SIMULATION-DRIVEN EXPLORATION OF RECONFIGURABLE OPTICAL DATA CENTERS FOR MULTI-JOB DISTRIBUTED AI TRAINING

MS (Research) Thesis

By:
Masabattula Teja Nikhil



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY INDORE

May 30, 2026

SIMULATION-DRIVEN EXPLORATION OF RECONFIGURABLE OPTICAL DATA CENTERS FOR MULTI-JOB DISTRIBUTED AI TRAINING

A THESIS

Submitted in partial fulfillment of the
requirements for the award of the degree
of
Master of Science

by
Masabattula Teja Nikhil



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY INDORE

May 30, 2026

INDIAN INSTITUTE OF TECHNOLOGY INDORE

CANDIDATE'S DECLARATION

I hereby certify that the work which is being presented in the thesis entitled **Simulation-Driven Exploration of Reconfigurable Optical Data Centers for Multi-Job Distributed AI Training** in the partial fulfillment of the requirements for the award of the degree of MASTER OF SCIENCE (RESEARCH) and submitted in the DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, Indian Institute of Technology Indore, is an authentic record of my own work carried out during the time period from August, 2024 to May, 2026 under the supervision of Dr. Ranveer Singh, associate professor in IIT Indore. The matter presented in this thesis has not been submitted by me for the award of any other degree of this or any other institute.

Signature with date 04/11/2026
Masabattula Teja Nikhil

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Signature of the Supervisor of
M.S(Research) Thesis

Masabattula Teja Nikhil has successfully given his M.S (Research) Oral Examination held on 04/11/2026

Signature(s) of Chairperson(OEB)
Date: 04/11/2026

Signature of Convener,DPGC
Date:- 11/11/2026

Signature of Thesis Supervisor
Date:- 04/11/2026

Signature of Head of Discipline
Date:- 17-11-2026

Acknowledgement

I am deeply grateful to my MS research advisor, Dr. Ranveer Singh and co-supervisor Dr. Sidharth Sharma, for giving me the opportunity to pursue this thesis and for their constant guidance, and support throughout my research journey. Their mentorship have been invaluable in shaping both my research approach and academic growth. They not only guided me in the right direction but also taught me how to conduct meaningful research by bridging mathematical theory with practical implementation. I am especially grateful for the freedom they provided me to explore ideas independently and develop my own perspective toward research. I sincerely thank Dr. Subhrangsu Mandal for his support and for introducing me to temporal graphs.

I am profoundly grateful to my family for their unconditional love, encouragement, and unwavering support throughout my academic journey. I would especially like to thank my father for always supporting my decisions, encouraging me to pursue what I truly wanted, and standing beside me throughout my journey.

I also thank the members of my thesis committee for their valuable feedback and careful review of this work. Finally, I am grateful to the anonymous reviewers of networking conferences for their constructive comments and suggestions, which significantly improved the quality of this thesis.

I would also like to acknowledge the Department of Computer Science and Engineering, Indian Institute of Technology, Indore, for providing the research environment and computational infrastructure that made this work possible. Finally, I express my deepest gratitude to everyone who supported and encouraged me throughout this journey. Their belief in me has been a constant source of strength and motivation.

Abstract

OCSes are emerging as a key building block of modern AI clusters. Hyperscalers such as Google and Meta are already deploying reconfigurable optical fabrics at scale. However, existing simulators do not model Optical Circuit Switch (OCS)-based interconnects in realistic multi-job AI datacenter environments. This limits the study of distributed AI training under dynamic circuit constraints. In this work, we extend a state-of-the-art distributed DNN training simulator to support multi-job execution and scale-out topologies with OCSes. Our framework models collective communication, circuit reconfiguration latency, time-sliced optical connectivity, and job-level contention. This enables end-to-end exploration of distributed training over reconfigurable fabrics. Using this simulator, we identify key performance challenges in shared OCS clusters, including sensitivity to reconfiguration delay and inefficiencies due to transient connectivity. To address these issues, we propose an Integer Linear Programming (ILP)-based circuit scheduler, an A*-based routing strategy for time-varying optical topologies, and controlled preemption mechanisms for circuit sharing. Our results provide first-order insights into the latency–utilization–performance trade-offs in OCS-enabled AI infrastructure and offer concrete design guidelines for building scalable, reconfigurable training platforms.

Keywords : Reconfigurable Datacenter Network (R-DCN), Network Simulation, Optical Circuit Switches (OCS), Network Performance Modeling.

Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 2
- 1.2 Contributions and Scope of Thesis . . . 4
- 1.3 Thesis Organization . . . 5

2 Background and Related works **7**
- 2.1 Communication Characteristics of Deep Neural Network (DNN) Training . . . 7
  - 2.1.1 Multi-GPU Training . . . 8
  - 2.1.2 Collective Algorithms . . . 11
  - 2.1.3 Collective Logical Topologies . . . 12
  - 2.1.4 Multi-Dimensional Collectives . . . 13
  - 2.1.5 Synchronization Barriers . . . 19
  - 2.1.6 Datacenter Network Topologies . . . 20
- 2.2 Primer on Reconfigurable Fabrics . . . 22
  - 2.2.1 Reconfiguration Schedule . . . 24
  - 2.2.2 Gaps & Challenges . . . 26
- 2.3 Gaps in Job-Scheduling . . . 26
- 2.4 Summary of Related works and their limitations . . . 27
  - 2.4.1 DNN Training Simulators. . . . 30

3 Reconfigurable DCN Design-Space **32**

4 Proposed Approaches **36**
- 4.1 Demand Matrix Construction . . . 37
- 4.2 Creation of Optical Circuit Schedule . . . 39
  - 4.2.1 Matrix-Decomposition Strategy. . . . 41
- 4.3 Inter-Job Scheduling . . . 44
  - 4.3.1 Job-Pinning Strategy . . . 44
  - 4.3.2 OCS Sharing Policy . . . 45
- 4.4 Multi-Job Modeling . . . 47

4.4.1 Coarse-Grained Preemption 48
4.4.2 Fine-Grained Preemption 49
4.5 Temporal Routing 50
4.5.1 Multi-path Routing Algorithm 51
4.5.2 Heuristic-Routing Algorithm 57

5 Simulator Architecture **60**
5.1 Simulator Design 60
5.1.1 Network Backend 62
5.1.2 The AstraSim Layer 70
5.1.3 The Control Plane 74

