t_1) + (t_4 - t_3) + (t_6 - t_5)$.

Based on the above monitored intra-host JCT, we can now estimate the value of $\hat{k}_{p_i}$ for each path. Specifically, we initialize $\hat{k}_{p_i}$ to 0 for each path and follow these procedures during application runtime: i) Upon receiving a completed request, we compare the monitored JCT with the base traverse time $T(p_i, r_t)_t$; ii) If the monitored value is larger, it indicates the queue exists in the path, and we increase $\hat{k}_{p_i}$ by one, otherwise we reset $\hat{k}_{p_i}$ to zero.

Discussion

Besides the common case, we additionally discuss several other cases as follows:

- **Request dependency.** Multi-path optimization can violate the sequential execution requirement for some requests. Therefore, we pre-check the request dependency feature in the abstraction (§6.2.1) and ensure that incoming requests with dependencies on other in-progress requests are executed on the same path.
- **Parallel processing at compute node.** For simplicity, the above formulation does not emphasize parallel processing, but it can be easily extended to the parallel case. We introduce a variable for the parallel capacity of the compute node and modify equations (6.1) and (6.2) to increase queue latency when the capacity is exceeded. The heuristic algorithm remains unchanged as $\hat{k}_{p_i}$ is not affected in the parallel context.
- **NIC without timestamp support.** LEO is currently built on hardware transport RDMA, where NICs widely support hardware timestamp services [24]. For NICs that do not support timestamp services, we may use software timestamps instead of hardware timestamps or record the end-to-end JCT. Both methods could reduce the accuracy of queue latency estimation. We leave accurate estimation for this scenario to future work.

6.2.3 LEO Programming

In this section, we illustrate LEO programming. We first introduce the unified LEO APIs and wrappers, and then present the programming procedures of LEO.

APIs and wrappers. LEO wraps its communication and memory configuration together with communication optimization in unified APIs and wrappers. As listed in Table 6.2, LEO APIs and wrappers include:

- `init` initializes the MINC communication. It automatically sets up direct communication across devices based on the registered handlers and functions.

APIs	Description
`init()`	Init LEO comm. context.
`send(m)`	Two-sided send a message.
`recv()` $\rightarrow$ `m`	Two-sided receive a message.
`write(m)`	One-sided write a message.
`read()` $\rightarrow$ `m`	One-sided read a message.
`handle(m,r)`	Handle a message for role.
Wrappers	**Description**
`reg_comm_func(f,n,r)`	Register a comm. function.
`reg_mem_func(f,n,r)`	Register a memory function.
`reg_ts_func(f,n)`	Register a timestamp function.
`reg_app_func(f,n,r)`	Register an application function.

Table 6.2: LEO APIs and wrappers. The developers can use this set of APIs and wrappers to 1) initialize MINC communication, 2) perform communication operations, 3) register hardware-specific communication, memory, timestamp service, and application functions.

- `send`, `recv`, `write`, and `read` are simple RDMA ib_verbs-style communication APIs for data transmission.

- `handle` is a unified API for handling incoming requests. Upon receiving a request, it automatically invokes the registered application functions at the specified device role. Note that the role can be either "cpu" or "acc".

- `reg_comm_func` is a function pointer wrapper that registers a hardware-specific communication function, such as `ibv_post_send` for send operation in RDMA NIC.

- `reg_mem_func` is a function pointer wrapper that registers a hardware-specific memory function, such as `malloc` for memory allocation in the host CPU.

- `reg_ts_func` is a function pointer wrapper that registers a timestamp service function, such as `rdtsc` for timestamping in RDMA NIC.

- `reg_app_func` is a function pointer wrapper that registers necessary functions for processing such as index searching in key-value store [169].

Programming procedures. Leveraging the unified LEO APIs and wrappers, the developers can program MINC applications easily with the following programming procedures:

1. Implement application functions. The developers should implement the application functions for processing the incoming requests at both CPU and accelerator.
2. Configure the communication path in `YAML` format based on the communication path abstraction in §6.2.1.
3. Configure the dataflows for each request type in `YAML` format based on the application abstractions in §6.2.1.
4. Write initial configuration code. The developers should write the initial configuration code with LEO APIs and wrappers to register all necessary functions, pin memory, configure memory mapping operations, and initialize the dataflows.
5. Finally, the developers should write the application processing code with LEO APIs.

6.2.4 LEO Architecture

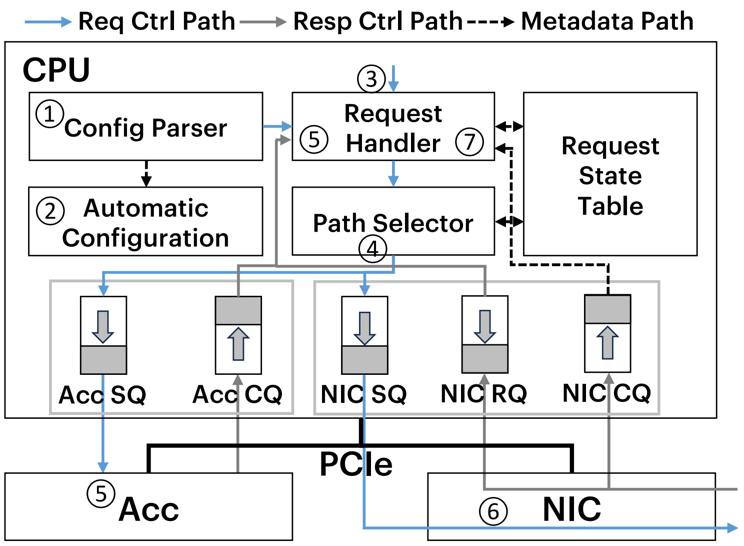


Figure 6.8: LEO architecture.

Taking the designs together, the architecture of LEO is depicted in Figure 6.8, which includes five major components together with five queues [17,21] for managing the communication of accelerator and NIC.

At the requester side, the workflow operates as follows:

1. The *config parser* parses the developer's `YAML` files, and then sends the dataflow information to the *request handler* as well as the hardware function wrappers to the *automatic configuration* module.

2. The automatic configuration module sets up the communication and registers memory regions between devices.

3. After the communication is established, the application inputs each request to the request handler. The handler registers the request in the request state table (RST) and hands it over to the *path selector*.

4. The path selector analyzes the request's historical path status in RST, executes the path selection algorithm (§6.2.2), and then selects the appropriate path.

5. If the selected path is Acc-CPU or Acc-Acc, the accelerator should process the request first. The request descriptor is enqueued to the Acc SQ and fetched by the accelerator. When the accelerator finishes processing, it enqueues the completion queue element (CQE) and notifies the request handler to update the request status, which then passes the request descriptor to the path selector. Otherwise, the NIC handles the request directly and this step is skipped.

6. The path selector then enqueues the request descriptor to the NIC SQ to transmit the request to the network. The NIC can fetch the request data either from CPU or accelerator, according to the descriptor information, and transmit it to the remote server.

7. When the NIC receives the corresponding response, it enqueues a CQE to NIC CQ, and the request handler obtains the hardware timestamp in CQE. The request handler then calculates the intra-host JCT of the request (§6.2.2), and updates the request status in RST.

At the responder side, the workflow operates similarly.

6.3 Implementation

We have implemented a fully functional LEO prototype for developing MINC communication. It currently supports representative FPGA-based smartNICs [11], SoC-based BlueField-3 [6] and GPU. Below we introduce the implementation details of LEO, our efforts undertaken for an FPGA-based emulation platform, and illustrate our automatic configuration scheme for different devices.

LEO Implementation. We implement all modules in Figure 6.8 in C/C++ and integrate LEO with MLNX_OFED-5.4-3.0.3.0, CUDA 12.4, NCCL 2.20.5, and DOCA 2.2.2. Specifically, we implement the config parser and automatic configuration as tools for developers to facilitate offline communication configuration, and the other modules as daemon processes during runtime. We create the 5 QPs in userspace for ease of deployment and to avoid kernel syscall overhead. For the timestamp service, we set flags of CQs for hardware timestamping [23], and record the software timestamps when enqueuing and dequeuing descriptors. To calculate the time interval, we enable PTP in NIC and convert HCA clock to ns using `mlx5dv_ts_to_ns` before calculation. While the timestamp is 64-bit, we use a 32-bit value to represent the time interval of remote delay to reduce the message header overhead. Additionally, we implement the LP algorithm in the path selector using `CPLEX` [118] and the heuristic algorithm purely in C++, both running on separate threads for online path selection.

FPGA-based Emulation Platform. To verify the genericity of LEO on different hardware models, we implement an FPGA-based hardware platform that contains a Xilinx VU35P FPGA [11] with a PCIe Gen3x16 interface and a 100 Gbps Ethernet port, and implement the logic of both the accelerator and RDMA NIC on this platform. Overall, we undertake two primary developments:

- Improving the efficiency of PCIe P2P. A conventional MMIO from host to accelerator offers a maximum capacity of 64 bytes per operation due to CPU limitation [3], thereby compromising the I/O efficiency involved in transferring bulk data in PCIe P2P scenario. Thus, we design the corresponding logic to support arbitrary bit read/write operations for PCIe P2P, bypassing CPU.

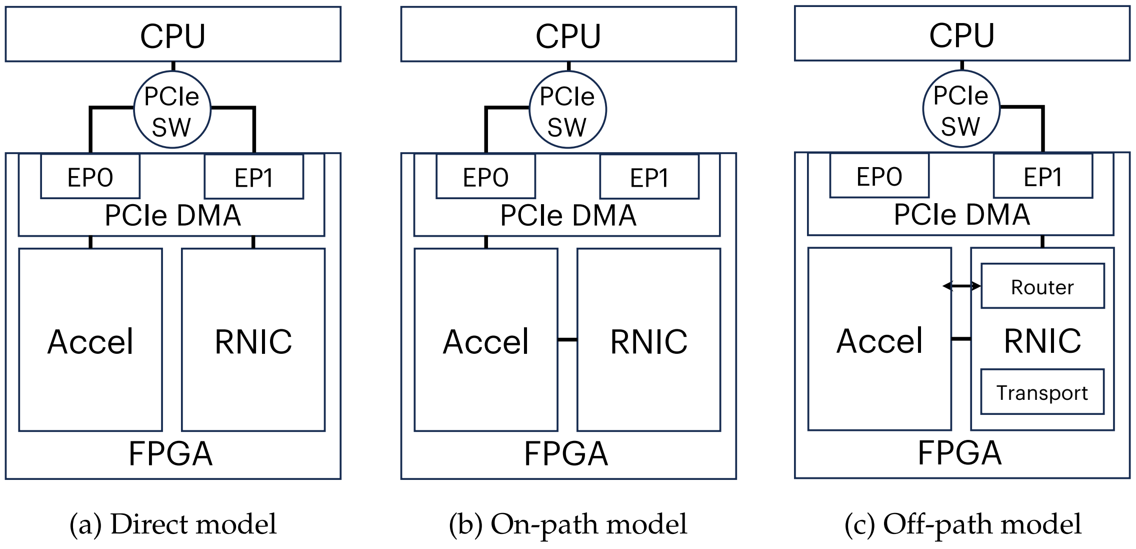


Figure 6.9: Hardware platform architecture.

- Emulating hardware model. We modify the transmission logic between modules in FPGA to emulate different hardware models. As shown in Figure 6.9, for the direct model, we isolate the accelerator and NIC from each other and allow their associated end-points (EPs) to connect with a common PCIe switch. For the on-path model, we connect the accelerator and NIC via AXI and only allow the accelerator-associated EP0 to interconnect with the CPU. For the off-path model, we only open the EP1 channel and add a router within RNIC to allow packet routing to/from the accelerator.

Modules	LUT	Registers	BRAM	URAM
PCIe P2P	3.3%	1.66%	3.01%	5%
Router	0.12%	0.09%	3.95%	0%
EBS lookup functions	4.8%	3.8%	12.9%	8.5%

Table 6.3: Resource usage of hardware platform in Xilinx VU35P FPGA board.

We list the detailed hardware resource consumption in Table 6.3. Note that the EBS lookup functions are developed for emulating three lookup functions in EBS which consumes on-chip resource, while for KVS, its index lookup consumes no on-chip resource as we store the large KV table (20M kv-pairs) in FPGA HBM (§6.4.1).

Automatic Communication Configuration. To free developers from the burden of complex configuration, LEO provides automatic communication configuration across devices with wrappers provided by developers. For general PCIe devices such as FPGA and RNIC, the automatic configuration process is as follows. First, the FPGA memory is mapped to the PCIe BAR space when empowering the device. Then LEO registers the corresponding BAR space and host memory as different memory regions (MRs) in RNIC, along with their virtual-to-physical address translation tables or function hook. Finally, LEO establishes inter-host communication following the classical RDMA initialization manner and exchanges memory region information between both sides to enable direct data transmission between local and remote devices. As this scheme strictly adheres to PCIe P2P and RDMA protocols [10,17], it is general for devices, *e.g.*, we enforce the scheme to configure BlueField with our wrappers using DOCA APIs. Moreover, we extend this scheme to support multiple accelerators case such as GPU-direct-FNIC with an additional role configuration. Specifically, LEO initially designates one of the accelerators as the master for activating direct transmission, and the other as the slave, *e.g.*, FPGA as master and GPU as slave, while keeping the other processes unchanged. Consequently, developers can facilitate direct transmissions between both intra- and inter-host devices without requiring expertise in system configuration.

6.4 Evaluation

In this section, we evaluate LEO's performance to answer the following questions:

- **How generic and efficient is LEO against the state-of-the-art frameworks when processing each application?** We show that LEO supports all types of MINC applications across different testbeds listed in Table 6.4, and can achieve 1.2–4.7× better performance than state-of-the-art frameworks (§6.4.2).

- **How effective is LEO in terms of its component and overhead?** We show that LEO's components effectively improves the performance of MINC communication (§6.4.3). Specifically, the multi-path optimization reduces the median latency of MINC communication by 1.3×. Besides, we also demonstrate that LEO introduces negligible CPU overhead and low processing time (§6.4.3).

Application	Testbed	HW Model	Baseline
KVS	F-NIC	Off-path	DrTM-KV [169]
DFS	BF-3	Off-path	DOCA [13]
EBS	F-NIC	On-path	Solar [119], iPipe [111]
DML	G+F-NIC	Direct	NCCL [7]

Table 6.4: Representative MINC applications with their employed testbed configurations, hardware models, and baselines.

6.4.1 Experimental Setup

Testbed configurations. We run our experiments using three testbed configurations: F-NIC, G+F-NIC, and BF-3. All testbeds use the same topology, *i.e.*, two servers directly connected with each other, but with different devices:

- F-NIC adopts Intel servers with 8-core Xeon Silver 4110 CPU, 192GB memory, PCIe 3.0, 100GE cables, and Xilinx VU35P FPGAs.
- G+F-NIC employs the same servers as F-NIC with the addition of NVIDIA A100 GPUs connected.
- BF-3 uses Intel servers with 32-core Xeon Silver 4309Y CPU, 512GB memory, PCIe 4.0, 200GE cables, and NVIDIA BlueField-3.

Case studies & baselines. To extensively evaluate LEO, we implement four representative MINC applications running on three types of hardware models and compare them with state-of-the-art baselines, as listed in Table 6.4.

- Key-value store: In-memory disaggregated key-value stores are widely deployed in modern data centers [168, 169]. We implement KVS by executing the index lookup in FPGA and facilitating one READ to retrieve the index from FPGA and another WRITE/READ for the value in host memory. We compare LEO with DrTM-KV [169] over the same off-path F-NIC testbed.
- Distributed file system: File replication is a key pillar in DFS and typically offloads file compression to an accelerator for processing [168]. We use DOCA [13] as the baseline

which compresses/decompresses 1MB file with 256KB IO chunks and a 51.8% compression ratio, and compare with LEO using the same off-path BF-3 testbed.

- Elastic block storage: Elastic block storage is a fundamental storage service for cloud providers [119]. As Solar [119] is not open-sourced, we implement its offloaded QoS, block, and address lookup functions in FPGA to emulate read requests[4]. iPipe [111] is a customized on-path smartNIC framework that supports dynamic function placement between the host CPU and smartNIC. We emulate it by first profiling the base JCT of offloading all functions in FPGA and then online invoking functions in the CPU to emulate actor migration[5] when current JCT is larger than base JCT and vice versa. We conduct these experiments using the same on-path F-NIC testbed.