6 Experimental Evaluation **78**

7 Conclusions and Future Work **84**

References

List of Figures

2.1 Illustrating the periodic behaviour of DNN training. 8
2.2 Data Parallelism training. The model is replicated across all GPUs and each GPU processes a different shard of the data. 9
2.3 Model Parallelism training. The model is partitioned across GPUs, each holding a portion of the parameters. 10
2.4 Hybrid parallel training. Data parallelism and model parallelism are combined to scale along both dimensions simultaneously. 10
2.5 Logical topologies used in collective communication: **(a)** Ring, **(b)** All-to-All, and **(c)** 2D-Mesh. 12
2.6 Sliced collective communication, where a gradient vector is partitioned into chunks that independently execute the collective operation. 14
2.7 Synchronization barriers at the collective level, enforcing that all GPUs complete a phase before any proceeds to the next. 15
2.8 Compute-communication overlap during pipelined AllReduce. While one chunk is in flight, the next chunk proceeds through computation. 16
2.9 Two dimensional Torus topology with wrap-around links. . 17
2.10 Multi-Dimensional (vs) Simple AllReduce 18
2.11 Synchronization barriers at model level during AI training. 20
2.12 Illustration of Datacenter front-end network and AI backend network [32]. 21
2.13 Rail-Optimized design with OCSes in the Spine Layer. . . 22
2.14 Illustration of difference between a packet switch and reconfigurable OCS switch. 23
2.15 Approximating high-radix topologies using optical circuit switches over a Spine-Leaf architecture. 24
2.16 Illustration of offline reconfigurable schedule. 25

4.1 Job pinning with two OCS switches and four jobs. 45
4.2 Illustration of Coarse-Grained preemption. 49
4.3 Illustration of Fine-Grained preemption 49
4.4 Reconfigurable network as a Temporal-Graph. 51
4.5 Temporal graph example with four nodes and time-varying edge availability. 56
4.6 The three foremost paths: (i) 1st Foremost (blue), (ii) 2nd Foremost (red), (iii) 3rd Foremost (teal). Coloured edges show the path taken; dashed grey edges are alternatives not chosen. 57
4.7 RDCN as a temporal graph 58

5.1 Simulator Architecture. 61
5.2 Temporal Graph Augmentation 69

6.1 JCT versus TSW for a single training job. 79
6.2 JCT (bars, left axis) and reconfiguration overhead (line, right axis) as a function of TSW under a $1\,\mu$s reconfiguration penalty. 80
6.3 JCT per job (stacked bars, left axis) and JCT spread across jobs (lines, right axis) under different preemption strategies (NP, CG, FG) as OCS count scales from 1 to 6. 82
6.4 Average JCT as a function of the routing balance factor α. 82

List of Abbreviations

OCS	Optical Circuit Switch
MEMS	Micro Electro Mechanical System
AI-DCN	AI Datacenter Networks
R-DCN	Reconfigurable Datacenter Network
ToR	Top-of-Rack
GPU	Graphics Processing Unit
DNN	Deep Neural Network
DP	Data Parallelism
MP	Model Parallelism
HP	Hybrid Parallelism
AR	All-Reduce
RS	Reduce-Scatter
AG	All-Gather
NCCL	NVIDIA Collective Communication Library
ILP	Integer Linear Programming

Chapter 1

Introduction

The compute throughput of modern AI accelerators is scaling significantly faster than the network interconnect fabrics that connect them, creating an increasing imbalance between computation and communication capabilities in large-scale AI clusters. As AI workloads continue to grow in scale and model complexity, this imbalance becomes more pronounced because higher compute speeds generate data faster than it can be communicated to other compute nodes, thereby increasing communication demands across/within the cluster. Consequently, the interconnect fabric emerges as a critical bottleneck, placing substantial pressure on modern datacenter infrastructures.

Modern Datacenters, are predominantly built using traditional packet-switched fabrics designed to efficiently support diverse and dynamic cloud traffic patterns, making them unnecessarily complex, energy-intensive, and over-provisioned for the comparatively structured and communication patterns exhibited by large-scale DNN training workloads in an AI-Datacenter.

To address these limitations, reconfigurable optical fabrics based on Optical Circuit Switching have emerged as a promising alternative. Optical Circuit Switches (OCS) follow the principle of **circuit switching** by dynamically changing their internal port connectivities to establish optical circuits between communicating endpoints. As communication demands evolve, these circuits can be reconfigured, allowing the network topology to adapt on demand. Unlike general datacenter traffic, DNN training workloads have a proper structure and are predictable – the same GPUs exchange the same amount of data in every iteration. This nature of DNN training traffic is the key advantage that an OCS can leverage to dynamically adapt the network to serve the network traffic more efficiently than the static **packet-switched** networks. OCS-based fabrics establish direct

high-bandwidth optical paths between communicating endpoints, reducing the overheads associated with multi-hop packet forwarding. Furthermore, their bufferless design and limited packet-processing requirements improve energy efficiency and contribute to better long-term cost efficiency. As a result, OCS-based fabrics have emerged as a promising alternative to conventional electronic switching infrastructures for AI training workloads.

That said, completely replacing existing electronic switching infrastructure would require significant changes to existing datacenter designs and deployment practices. This work therefore considers a hybrid packet-circuit switched network architecture that builds on top of the existing infrastructure by augmenting it with optical circuit switches. However, deploying OCS-based fabrics in multi-job training environments introduces a new set of challenges. periodic reconfigurations add per-switch (circuit-switching) overhead, and co-located jobs contend for circuits in both space and time, complicating fabric scheduling, routing, and sharing of backend network fabric. The resulting interplay between reconfiguration delays, collective communication patterns, and inter-job contention creates complex trade-offs that are difficult to capture analytically.

This motivates the need for a detailed investigation into the performance characteristics of reconfigurable optical fabrics under realistic distributed training workloads and multi-tenant deployment conditions. In particular, understanding how temporal reconfiguration behavior interacts with collective communication patterns is critical for designing efficient scheduling, routing, and resource allocation mechanisms for future AI Datacenter Networks (AI-DCN).

1.1 Problem Statement

The need for such an investigation naturally leads to a central question: given a set of distributed DNN training jobs co-located on a shared Graphics Processing Unit (GPU) cluster connected by an OCS-based reconfigurable fabric, how should the network be scheduled, routed, and shared across jobs so that the overall communication overheads experienced by each job can be minimized.

This problem is difficult because it requires reasoning simultaneously about three tightly coupled phenomena. First, the dependency structure of the training loop – forward pass, backward pass, and gradient collectives happen at a per-layer granularity, and each

step has hard dependencies on the previous one. Second, the temporal behavior of the reconfigurable fabric – OCS switches introduce reconfiguration delays, Top-of-Rack (ToR) packet switches build up queues when circuits are unavailable, and individual chunks must be routed through a network whose connectivity changes over time. Third, the co-scheduling of jobs – all compete for the same shared backend network infrastructure, and decisions made for one affect the others. None of these can be reasoned about in isolation without losing sight of how it affects the other two.

Existing tools do not jointly model these three aspects with enough detail to study how they interact. As a result, important questions remain difficult to answer, such as how a missed OCS circuit delays the backward pass, or how OCS scheduling decisions affect the communication overhead of co-located jobs. This thesis builds that tooling and uses it to study these interactions in a structured manner, surfacing the underlying problems that must be solved in order to answer the central question above. Specifically, this thesis identifies and addresses the following four problems:

P1 – Design space exploration of R-DCN for AI training. OCS-based fabrics have several configurable parameters such as number of OCSes, reconfiguration schedule, reconfiguration frequency, etc. These parameters interact with the DNN model architecture and the parallelism strategies (detailed discussion in §2.1.1) in ways that are not obvious. To study this, we build a full-stack discrete-event simulator that enables systematic evaluation of R-DCN configurations under realistic distributed training workloads.

P2 – Backpressure induced by reconfigurable fabrics on ToR switches. OCS devices carry no packet buffers. When a circuit is inactive, chunks destined for that circuit have nowhere to go and accumulate in the ToR switch buffers, waiting for the circuit to reopen. If a large number of chunks arrive during the same inactive period, the ToR queues can grow significantly and may eventually exhaust the available buffer capacity, leading to packet drops and retransmissions. These delays can slow down the collectives that depend on the affected chunks and ultimately degrade training performance. To address this problem, we propose a multi-path routing strategy that steers traffic towards circuits that are currently active, distributing the load across multiple OCS switches and reducing the queuing pressure at ToR switches.

P3 – Multi-model training on AI R-DCN. When multiple training jobs share the same OCS fabric, their traffic demands can conflict — two jobs may simultaneously need

circuits through the same OCS port, which can only serve one connection at a time. A naive approach increases the number of reconfigurations and extends the time each job waits for its circuits, stalling its collective operations (detailed discussion in §2.1.2). To address this, we propose a demand-aware OCS scheduling framework that constructs per-job circuit schedules, resolves conflicts through conflict-free matching decomposition, and time-multiplexes the fabric across co-located jobs with minimal reconfiguration overhead.

P4 – Routing in reconfigurable networks. Reconfigurable networks provide time-varying connectivity, where links appear and disappear as optical circuits are reconfigured. This violates a fundamental assumption made by conventional routing algorithms, that assumes network connectivity remains stable over time. As a result, routes that appear optimal when computed may later incur significant delays because some of their constituent circuits are temporarily unavailable. To address this challenge, we propose a family of temporal routing algorithms that incorporate circuit availability into route computation and identify low-latency paths through the reconfigurable fabric.

1.2 Contributions and Scope of Thesis

To summarize, this thesis makes the following contributions:

1. **A simulation framework for DNN training on R-DCN**. The simulator models the DNN training and networking stack, ranging from per-layer gradient communication and collective operation scheduling within GPUs to chunk-level transmission over multi-level network topologies and OCS reconfiguration events. It supports multiple parallelism strategies, online and offline routing algorithms, multi-job execution, and a broad range of reconfigurable network design-space parameters. Furthermore, it enables the simulation of multiple training workloads sharing a common GPU cluster. Together, these capabilities provide a foundation for systematically studying the interactions between training workloads and reconfigurable network fabrics.

2. **Characterization of the reconfigurable DCN design-space for AI training**. To the best of our knowledge, this is the first structured analysis of how OCS topology parameters interact with model architecture and parallelism strategy to determine training throughput. We identify the patterns that emerge when

reconfigurable networks are adopted for AI training across a representative set of transformer-based workloads.

3. **A family of temporal routing algorithms for R-DCN**. We design two routing algorithms that explicitly model time-varying link availability in OCS-based networks. The globally-aware algorithm pre-computes acceleration matrices to trace foremost paths efficiently, while the locally-aware algorithm operates with lower pre-processing cost for online use. Together, these algorithms cover a range of trade-offs between solution quality and computational cost, and are applicable to any R-DCN that operates over a periodic circuit schedule.

4. **A multi-job OCS scheduling algorithm for co-located training workloads**. We formulate the problem of scheduling OCS circuits across multiple co-located training jobs as a constrained optimization problem, where the constraints capture the physical limitations of optical circuit switches and the traffic demands derived from each job's communication pattern. We propose an ILP-based approach to construct an efficient offline circuit schedule. In addition, we propose preemption mechanisms that exploit the structured phase behavior of DNN training, where computation and communication phases alternate, to handle network contention during multi-job execution, improving fairness and job completion times.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides the background and related work. It begins with the communication characteristics of distributed DNN training, covering how collective operations arise from the training loop and what demands they place on the network. It then introduces reconfigurable fabrics and the principles of OCS-based datacenter networks. It further reviews existing work on job scheduling for shared GPU clusters and identifies the gaps that this thesis addresses. The chapter closes with a summary of related work. Chapter 3 examines the design space parameters specific to reconfigurable networks, establishing the key variables that govern the behavior of the fabric under training workloads. Chapter4 presents the proposed approaches for the four research problems identified in Section 1.1, covering the

OCS scheduling framework, the preemption scheduling strategies, the temporal routing algorithms. Chapter 5 describes the architecture of the simulator, detailing both the network backend and the system-level components that together model the full stack from GPU compute to OCS circuit scheduling. Chapter 6 presents the experimental evaluation, validating the simulator and assessing the proposed contributions across representative model architectures and cluster configurations. Chapter 7 concludes the thesis with a summary of contributions and directions for future work.

Chapter 2

Background and Related works

This chapter provides the background necessary to understand the work presented in this thesis and places it in the context of existing research. It begins by describing the communication characteristics of distributed DNN training, including parallelism strategies, collective operations, and the traffic patterns they generate. It then introduces reconfigurable datacenter network (R-DCN) fabrics, focusing on OCS-based architectures and their scheduling models. Next, the chapter reviews prior work on job scheduling in GPU clusters and highlights the research gaps addressed by this thesis. Finally, it summarizes the related literature and positions the contributions of this work with respect to prior art.