- Distributed machine learning: Distributed machine learning has been deployed extensively in modern data centers [7, 9, 80, 156]. We use only the RNIC module in FPGA and configure GPUDirect-FNIC with automatic configuration. We compare LEO with NCCL [7] over the same direct G+F-NIC testbed.

In all experiments, we add delays at the requester NIC to simulate network variation between hosts [190], with the value ranging from 3 to 9µs according to [189].

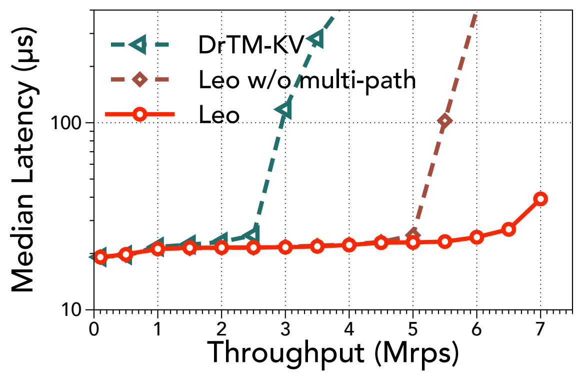
Metrics. We report the median latency under different request rates for latency-intensive key-value stores and throughput for other throughput-intensive applications.

6.4.2 End-to-end Application Performance

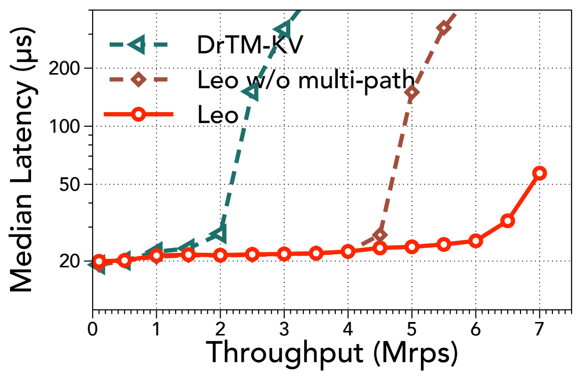
Key-value store. We use YCSB-C and B [50] as our workloads, as used in prior work [169]. The YCSB-C workload only contains the `GET` request. As shown in Figure 6.10a, we observe that the LEO counterpart without multi-path support starts to increase its median latency when system workload is above 5Mrps, outperforming the CPU-based DrTM-KV by $\sim$1.5$\times$. The reason for the higher sustainable throughput is attributed to the efficient computation at accelerators. LEO outperforms both its variant and DrTM-KV by delivering stably low latency and higher sustainable throughput. Specifically, LEO

[4]The SEC and CRC parts in Solar are handled by RDMA transport.

[5]We omit the procedure of actor migration for ease of emulation, but it can actually degrade performance.



(a) YCSB-C workload.



(b) YCSB-B workload.

Figure 6.10: Performance of key-value store. LEO delivers as much as 2× and 3× sustainable throughput compared to the baselines, respectively.

delivers 1.2–2× higher sustainable throughput over its counterparts, as its system demonstrates higher computation capacity by introducing both CPU and accelerator for handling the incoming request streams. Meanwhile, LEO exhibits stably low median latency with the increasing of the offered load. The reason comes from its careful path-selection algorithm, which helps LEO to balance the load among multiple computation sources and mitigate the potential temporal overload.

For the YCSB-B workload, it contains both `GET` and `PUT` requests. For the `PUT` request, it can only be processed by the CPU, since there are currently no sophisticated mechanisms for maintaining data consistency at the accelerator. The scheduling in this workload is more complex as the placement constraints and the uniform workload regarding execution times. As shown in Figure 6.10b, both LEO and its variant deliver as much as 3× higher throughput than DrTM-KV, while LEO achieves the highest throughput. The reason aligns with the case in Figure 6.10a. This experiment further demonstrates the benefits of LEO's incorporation of CPU and accelerator and effective scheduling with balancing load among multiple paths.

Distributed file system. We conduct the experiments by compressing the 1MB file at the requester and write to the responder for decompression. Figure 6.11 shows the throughput with an increasing number of cores on both sides. We find that DOCA performs the worst because its compression time is longer than that of the CPU, *i.e.*, 2.18ms vs 0.89ms, due to the fact that the compression firmware on BF-3 is less powerful than a single core at

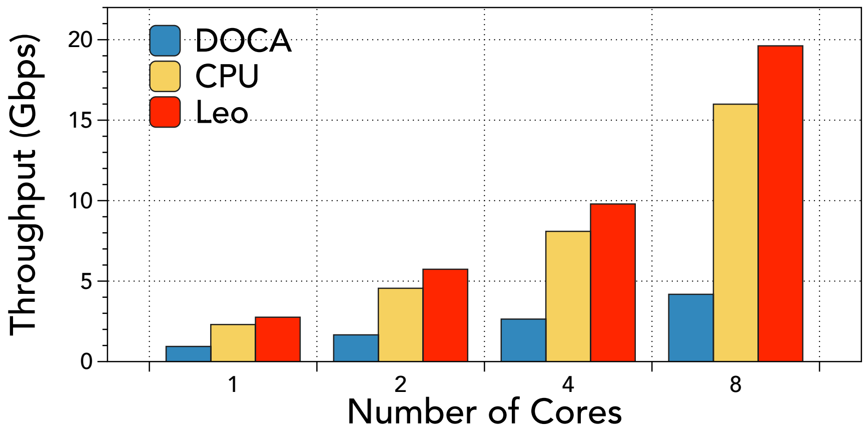
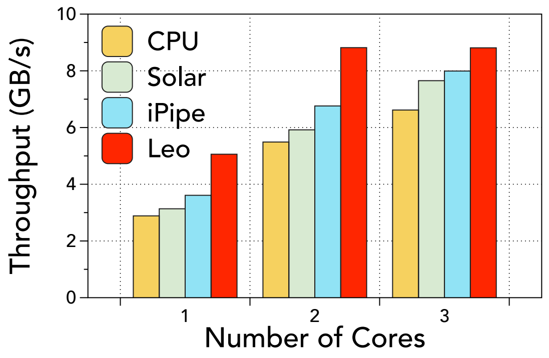


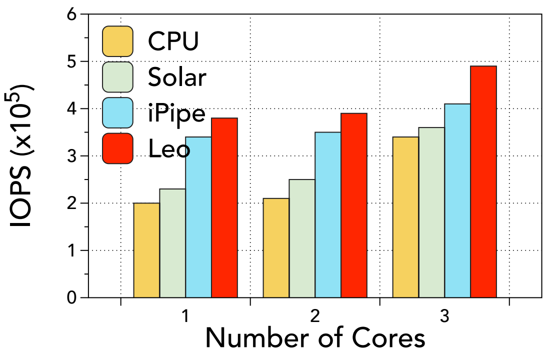
Figure 6.11: Performance of distributed file system. LEO outperforms baselines by 1.19-4.69× as the number of cores increases.

the server. When increasing the number of cores, both approaches show improved performance due to better decompression pipeline at the responder. Overall, LEO outperforms DOCA and CPU by 1.19–4.69× as the number of clients increases. The enhancement is attributed to LEO's effective utilization of hybrid resources.

Elastic block storage. We use the same experiment setting of [119] and compare the 64KB throughput and 4KB IOPS (I/O per second) of LEO with CPU, Solar, and iPipe.



(a) Throughput of 64KB read.



(b) IOPS of 4KB read.

Figure 6.12: Performance of elastic block storage. LEO achieves as much as 1.75× and 1.91× better throughput and IOPS compared to baselines respectively.

The results shown in Figure 6.12 reveal that Solar achieves better performance than CPU-based solution, as it reduces the processing latency via dataplane offloading. iPipe is better for migrating some functions to CPU according to the online end-to-end JCTs. It achieves better throughput and IOPS by 1.04–1.25× and 1.19–1.71×, respectively, than

CPU and Solar with different number of cores. However, it performs worse than LEO as its migration signal, end-to-end JCT, can be affected by network variation. The fluctuation may result in a false migration from a good offloading placement to a worse one. Besides, we remind readers that we did not emulate the actor migration process, which could result in even worse performance with frequent migrations. Overall, LEO demonstrates the highest performance among all, thanks to its better resource utilization with multi-path and intra-host JCT signal for load balancing. Specifically, with LEO, the throughput of 64KB read increases by 1.14–1.75× given the varying number of CPU cores, while the IOPS of 4KB read also exhibits a rise of 1.1–1.9×.

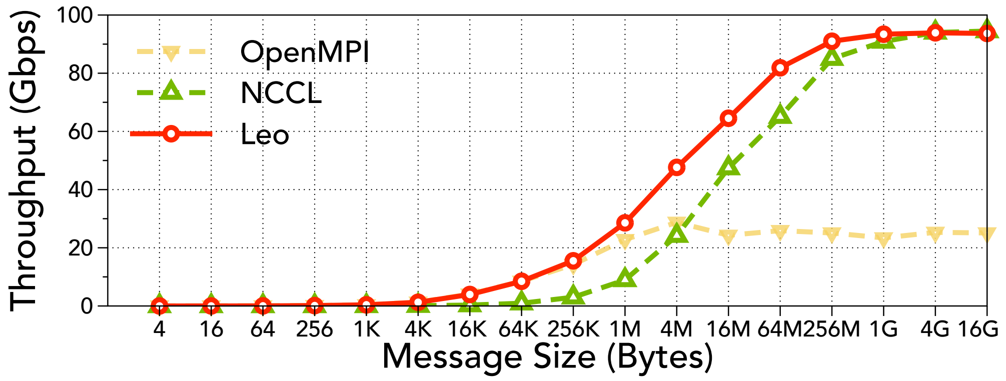
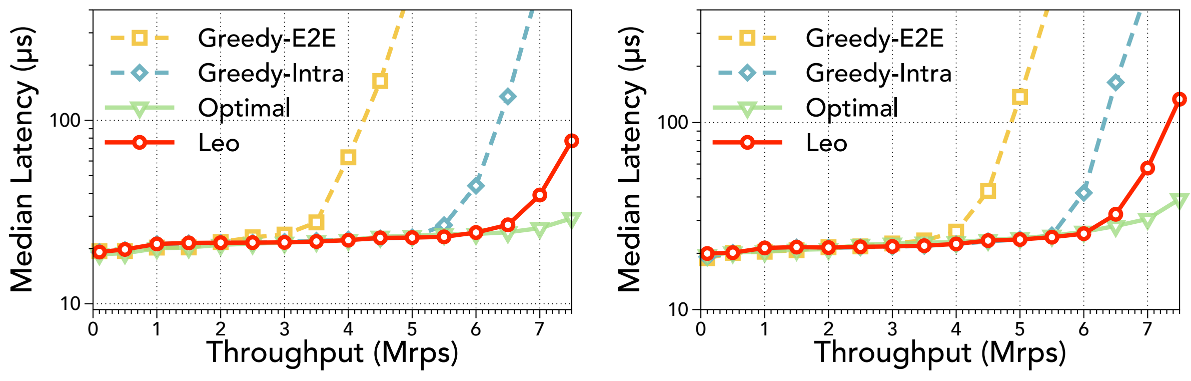


Figure 6.13: Performance of DML allreduce. LEO achieves on average 4× higher throughput than NCCL for message sizes ranging from 4KB to 1GB, and comparable throughput for short and long messages.

Distributed machine learning. Allreduce is a key communication pattern in distributed machine learning, with varying message sizes representing different sizes of gradients and parameters. We compare the allreduce throughput of LEO, NCCL, and OpenMPI with different message sizes. As shown in Figure 6.13, LEO consistently delivers the highest throughput in all message sizes. For large messages ($\geq$1MB), LEO shows a substantial throughput improvement over OpenMPI. Compared to NCCL, LEO's throughput is on average 4.08× higher for messages ranging from 4KB to 1GB. Other message sizes perform similarly. OpenMPI performs worst among all due to the low processing capability of the CPU. The benefit of LEO is attributed to its use of idle compute resources through its multi-path design, allowing it to saturate the link faster than NCCL.

6.4.3 LEO Deep Dive



(a) YCSB-C workload. (b) YCSB-B workload.

Figure 6.14: Effectiveness of LEO multi-path optimization. The performance of the heuristic algorithm is 1.18–2.17× better than greedy algorithms, and close to the optimal solution.

Effectiveness of path selection algorithm. We compare the heuristic algorithm adopted in LEO with two greedy algorithms, *i.e.*, selecting path with minimal end-to-end JCTs (Greedy-E2E) and minimal intra-host JCTs (Greedy-Intra), as well as the optimal LP algorithm in the key-value store application. We do not predict the request sequence but directly pre-record the history of the request sequence and solve the LP problem offline for each step to obtain the performance of the optimal solution. The results are shown in Figure 6.14. We observe that Greedy-E2E performs the worst as it can misselect paths due to network fluctuation between hosts. Greedy-Intra performs better since it only uses the intra-host JCT for selection. However, this algorithm operates in a greedy manner but overlook the temporal queuing delay effect, *i.e.*, the communication toll, caused by the current path decision (similar to k=1 algorithm illustrated in §6.2.2), and therefore performs poorly over a horizon of requests. LEO demonstrates 1.18–2.17× performance gain over the two greedy algorithms, stemming from both the intra-host JCT to reflect path status and the consideration of communication toll. Besides, we also observe that LEO achieves comparable performance to the optimal solution, which knows the whole request sequence in advance. This observation reveals the effectiveness of the heuristic algorithm in LEO.

LEO framework overhead. We seek to understand the framework overhead of LEO. We measure the CPU utilization of the heuristic algorithm, the processing time of algorithms and analyze the header overhead. We find that LEO can run the heuristic algorithm for µs-level applications, with an average cost of 10.3% CPU utilization. We also report the path selection time for one request. Specifically, the LP solver takes as much as 2.4ms, and the heuristic algorithm consumes only $\sim$1.5µs, demonstrating its applicability in real-time path selection. For the message header, it takes up as much as 4 bytes for each response message. The overhead depends on the response size, ranging from 1.6×10^{-5} for DFS with 256KB chunks to 5.9% for KVS with 64B values in the case studies.

6.5 Discussion

Dataflows with multiple accelerators. While we focus on the MINC communication with only a single accelerator in this paper, we claim that LEO can be easily extended to support multiple accelerators. Specifically, the automatic configuration (§6.3) supports the case of multiple accelerators. Besides, the multi-path optimization (§6.2.2) is also applicable to multiple accelerators, *i.e.*, adding the compute time of extra accelerators to the corresponding path.

Multi-tenancy. LEO does not yet support multi-tenancy. This is because MINC applications are usually processed solely on the accelerator for high performance. To support the multi-tenancy scenario, LEO should be able to support different scheduling policies for fairly sharing the accelerator among different applications. Besides, the multi-path optimization should be maintained by the infrastructure maintainers for a global view among applications. We leave the support of multi-tenancy as future work.

Deployment at scale. At present, LEO supports only single connections and is not yet scalable to large-scale clusters involving multiple concurrent connections. To enable high performance in such environments, significant modifications are required in two key areas: (1) the communication abstractions, particularly hardware abstractions, and (2) the multi-path optimization. The hardware abstractions can be enhanced by incorporating averaged performance metrics, while the multi-path optimization should be managed with a global perspective by infrastructure maintainers, as discussed above. Expanding LEO for large-scale deployments presents a promising direction for future research.

6.6 Related work

Network-software co-design of INC. There have been several systems with network-software co-design for INC, as illustrated in §6.1.1. These frameworks are usually designed for specific applications, and are not flexible enough to support the development of other applications.

PINC frameworks. There have been several frameworks proposed recently to facilitate PINC application development. NetRPC [193] advocates an RPC programming model for software developers to describe their PINC applications. Lyra [62] offers a one-big-pipeline abstraction to allow developers to express their intent with simple statements. ClickINC [174] designs a unified and automated workflow for developers to embrace PINC programming and deployment. These works are generic to serve PINC hardware rather than MINC hardware.

MINC frameworks. iPipe [111] and Floem [131] are two representative frameworks for MINC applications. iPipe proposes an actor-based framework to automate fine-grained actor migration between CPU and on-path smartNICs. Floem is a smartNIC compiler that eases the developers' programming effort. However, these frameworks are customized for smartNICs only, but do not support other PCIe devices such as GPUs and FPGAs. Moreover, they either do not support hybrid resource utilization or perform poor load balancing when inter-host network fluctuates (§6.4.2).