2.1 Communication in Distributed DNN Training

Training a neural network at scale is fundamentally a distributed computing problem. As models grow larger and datasets grow bigger, a single GPU is no longer sufficient in terms of both memory and compute. This section describes how neural network training works, why it must be distributed across multiple GPUs, and what communication patterns emerge as a result. Understanding these patterns is essential context for the rest of this thesis, since they are precisely what the OCS-based network fabric is designed to serve.

Neural network training involves two phases. In the forward pass, each layer computes activations from its inputs and passes them to the next layer. In the backward pass, which begins at the final layer, the gradient of the loss with respect to the weights is computed, and the parameters of each layer are updated in sequence. This two-phase cycle repeats for

every iteration, making the training loop inherently periodic, as illustrated in Figure 2.1.

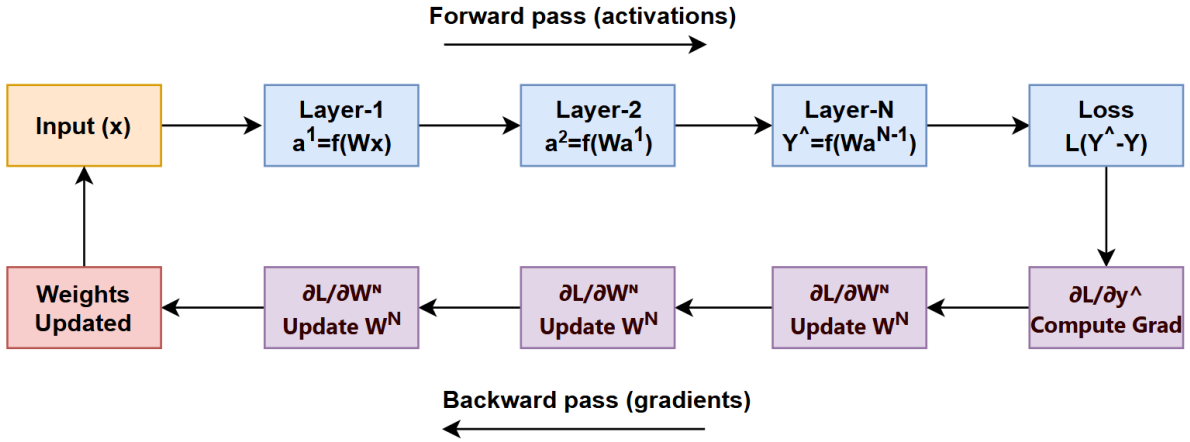

Figure 2.1: Illustrating the periodic behaviour of DNN training.

Training a neural network at scale requires a model architecure (scale is defined in terms of number of parameters) and a dataset. A single GPU has limited memory and compute, making it physically impossible to train large models or process massive datasets alone. At scale, the bottleneck might be in the data dimension (massive datasets - 10TB/120TB) or the model dimension (enormous parameter counts - 500 Billion parameters/2 trillion parameters). Depending on which constraint dominates, different parallelism strategies are employed to distribute the training across multiple GPUs connected via a network fabric, as described in Section 2.1.1.

2.1.1 Multi-GPU Training

When training is distributed across multiple GPUs, the workload must be partitioned and placed in a way that keeps all GPUs busy while minimizing the communication overhead between them. The partitioning strategy determines which data each GPU processes, which part of the model it holds, and what communication it must perform with other GPUs at each training step. There are three fundamental approaches to this partitioning, each targeting a different bottleneck and generating a different communication pattern.

1. **Data Parallelism (DP).** is used when the dataset is too large to process efficiently on a single GPU. As shown in Figure 2.2, the model is replicated across all GPUs and the training data is split and distributed among them. Each GPU runs the forward pass independently on its own data shard since every GPU holds a complete

copy of the model. During the backward pass, the gradients computed on each GPU are aggregated across all GPUs so that every replica applies the same weight update at the end of the iteration, keeping all model copies consistent. This requires every GPU to both send its gradients to all others and receive their gradients in return, making the gradient synchronization a dominant communication pattern in Data Parallelism training.

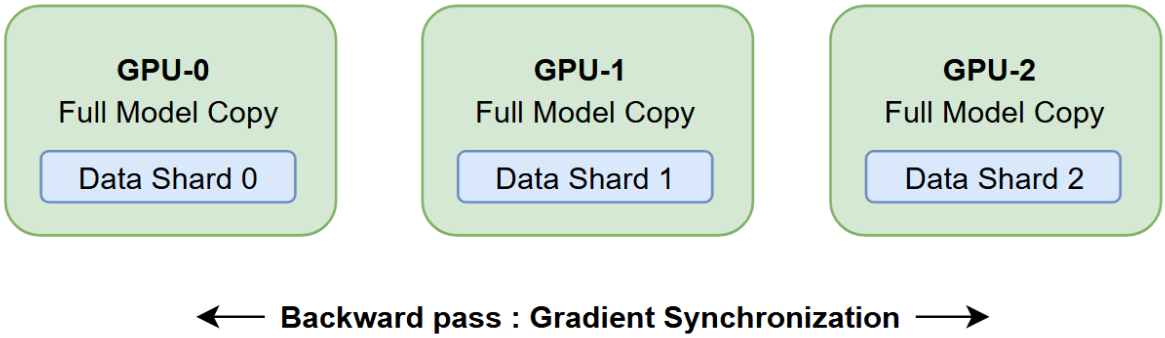

Figure 2.2: Data Parallelism training. The model is replicated across all GPUs and each GPU processes a different shard of the data.

2. **Model Parallelism (MP).** is used when the model is too large to fit in the memory of a single GPU. As shown in Figure 2.3, the model is partitioned across GPUs, so that each GPU holds only a portion of the parameters. This partitioning can be done in two ways. In pipeline parallelism, consecutive layers are assigned to different GPUs, and activations are passed between them as data flows through the network – so communication happens between neighboring pipeline stages, carrying activation tensors in the forward pass and gradient tensors in the backward pass. In tensor parallelism, individual weight matrices are split across GPUs and each GPU computes a partial result, which must then be combined with the partial results from all other GPUs before the output of that layer is complete – so communication happens after every layer, involving all GPUs in the same tensor-parallel group simultaneously.