Hardware-controlled communication interface. PANIC [106] is a high-performance NIC design that supports diverse offloading functions for multi-tenancy. FpgaNIC [165] allows GPU to directly drive NIC for fast processing. ARK [76] proposes a GPU-controlled hardware DMA engine that eliminates the I/O overhead on GPU cores. FlexDriver [56] builds an on-accelerator hardware module to support the direct drive of the accelerator upon NICs. These optimized hardware interfaces are orthogonal to LEO and can be integrated into LEO.

6.7 Conclusion

This paper presents LEO, a generic and efficient communication framework for MINC applications. LEO makes the following major contributions: i) proposing generic abstractions for both application and hardware; ii) designing a built-in multi-path communication optimization to enhance communication efficiency; iii) exposing unified APIs and wrappers to simplify programming for developers and support automatic communication configuration. We have implemented a prototype of LEO and evaluated it extensively with four case studies over three testbeds. Our results confirm that LEO is both generic and efficient for MINC applications. We hope LEO could serve as a stepping stone for the development of MINC applications and inspire research for MINC.

CHAPTER 7

CONCLUSION AND FUTURE WORKS

7.1 Conclusion and Discussion

7.1.1 Conclusion

My PhD research centers around one question: *How to build communication-efficient distributed training systems to address the varied communication issues encountered during end-to-end distributed training?* This dissertation delineates my research efforts on building communication-efficient distributed training systems through paradigm-specific optimizations for the model training stage and hardware-accelerated optimizations for the data and model management stages. The resulting optimized systems are full-stack solutions tailored for hardware-accelerated GNN and LLM training.

- For the sampling-based GNN training paradigm, we optimize the graph sampling algorithm with DGS (Chapter 3). DGS is a communication-efficient graph sampling framework for distributed GNN training. Its key idea is to reduce network communication cost by sampling neighborhood information based on the locality of the neighbor nodes in the cluster, and sampling data at both node and feature levels. As a result, DGS strikes a balance between communication efficiency and model accuracy, and integrates seamlessly with distributed GNN training systems.
- For the full-graph GNN training paradigm, we optimize the parallelization strategy with G3 (Chapter 4). G3 incorporates GNN hybrid parallelism to scale out full-graph training with meticulous peer-to-peer intermediate data sharing, and accelerates the training process by balancing workloads among workers through locality-aware iterative partitioning, and overlapping communication with computation through a multi-level pipeline scheduling algorithm. Although initially tailored for GNN training, we believe the fundamental principle of peer-to-peer sharing data in hybrid parallelism can be generalized to other training tasks.

- For the LLM training paradigm, we orchestrates all coflow types at the cluster level during LLM training with Hermod (Chapter 5). Hermod employs the insight that coflows are characterized by three model factors, and designs (1) model factor-driven inter-coflow scheduling to align with the LLM training DAG, and (2) matching-based intra-coflow scheduling to maximize transmission rates for high-priority coflows. Extensive evaluations validate that Hermod achieves its design goals. We believe our revisit of coflow scheduling in LLM training will inspire further LLM training optimizations.

- To enhance communication performance in distributed services, we integrate hardware acceleration with LEO (Chapter 6), a hardware-accelerated communication framework for distributed services. LEO offers 1) a communication path abstraction to describe various distributed services with predictable communication performance across edge accelerators, 2) unified APIs and wrappers to simplify programming experience with automatic communication configuration, and 3) a built-in multi-path communication optimization strategy to enhance communication efficiency. We believe LEO will serve as a stepping stone for the development of hardware-accelerated distributed services in distributed training systems.

7.1.2 Discussion

System deployment. All of these optimized systems require code modifications to distributed training frameworks and libraries such as PyTorch, Megatron, DGL, and IB verbs. Fortunately, DGS, G3, and Hermod can be readily deployed using our pre-configured Docker containers[1] [26, 27]. However, LEO necessitates hardware-level modifications for data movement operations, making its deployment more complex. We believe that the optimized systems presented in this dissertation will serve as a foundation for further research on communication-efficient distributed training.

Adapting methodologies across systems. We envision that the methodologies developed in DGS, G3, and Hermod can be extended to complement one another. For example, G3 incorporates intermediate peer-to-peer communication among workers, which can be modeled as coflows. The coflow scheduling methodology from Hermod can be ap-

[1]The Docker image for Hermod will be made publicly available upon publication.

plied to G3 by formulating the GNN training DAG and scheduling coflows accordingly. Likewise, the multi-level pipeline scheduling approach in G3 can optimize fine-grained request scheduling in the hardware accelerators in LEO. Unifying and extending these methodologies across different systems enables the development of more communication-efficient distributed training systems.

7.2 Future Directions

Network Transport for Scale-up Networks. Scale-up networks, defined as network architectures within computer clusters, are designed to maximize intra-cluster communication bandwidth while minimizing latency. These networks have become indispensable in the AI computing era, largely driven by the exponential growth in demand for ultra-high interconnect bandwidth, reaching terabits per second (Tbps). Traditionally, these networks utilize simple load/store primitives to achieve lower latency than DMA [28,29], along with fine-grained compute-communication overlap to reduce the context-switching overhead between CPUs and GPUs. However, scalability remains a challenge due to limited chip buffer capacity. Unlike DMA, which stores only data indices in IO-Dies, load/store primitives store complete data to prevent overwriting by subsequent instructions and to enable retransmission when necessary. While this approach improves data integrity, it imposes greater memory demands, exacerbating scalability limitations. Furthermore, the absence of effective congestion control and load balancing mechanisms in these networks can degrade performance under heavy traffic. Addressing these limitations requires the development of next-generation network transport that integrates reliability, congestion control, and load balancing for scalable scale-up networks.

Error-tolerant Data Loading. Existing works primarily focus on the core training process, but neglect the potential issues faced in other necessary processes such as data loading. However, prior studies [70] have shown that data loading can account for up to 29.5% of the total training time, leading to significant GPU idling. Moreover, in LLM training, the tremendous volume of data for loading can be error-prone. This is due to the fact that data loading relies on distributed cache to speed up the loading process, requiring sophisticate and resilient operations such as cache update coordination, etc. These operations are data dependent and hence instable across training jobs. In this way, an error-tolerant data loading is worth investigating to improve the robustness of LLM training.

CHAPTER 8

PHD'S PUBLICATIONS

Conference Publications

- Harmonia: A Unified Framework for Heterogeneous FPGA Acceleration in the Cloud.

  Luyang Li, Heng Pan, **Xinchen Wan**, Kai Lv, Zilong Wang, Qian Zhao, and Feng Ning, Qingsong Ning, Shideng Zhang, Zhenyu Li, Layong Luo, Gaogang Xie. In **ASPLOS 2025**.

- A Generic and Efficient Communication Framework for Message-level In-Network Computing.

  Xinchen Wan, Luyang Li, Han Tian, Xudong Liao, Xinyang Huang, Chaoliang Zeng, Zilong Wang, Xinyu Yang, Ke Cheng, Qingsong Ning, Guyue Liu, Layong Luo, and Kai Chen. In **INFOCOM 2025**.

- Design and Operation of Shared Machine Learning Clusters on Campus.

  Kaiqiang Xu, Decang Sun, Hao Wang, Zhenghang Ren, **Xinchen Wan**, Xudong Liao, Zilong Wang, Junxue Zhang, and Kai Chen. In **ASPLOS 2025**.

- Achieving Fairness Generalizability for Learning-based Congestion Control with Jury.

  Han Tian, Xudong Liao, Decang Sun, Chaoliang Zeng, Yilun Jin, Junxue Zhang, **Xinchen Wan**, Zilong Wang, Yong Wang, and Kai Chen. In **EuroSys 2025**.

- Fast, Scalable, and Accurate Rate Limiter for RDMA NICs.

  Zilong Wang, **Xinchen Wan**, Luyang Li, Yijun Sun, Peng Xie, Xin Wei, Qingsong Ning, Junxue Zhang, and Kai Chen. In **SIGCOMM 2025**.

- Astraea: Towards Fair and Efficient Learning-based Congestion Control.

  Xudong Liao, Han Tian, Chaoliang Zeng, **Xinchen Wan**, and Kai Chen. In **EuroSys 2024**.

- Accelerating Neural Recommendation Training with Embedding Scheduling.

  Chaoliang Zeng, Xudong Liao, Xiaodian Cheng, Han Tian, **Xinchen Wan**, Hao Wang, and Kai Chen. In **NSDI 2024**.

- Towards Domain-Specific Network Transport for Distributed DNN Training.

  Hao Wang, Han Tian, Jingrong Chen, **Xinchen Wan**, Jiacheng Xia, Gaoxiong Zeng, and Wei Bai, Junchen Jiang, Yong Wang, Kai Chen. In **NSDI 2024**.

- Scalable and Efficient Full-Graph GNN Training for Large Graphs.

  Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. In **SIGMOD 2023**.

- SRNIC: A scalable architecture for RDMA NICs.

  Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, **Xinchen Wan**, and Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, Tianhao Wang, Weicheng Ling, Kejia Huo, Pingbo An, Kui Ji, Shideng Zhang, Bin Xu, Ruiqing Feng, Tao Ding, Kai Chen, Chuanxiong Guo. In **NSDI 2023**.

- DGS: Communication-Efficient Graph Sampling for Distributed GNN Training.

  Xinchen Wan, Kai Chen, and Yiming Zhang. In **ICNP 2022**.

Journal Publications

- Scalable and Efficient Full-Graph GNN Training for Large Graphs.

  Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. In **PACMMOD**.

Workshop Publications

- Accurate and Scalable Rate Limiter for RDMA NICs.

  Zilong Wang, **Xinchen Wan**, Chaoliang Zeng, and Kai Chen In **APNet 2023**.

- Rat-resilient allreduce tree for distributed machine learning.

  Xinchen Wan, Hong Zhang, Hao Wang, Shuihai Hu, Junxue Zhang, and Kai Chen. In **APNet 2020**.

REFERENCES

[1] OGB Leaderboards. https://ogb.stanford.edu/docs/leader_nodeprop/, 2021.

[2] System V IPC for Python. https://semanchuk.com/philip/sysv_ipc, 2021.

[3] Dma/bridge subsystem for pci express. https://www.amd.com/content/dam/xilinx/support/documents/ip_documentation/xdma/v4_1/pg195-pcie-dma.pdf, 2022.

[4] Intel tofino p4-programmable ethernet switch asic that delivers better performance at lower power. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html, 2023.

[5] Intel vtune profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html, 2023.

[6] Nvidia bluefield networking platform. https://www.nvidia.com/en-us/networking/products/data-processing-unit/, 2023.

[7] Nvidia collective communication library. https://developer.nvidia.com/nccl, 2023.

[8] Nvidia gpu performance. https://www.nvidia.com/en-us/data-center/, 2023.

[9] Nvidia gpudirect. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html, 2023.

[10] Pci-sig specifications. https://pcisig.com/specifications, 2023.

[11] Virtex ultrascale+ fpga. https://www.xilinx.com/products/boards-and-kits/device-family/nav-virtex-ultrascale-plus.html, 2023.

[12] Bytescheduler source code. https://github.com/bytedance/byteps/tree/bytescheduler, 2024.

[13] Doca compress. https://docs.nvidia.com/doca/sdk/doca+compress/index.html, 2024.

[14] Doubling all2all performance with nvidia collective communication library 2.12. https://developer.nvidia.com/blog/doubling-all2all-performance-with-nvidia-collective-communication-libra 2024.

[15] Flexflow source code. https://github.com/flexflow/FlexFlow, 2024.

[16] htsim packet-level simulator. https://github.com/nets-cs-pub-ro/NDP/wiki/NDP-Simulator, 2024.

[17] Infiniband specification. https://www.afs.enea.it/asantoro/V1r1_2_1.Release_12062007.pdf, 2024.

[18] Massively scale your deep learning training with nccl 2.4. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/, 2024.

[19] Megatron-lm moe token dispatcher. https://github.com/NVIDIA/Megatron-LM/blob/main/megatron/core/transformer/moe/token_dispatcher.py#L386, 2024.

[20] Nvidia dgx superpod. https://www.nvidia.com/en-us/data-center/dgx-superpod/, 2024.

[21] Nvme specification. https://nvmexpress.org/specifications/, 2024.

[22] Supporting overlapping collective communication for data parallel. https://github.com/NVIDIA/Megatron-LM/issues/391, 2024.

[23] Time-stamp service. https://docs.nvidia.com/networking/display/mlnxofedv571020/time-stamping#src-2396584992_TimeStamping-time-stampingservice, 2024.

[24] Time-stamping service. https://docs.nvidia.com/networking/display/mlnxofedv473290/time-stamping, 2024.

[25] Configuration guidelines for diffserv service classes. https://datatracker.ietf.org/doc/html/rfc4594, 2025.

[26] Docker image of dgs. https://hub.docker.com/repository/docker/wxcsky/dgs/general, 2025.

[27] Docker image of g3. https://hub.docker.com/repository/docker/wxcsky/g3/general, 2025.

[28] Nvlink and nvlink switch. https://www.nvidia.com/en-sg/data-center/nvlink/, 2025.

[29] Ualink alliance for accelerator interconnects. https://nand-research.com/wp-content/uploads/2024/05/2024-05-30-RN-UALink-1.pdf, 2025.

[30] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proc. USENIX OSDI*, 2016.

[31] Dennis Abts, Garrin Kimmell, Andrew Ling, John Kim, Matt Boyd, Andrew Bitar, Sahil Parmar, Ibrahim Ahmed, Roberto DiCecco, David Han, et al. A software-defined tensor streaming multiprocessor for large-scale machine learning. In *Proc. IEEE/ACM ISCA*, 2022.

[32] Ravichandra Addanki, Peter W. Battaglia, David Budden, Andreea Deac, Jonathan Godwin, Thomas Keck, Wai Lok Sibon Li, Alvaro Sanchez-Gonzalez, Jacklynn Stott, Shantanu Thakoor, and Petar Velickovic. Large-scale graph representation learning with very deep gnns and self-supervision. In *arXiv*, 2021.

[33] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. Sincronia: Near-optimal network design for coflows. In *Proc. ACM SIGCOMM*, 2018.

[34] Alibaba. Euler, 2020.

[35] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. *Proc. ACM SIGCOMM*, 2013.

[36] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proc. ACM WWW*, 2011.

[37] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. ACM WWW*, 2004.

[38] Tommaso Bonato, Abdul Kabbani, Ahmad Ghalayini, Mohammad Dohadwala, Michael Papamichael, Daniele De Sensi, and Torsten Hoefler. Arcane: Adaptive routing with caching and network exploration, 2024.

[39] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Proc. NeurIPS*, 2020.

[40] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. Dgcl: An efficient communication library for distributed gnn training. In *Proc. ACM EuroSys*, 2021.

[41] Jiamin Cao, Yu Guan, Kun Qian, Jiaqi Gao, Wencong Xiao, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. Crux: Gpu-efficient communication scheduling for deep learning training. In *Proc. ACM SIGCOMM*, 2024.

[42] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *Proc. IEEE/ACM MICRO*, 2016.

[43] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv*, 2018.

[44] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *arXiv*, 2015.

[45] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proc. ACM SIGKDD*, 2019.

[46] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *Proc. ACM HotNets*, 2012.

[47] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. *Proc. ACM SIGCOMM*, 2015.

[48] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. *Proc. ACM SIGCOMM*, 2011.

[49] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proc. ACM SIGCOMM*, 2014.

[50] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proc. ACM SoCC*, 2010.

[51] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[52] Matthew T. Dearing and Xiaoyan Wang. Analyzing the performance of graph neural networks with pipe parallelism. 2021.

[53] Fahad R Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized task-aware scheduling for data center networks. *Proc. ACM SIGCOMM*, 2014.

[54] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 2012.

[55] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv*, 2024.

[56] Haggai Eran, Maxim Fudim, Gabi Malka, Gal Shalom, Noam Cohen, Amit Hermony, Dotan Levi, Liran Liss, and Mark Silberstein. Flexdriver: A network driver for your accelerator. In *Proc. ACM ASPLOS*, 2022.

[57] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *JMLR*, 2022.

[58] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv*, 2019.

[59] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking:{SmartNICs} in the public cloud. In *Proc. USENIX NSDI*, 2018.

[60] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *Proc. OSDI*, 2021.

[61] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. Rdma over ethernet for distributed training at meta scale. In *Proc. ACM SIGCOMM*, 2024.

[62] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proc. ACM SIGCOMM*, 2020.

[63] Andrew Gibiansky. Bringing hpc techniques to deep learning. *Baidu Research, Tech. Rep.*, 2017.