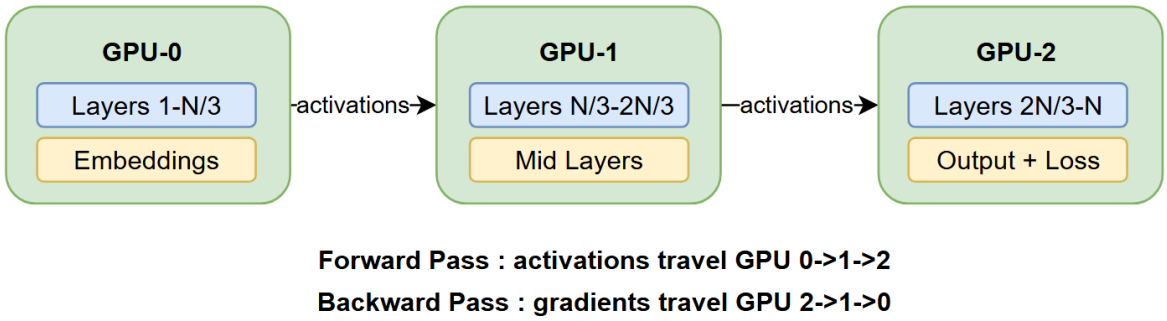

Figure 2.3: Model Parallelism training. The model is partitioned across GPUs, each holding a portion of the parameters.

3. **Hybrid Parallelism (HP).** HP combines both strategies simultaneously, as shown in Figure 2.4. A typical configuration organizes GPUs into groups where model parallelism is applied within each group and data parallelism is applied across groups. This allows training to scale along both the data and model dimensions at the same time. The resulting communication pattern is a combination of both strategies: within each group, GPUs exchange activations or partial results depending on whether pipeline or tensor parallelism is used, while across groups, GPUs synchronize their gradients at the end of each iteration. This means communication happens simultaneously at two different scopes: within the group over high-bandwidth intra-server links, and across groups over the inter-rack network fabric.

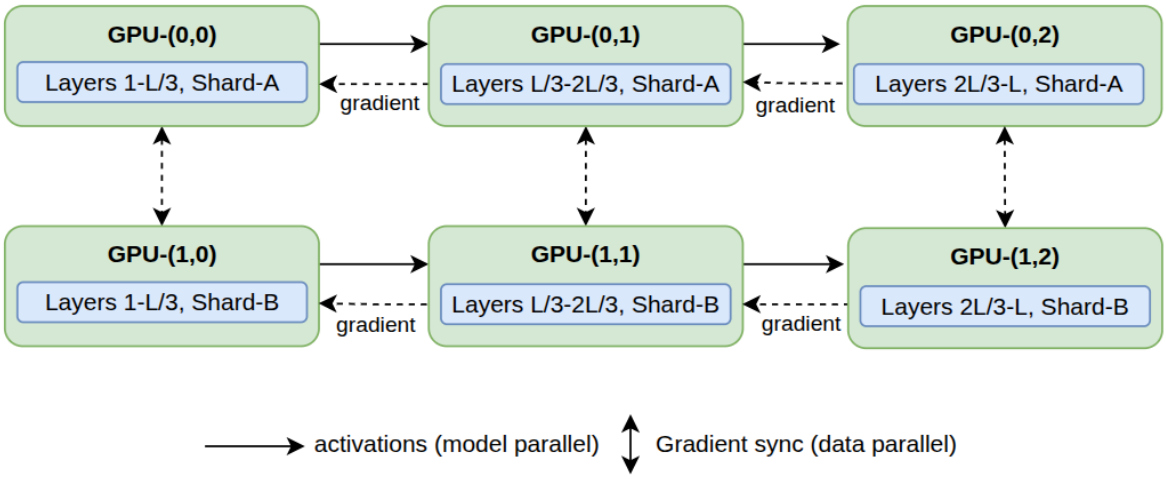

Figure 2.4: Hybrid parallel training. Data parallelism and model parallelism are combined to scale along both dimensions simultaneously.

In this work, we evaluate our proposed approaches using a Megatron-LM [22] style hybrid parallel configuration, as it is one of the most widely adopted frameworks for large-scale training and represents a realistic and well-understood benchmark for distributed DNN workloads.

2.1.2 Collective Algorithms

The parallelism strategies described above all require GPUs to exchange data with each other at various points during training. Regardless of which strategy is used, this communication always reduces to one of a small set of well-defined patterns where each GPU contributes some data, an operation is applied, and each GPU receives a result. These patterns are called collective communication operations, and they are the fundamental building blocks through which distributed training coordinates across GPUs. Understanding these operations is important for this work, because the traffic they generate (in terms of which GPUs communicate, how much data they exchange, and in what pattern) is precisely what the OCS-based network fabric must be designed to serve. The most commonly used collectives in distributed training are described below :

1. **Broadcast.** One GPU sends the same data to all other GPUs. Used to distribute model weights or hyperparameters from a root GPU to all replicas.

2. **Scatter.** One GPU splits its data into equal chunks and sends each chunk to each GPU such that each GPU receives only its designated portion.

3. **Gather.** Each GPU sends its local chunk to one designated GPU, which assembles all chunks into the complete data. This is opposite to Scatter operation.

4. **All-Gather (AG).** Every GPU contributes its local chunk and receives the chunks from all other GPUs, so that every GPU ends up with the complete data. Used in tensor parallelism to reconstruct full activation tensors after a partial computation.

5. **Reduce.** Data from all GPUs is aggregated (e.g., summed, averaged) into a single result, and the result is placed on one designated GPU. Used when only one GPU needs the aggregated result.

6. **All-Reduce (AR).** An All-Reduce operation is a combination of Reduce-Scatter (RS) and AG (two stage collective). Its Like the reduce operation only, but the

aggregated result is delivered to every GPU. This is the key collective in data parallelism, where after each backward pass, all GPUs run All-Reduce to synchronize their gradients before applying the weight update.

7. **RS.** The data from all GPUs is first aggregated and then scattered, so each GPU receives only its designated portion of the aggregated result. Used in memory-efficient training strategies such as ZeRO [29], where each GPU is responsible for updating only a subset of the parameters.

2.1.3 Collective Logical Topologies

Knowing which collective operation is used is only part of the picture. When more than two GPUs participate in a collective, a structured communication pattern is needed to coordinate them efficiently. This structure is called a logical topology which defines which GPU communicates with which, and in what order, to complete the collective. The final result is the same regardless of which topology is used, but the communication pattern, bandwidth utilization, and latency differ significantly. Choosing the right topology for the right collective and the right network is therefore an important design decision. The three most commonly used logical topologies are illustrated in Figure 2.5.