[64] Yuntao Gui, Yidi Wu, Han Yang, Tatiana Jin, Boyang Li, Qihui Zhou, James Cheng, and Fan Yu. Hgl: accelerating heterogeneous gnn training with holistic representation and optimization. In *Proc. IEEE SC*, 2022.

[65] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv*, 2025.

[66] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proc. NeurIPS*, 2017.

[67] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *Proc. MLSys*, 2019.

[68] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. IEEE CVPR*, 2016.

[69] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proc. ACM WWW*, 2017.

[70] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. Characterization of large language model development in the datacenter. In *Proc. USENIX NSDI*, 2024.

[71] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv*, 2020.

[72] Xia Hu, Lei Tang, Jiliang Tang, and Huan Liu. Exploiting social relations for sentiment analysis in microblogging. In *Proc. ACM WSDM*, 2013.

[73] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. Understanding and bridging the gaps in current gnn performance optimizations. In *Proc. ACM PPoPP*, 2021.

[74] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. Adaptive sampling towards fast graph representation learning. *arXiv*, 2018.

[75] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Ef-

ficient training of giant neural networks using pipeline parallelism. *Proc. NeurIPS*, 2019.

[76] Changho Hwang, KyoungSoo Park, Ran Shu, Xinyuan Qu, Peng Cheng, and Yongqiang Xiong. {ARK}:{GPU-driven} code execution for distributed deep learning. In *Proc. USENIX NSDI*, 2023.

[77] Anand Jayarajan, Jinliang Wei, Garth A. Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. In *Proc. MLSys*, 2019.

[78] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proc. MLSys*, 2020.

[79] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv*, 2024.

[80] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed {DNN} training in heterogeneous gpu/cpu clusters. In *Proc. USENIX OSDI*, 2020.

[81] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *Proc. USENIX NSDI*, 2024.

[82] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. {NetChain}:{Scale-Free}{Sub-RTT} coordination. In *Proc. USENIX NSDI*, 2018.

[83] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proc. ACM SOSP*, 2017.

[84] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. Tpu v4: An

optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proc. IEEE/ACM ISCA*, 2023.

[85] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E Leiserson, and Jie Chen. Accelerating training and inference of graph neural networks with fast sampling and pipelining. In *Proc. MLSys*, 2022.

[86] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv*, 2020.

[87] Anuradha Karunarathna, Dinika Senarath, Shalika Madhushanki, Chinthaka Weerakkody, Miyuru Dayarathna, Sanath Jayasena, and Toyotaro Suzumura. Scalable graph convolutional network based link prediction on a distributed graph database server. In *Proc. IEEE CLOUD*, 2020.

[88] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 1998.

[89] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proc. IEEE SC*, 1998.

[90] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *Proc. ACM SOSP*, 2021.

[91] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv*, 2016.

[92] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proc. MLSys*, 2023.

[93] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Proc. NeurIPS*, 2012.

[94] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 1955.

[95] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proc. ACM SIGCOMM*, 2020.

[96] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M Swift. Atp: In-network aggregation for multi-tenant learning. In *Proc. USENIX NSDI*, 2021.

[97] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proc. ACM SOSP*, 2017.

[98] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proc. ACM SIGCOMM*, 2016.

[99] Guohao Li, Matthias Müller, Bernard Ghanem, and Vladlen Koltun. Training graph neural networks with 1000 layers. In *Proc. ICML*, 2021.

[100] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. Accelerating distributed {MoE} training and inference with lina. In *Proc. USENIX ATC*, 2023.

[101] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proc. USENIX OSDI*, 2014.

[102] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv*, 2020.

[103] Wenxue Li, Xiangzhou Liu, Yuxuan Li, Yilun Jin, Han Tian, Zhizhen Zhong, Guyue Liu, Ying Zhang, and Kai Chen. Understanding communication characteristics of distributed training. In *Proc. ACM APNet*, 2024.

[104] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G Andersen, and Michael J Freedman. Be fast, cheap and in control with {SwitchKV}. In *Proc. USENIX NSDI*, 2016.

[105] Xudong Liao, Yijun Sun, Han Tian, Xinchen Wan, Yilun Jin, Zilong Wang, Zhenghang Ren, Xinyang Huang, Wenxue Li, Kin Fai Tse, et al. mfabric: An efficient and scalable fabric for mixture-of-experts training. *arXiv*, 2025.

[106] Jiaxin Lin, Kiran Patel, Brent E Stephens, Anirudh Sivaraman, and Aditya Akella. Panic: A high-performance programmable nic for multi-tenant networks. In *Proc. USENIX OSDI*, 2020.

[107] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proc. ACM SoCC*, 2020.

[108] Juncai Liu, Jessie Hui Wang, and Yimin Jiang. Janus: A unified distributed training framework for sparse mixture-of-experts models. In *Proc. ACM SIGCOMM*, 2023.

[109] Kefei Liu, Zhuo Jiang, Jiao Zhang, Shixian Guo, Xuan Zhang, Yangyang Bai, Yongbin Dong, Feng Luo, Zhang Zhang, Lei Wang, et al. R-pingmesh: A service-aware roce network monitoring and diagnostic system. In *Proc. ACM SIGCOMM*, 2024.

[110] Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong, Lizhuang Tan, Tian Pan, and Tao Huang. Hostping: Diagnosing intra-host network bottlenecks in {RDMA} servers. In *Proc. USENIX NSDI*, 2023.

[111] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proc. ACM SIGCOMM*. 2019.

[112] Shuo Liu, Qiaoling Wang, Junyi Zhang, Wenfei Wu, Qinliang Lin, Yao Liu, Meng Xu, Marco Canini, Ray CC Cheung, and Jianfei He. In-network aggregation with transport transparency for distributed training. In *Proc. ACM ASPLOS*, 2023.

[113] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing. In *arXiv*, 2021.

[114] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *Proc. USENIX FAST*, 2019.

[115] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. {Multi-Path} transport for {RDMA} in datacenters. In *Proc. USENIX NSDI*, 2018.

[116] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: parallel deep neural network computation on large graphs. In *Proc. USENIX ATC*, 2019.

[117] Kshiteej Mahajan, Ching-Hsiang Chu, Srinivas Sridharan, and Aditya Akella. Better together: Jointly optimizing {ML} collective scheduling and execution planning using {SYNDICATE}. In *Proc. USENIX NSDI*, 2023.

[118] CPLEX User's Manual. Ibm ilog cplex optimization studio. *Version*, 1987.

[119] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In *Proc. ACM SIGCOMM*, 2022.

[120] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *Proc. USENIX OSDI*, 2021.

[121] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and

Ion Stoica. Ray: A distributed framework for emerging AI applications. In *Proc. USENIX OSDI*, 2018.

[122] Hesham Mostafa. Sequential aggregation and rematerialization: Distributed full-batch training of graph neural networks on large graphs. In *arXiv*, 2021.

[123] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proc. IEEE/ACM ISCA*, 2022.

[124] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proc. ACM SOSP*, 2019.

[125] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proc. IEEE SC*, 2021.

[126] OpenAI. Gpt-4 technical report. *ArXiv*, 2023.

[127] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[128] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 2009.

[129] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. Sancus: sta le n ess-aware c omm u nication-avoiding full-graph decentralized training in large-scale graph neural networks. In *Proc. VLDB Endow.*, 2022.

[130] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proc. ACM SOSP*, 2019.

[131] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas E Anderson. Floem: A programming system for nic-accelerated network applications. In *Proc. USENIX OSDI*, 2018.

[132] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, et al. Alibaba hpn: A data center network for large language model training. In *Proc. ACM SIGCOMM*, 2024.

[133] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *Proc. ICML*, 2021.

[134] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. *Proc. ACM SIGCOMM*, 2011.

[135] Sudarsanan Rajasekaran, Sanjoli Narang, Anton A Zabreyko, and Manya Ghobadi. Mltcp: Congestion control for dnn training. *arXiv*, 2024.

[136] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *Proc. IEEE SC*, 2020.

[137] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proc. ACM SIGKDD*, 2020.

[138] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *Proc. USENIX NSDI*, 2021.

[139] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In *Proc. ACM SOSP*, 2021.

[140] Marco Serafini. Scalable graph neural network training: The case for sampling. *SIGOPS Oper. Syst. Rev.*, 2021.

[141] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv*, 2018.

[142] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. {TACCL}: Guiding collective algorithm synthesis using communication sketches. In *Proc. USENIX NSDI*, 2023.

[143] Vinay Shankarkumar, Laurent Montini, Tim Frost, and Greg Dowd. Precision time protocol version 2 (ptpv2) management information base. Technical report, 2017.

[144] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv*, 2017.

[145] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv*, 2019.

[146] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. Dorylus: affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads. In *Proc. USENIX OSDI*, 2021.

[147] Alok Tripathy, Katherine Yelick, and Ayd n Buluç. Reducing communication in graph neural network training. In *Proc. IEEE SC*, 2020.

[148] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Proc. NeurIPS*, 2017.

[149] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv*, 2017.

[150] Borui Wan, Mingji Han, Yiyao Sheng, Zhichao Lai, Mofan Zhang, Junda Zhang, Yanghua Peng, Haibin Lin, Xin Liu, and Chuan Wu. Bytecheckpoint: A unified checkpointing system for llm development. *arXiv*, 2024.

[151] Cheng Wan, Youjie Li, Ang Li, Nam Sung Kim, and Yingyan Lin. Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. *Proc. MLSys*, 2022.

[152] Cheng Wan, Youjie Li, Cameron R Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication. In *arXiv*, 2022.

[153] Xinchen Wan, Kai Chen, and Yiming Zhang. Dgs: Communication-efficient graph sampling for distributed gnn training. In *Proc. IEEE ICNP*, 2022.

[154] Xinchen Wan, Luyang Li, Han Tian, Xudong Liao, Xinyang Huang, Chaoliang Zeng, Zilong Wang, Xinyu Yang, Ke Cheng, Qingsong Ning, Guyue Liu, Layong Luo, and Kai Chen. A generic and efficient communication framework for message-level in-network computing. In *Proc. IEEE INFOCOM*, 2025.

[155] Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. Scalable and efficient full-graph gnn training for large graphs. *Proc. ACM SIGMOD*, 2023.

[156] Xinchen Wan, Hong Zhang, Hao Wang, Shuihai Hu, Junxue Zhang, and Kai Chen. Rat-resilient allreduce tree for distributed machine learning. In *Proc. ACM APNet*, 2020.

[157] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. Towards {Domain-Specific} network transport for distributed {DNN} training. In *Proc. USENIX NSDI*, 2024.

[158] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks, 2020.

[159] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. Neutronstar: distributed gnn training with hybrid dependency management. In *Proc. ACM SIGMOD*, 2022.

[160] Weiyan Wang, Cengguang Zhang, Liu Yang, Kai Chen, and Kun Tan. Addressing network bottlenecks with divide-and-shuffle synchronization for distributed dnn training. In *Proc. IEEE INFOCOM*, 2022.

[161] Weiyang Wang, Manya Ghobadi, Kayvon Shakeri, Ying Zhang, and Naader Hasani. Rail-only: A low-cost high-performance network for training llms with trillion parameters.

[162] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. {TopoOpt}: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *Proc. USENIX NSDI*, 2023.

[163] Yiding Wang, Decang Sun, Kai Chen, Fan Lai, and Mosharaf Chowdhury. Efficient dnn training with knowledge-guided layer freezing. In *arXiv*, 2022.

[164] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An adaptive and efficient runtime system for {GNN} acceleration on gpus. In *Proc. USENIX OSDI*, 2021.

[165] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. {FpgaNIC}: An {FPGA-based} versatile 100gb {SmartNIC} for {GPUs}. In *Proc. USENIX ATC*, 2022.

[166] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, et al. {SRNIC}: A scalable architecture for {RDMA}{NICs}. In *Proc. USENIX NSDI*, 2023.

[167] Zilong Wang, Xinchen Wan, Luyang Li, Yijun Sun, Peng Xie, Xin Wei, Qingsong Ning, Junxue Zhang, and Kai Chen. Fast, scalable, and accurate rate limiter for rdma nics. In *Proc. ACM SIGCOMM*, 2024.

[168] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path {SmartNIC} for accelerating distributed systems. In *Proc. USENIX OSDI*, 2023.

[169] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *Proc. USENIX OSDI*, 2018.

[170] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In *Proc. ICML*, 2019.

[171] Kaiqiang Xu, Decang Sun, Hao Wang, Zhenghang Ren, Xinchen Wan, Xudong Liao, Zilong Wang, Junxue Zhang, and Kai Chen. Design and operation of shared machine learning clusters on campus. In *Proc. ACM ASPLOS*, 2025.

[172] Kaiqiang Xu, Xinchen Wan, Hao Wang, Zhenghang Ren, Xudong Liao, Decang Sun, Chaoliang Zeng, and Kai Chen. Tacc: A full-stack cloud computing infrastructure for machine learning tasks. *arXiv*, 2021.

[173] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2019.

[174] Wenquan Xu, Zijian Zhang, Yong Feng, Haoyu Song, Zhikang Chen, Wenfei Wu, Guyue Liu, Yinchao Zhang, Shuxin Liu, Zerui Tian, et al. Clickinc: In-network computing as a service in heterogeneous programmable data-center networks. In *Proc. ACM SIGCOMM*, 2023.

[175] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. Qwen2 technical report. *arXiv*, 2024.

[176] Liang Yao, Chengsheng Mao, and Yuan Luo. Graph convolutional networks for text classification. In *Proc. AAAI*, 2019.

[177] Rui Ye, Xin Li, Yujie Fang, Hongyu Zang, and Mingzhong Wang. A vectorized relational graph convolutional network for multi-relational network alignment. In *Proc. IJCAI*, 2019.

[178] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proc. ACM SIGCOMM*, 2015.

[179] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnnexplainer: Generating explanations for graph neural networks. *Proc. NeurIPS*, 2019.

[180] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proc. ACM SIGKDD*, 2018.

[181] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *arXiv*, 2018.

[182] Jiaxuan You, Bowen Liu, Rex Ying, Vijay Pande, and Jure Leskovec. Graph convolutional policy network for goal-directed molecular graph generation. In *Proc. NeurIPS*, 2018.

[183] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proc. USENIX HotCloud*, 2010.

[184] Chaoliang Zeng, Xudong Liao, Xiaodian Cheng, Han Tian, Xinchen Wan, Hao Wang, and Kai Chen. Accelerating neural recommendation training with embedding scheduling. In *Proc. USENIX NSDI*, 2024.

[185] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. *arXiv*, 2019.

[186] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. Agl: A scalable system for industrial-purpose graph machine learning. *Proc. VLDB Endow.*, 2020.

[187] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In *Proc. USENIX ATC*, 2017.

[188] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. Coda: Toward automatically identifying and scheduling coflows in the dark. In *Proc. ACM SIGCOMM*, 2016.

[189] Junxue Zhang, Wei Bai, and Kai Chen. Enabling ecn for datacenter networks with rtt variations. In *Proc. ACM CoNEXT*, 2019.

[190] Junxue Zhang, Chaoliang Zeng, Hong Zhang, Shuihai Hu, and Kai Chen. Liteflow: towards high-performance adaptive neural networks for kernel datapath. In *Proc. ACM SIGCOMM*, 2022.

[191] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *Proc. NeurIPS*, 2018.

[192] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. {FPGA-Accelerated} compactions for {LSM-based}{Key-Value} store. In *Proc. USENIX FAST*, 2020.

[193] Bohan Zhao, Wenfei Wu, and Wei Xu. {NetRPC}: Enabling {In-Network} computation in remote procedure calls. In *Proc. USENIX NSDI*, 2023.

[194] Yangming Zhao, Kai Chen, Wei Bai, Minlan Yu, Chen Tian, Yanhui Geng, Yiming Zhang, Dan Li, and Sheng Wang. Rapier: Integrating routing and scheduling for coflow-aware data center networks. In *Proc. IEEE INFOCOM*, 2015.

[195] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv*, 2023.

[196] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: Distributed graph neural network training for billion-scale graphs. *arXiv*, 2020.

[197] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: A comprehensive graph neural network platform. *arXiv*, 2019.

[198] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *Proc. USENIX OSDI*, 2016.

[199] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. Layer-dependent importance sampling for training deep and large graph convolutional networks. *arXiv*, 2019.

[200] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, et al. Resiliency at scale: Managing {Google's}{TPUv4} machine learning supercomputer. In *Proc. USENIX NSDI*, 2024.