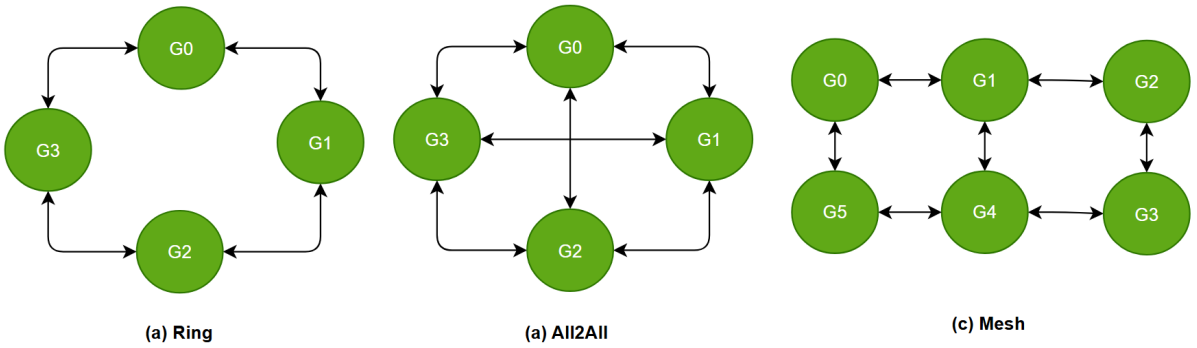

Figure 2.5: Logical topologies used in collective communication: **(a)** Ring, **(b)** All-to-All, and **(c)** 2D-Mesh.

In a **ring** topology, shown in Figure 2.5(a), GPUs are arranged in a logical circle and each GPU communicates only with its two immediate neighbors. Data flows around the ring in steps, with each GPU passing data to the next. This makes the ring highly bandwidth-efficient since every link in the ring is utilized uniformly, and it scales well with the number of GPUs. Ring All-Reduce, which is the dominant algorithm for gradient

synchronization in data parallelism, is built directly on this topology.

In an **All-to-All** topology, shown in Figure 2.5(b), every GPU communicates directly with every other GPU simultaneously. This provides maximum connectivity and is used in collectives where each GPU must send distinct data to every other GPU, such as in Mixture-of-Experts routing. The cost is that communication volume grows quadratically with the number of GPUs, making it the most network-intensive topology.

In a **2D-Mesh** topology, shown in Figure 2.5(c), GPUs are arranged in a two-dimensional grid and each GPU communicates with its immediate neighbors along both dimensions. This is a natural fit for hierarchical communication patterns where intra-rack and inter-rack communication can be mapped to different dimensions of the grid.

The key property that distinguishes AI workload from general datacenter traffic is that it is structured, periodic, and predictable where the same GPUs exchange the same volumes of data in the same pattern every iteration. This predictability is the property that makes demand-aware scheduling of reconfigurable fabrics possible, and it is the starting point for the work presented in this thesis.

2.1.4 Multi-Dimensional Collectives

The logical topologies described above operate over a flat group of GPUs. In practice, GPU clusters are organised hierarchically, for example, GPUs within a server are connected via high-bandwidth NVLink, servers within a rack are connected via intra-rack switches, and racks are connected via the inter-rack fabric. Performing a collective over a flat group in such a hierarchy is inefficient because it ignores the difference in bandwidth and latency between these levels. Multi-dimensional and hierarchical collective topologies such as binary trees and torus are used for this purpose, with the objective of maximizing bandwidth utilization at each level of the hierarchy by keeping traffic local wherever possible and only crossing higher-level links when necessary.

Two techniques make this decomposition practical at scale: slicing, which breaks a large collective into smaller independent units that can be processed concurrently, and pipelining, which overlaps the communication of one unit with the computation of the next.

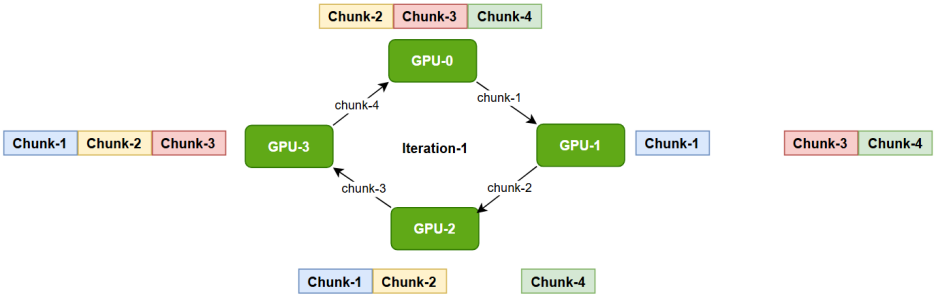

Figure 2.6: Sliced collective communication, where a gradient vector is partitioned into chunks that independently execute the collective operation.

Sliced Collectives

In this approach, a gradient vector of size N is partitioned into k equal-sized sub-gradients, each of size N/k, where k is number of GPUs participating in the collective. The collective communication is then performed iteratively across these chunks as shown in Figure.2.6, where each chunk independently executes the collective operation with synchronization barriers across iterations as shown in Figure. 2.7. Multiple chunks may progress concurrently, and the overall collective operation is considered complete only after all k chunks finish their respective communication phases.

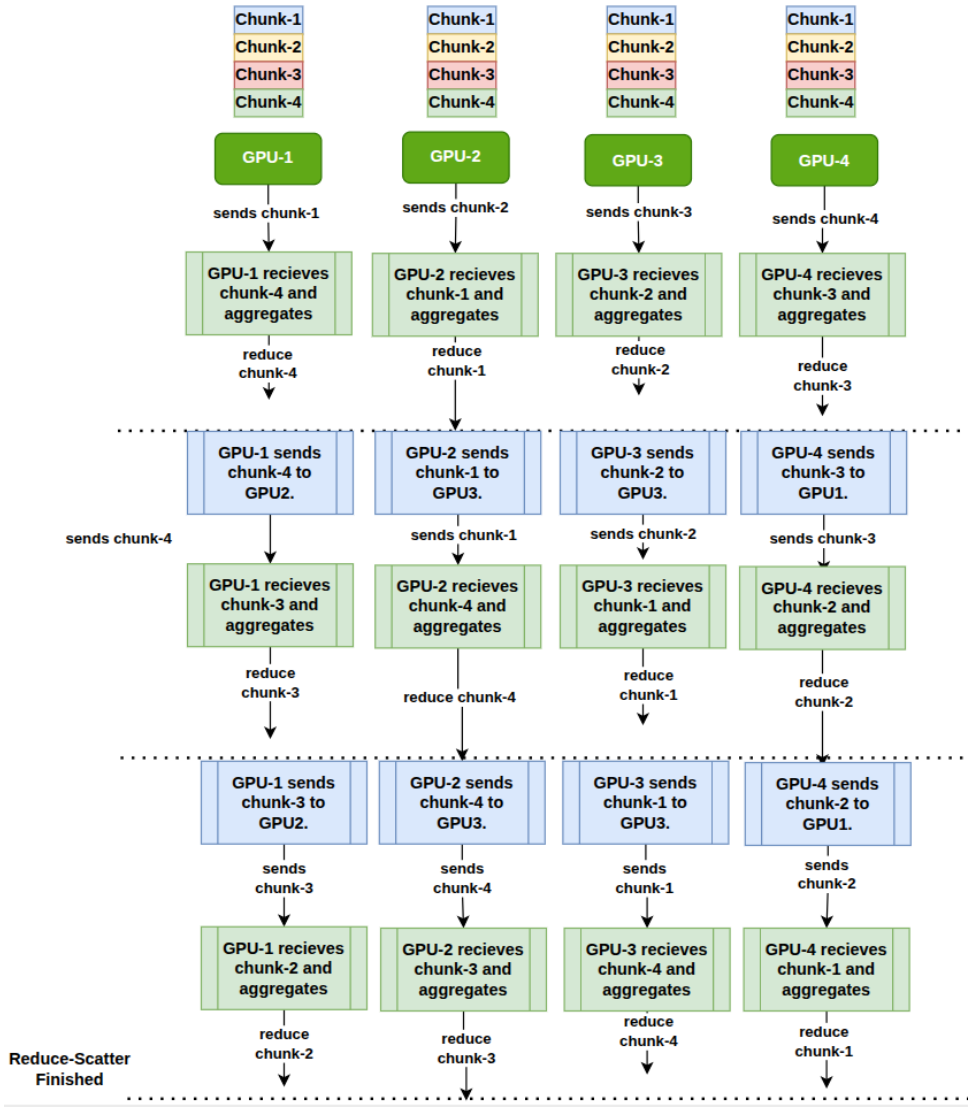

Figure 2.7: Synchronization barriers at the collective level, enforcing that all GPUs complete a phase before any proceeds to the next.

Pipelining

Slicing alone does not eliminate the idle time between communication and computation. Pipelining addresses this by allowing different chunks to progress through their computation and communication phases concurrently rather than waiting for strict synchronization after every chunk. As illustrated in Figure 2.8, while one chunk is being communicated across GPUs, the next chunk can already begin its computation phase. This overlap between communication and computation reduces GPU idle time and improves overall throughput. The degree of overlap depends on the relative duration of the compute and communication phases. when communication is the bottleneck, pipelining exposes this directly, and it is in precisely these scenarios that the efficiency of the

underlying network fabric becomes the determining factor in training speed.

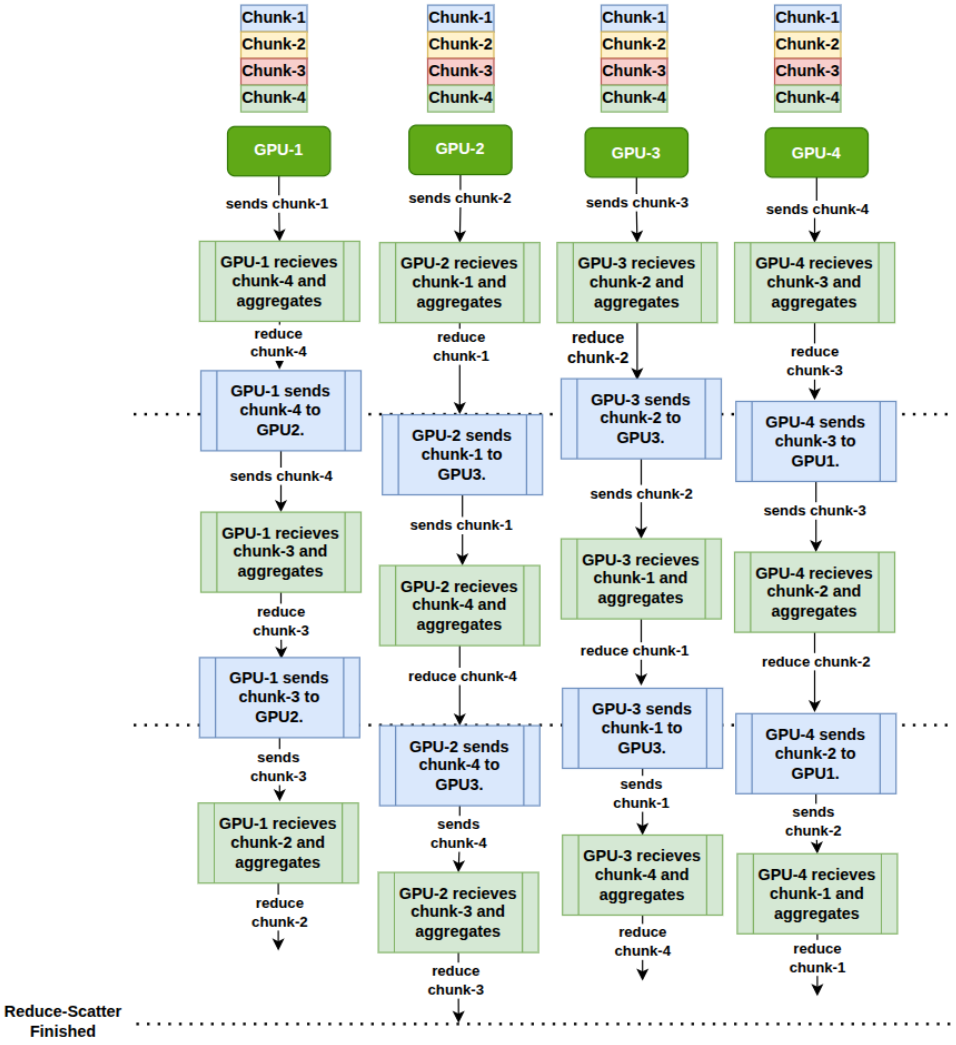

Figure 2.8: Compute-communication overlap during pipelined AllReduce. While one chunk is in flight, the next chunk proceeds through computation.

Hierarchical Collective Execution

Multi-dimensional collectives are communication methods in distributed deep learning that split collective operations across multiple parallel dimensions instead of treating the entire gradient tensor as one large block. As models and the number of GPUs continue to grow, single-dimensional communication methods become less efficient, leading to the need for more structured and scalable approaches.

Formally, consider a distributed training setup with P participating GPUs arranged across d dimensions, such that $P = P_1 \times P_2 \times P_3 ... \times P_d$. A multi-dimensional collective decomposes a single global collective into a sequence of lower-dimensional collectives, each

operating along one dimension of this process grid. For example, in a two-dimensional decomposition with $P = P_{intra} \times P_{inter}$, an AllReduce is first performed across the P_{intra} GPUs within each node over the inter-node fabric. This two-phase approach substantially reduces the volume of data traversing the slower inter-node links, as the intra-node reduction consolidates contributions before they are communicated globally.

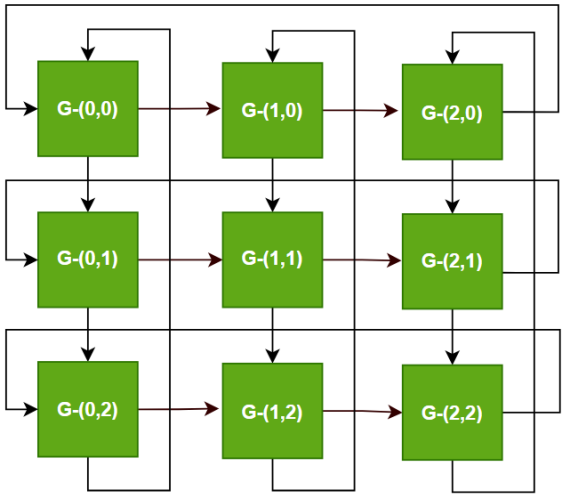

Figure 2.9: Two dimensional Torus topology with wrap-around links.

Consider a 2D Torus topology shown in Figure.2.9, performing an AllReduce operation across 9 GPUs arranged in a 3×3 configuration. Each GPU is assigned a logical coordinate (x, y) where $x, y \in \{0, 1, 2\}$. The two dimensions correspond to distinct physical interconnect technologies:

- **X-dimension**: High-bandwidth NVLink interconnect
- **Y-dimension**: Rack level interconnect (Ethernet/Infiniband)

Each dimension independently forms a ring of 3 GPUs, and due to the torus connectivity, every GPU is connected to its neighboring GPUs in both directions along each dimension.

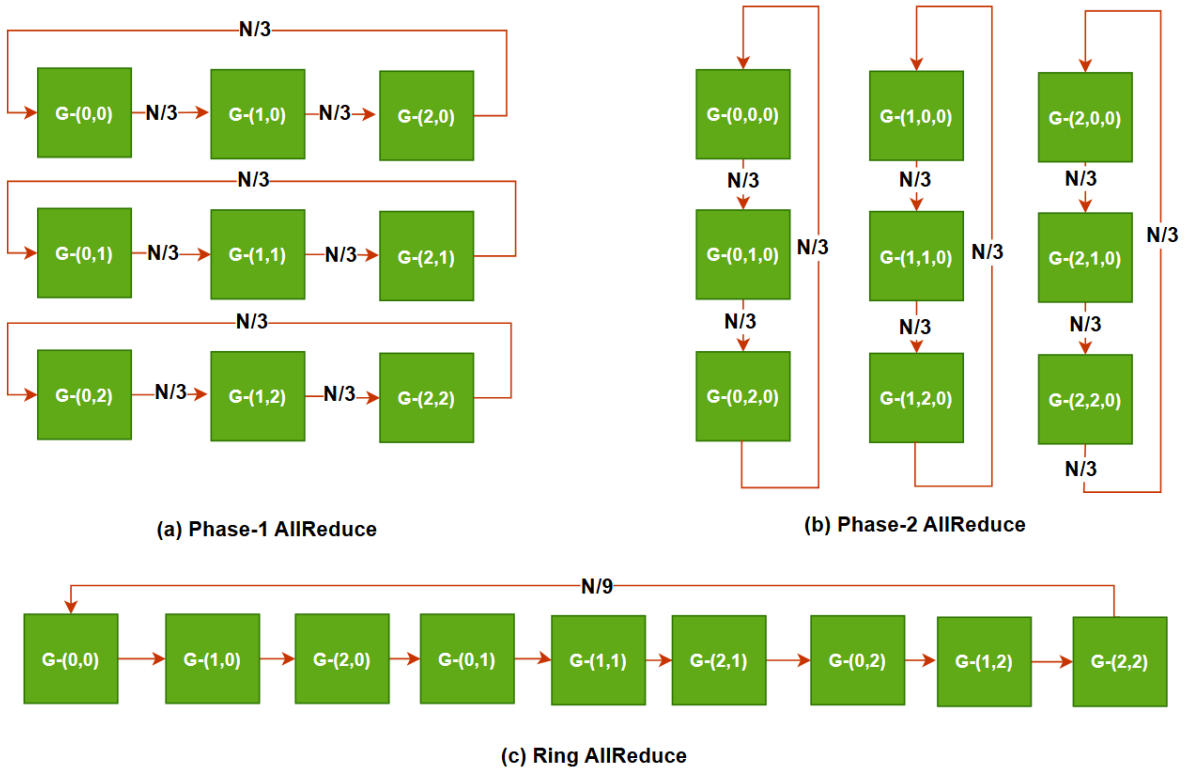

Figure 2.10: Multi-Dimensional (vs) Simple AllReduce

Multi-dimensional AllReduce is executed in two sequential phases, one along each dimension, as shown in Figure. 2.10. During Phase-1 (Figure. 2.10(a)), GPUs perform an AllReduce operation along the X-dimension rings, while during Phase-2 (Figure. 2.10(b)), the partially reduced results are further reduced across the Y-dimension rings.

For a ring of size 3, the AllReduce operation requires only 4 (2 for reduce-scatter, 2 for all-gather) iterations in each phase. Consequently, the total amount of data transmitted through any link during a phase is:

$$4 \times \frac{N}{3},$$

where N denotes the total gradient size. Since the operation is decomposed across two dimensions, the communication load on both NVLink and Ethernet links remains bounded by this factor.

In contrast, a conventional single-ring AllReduce spanning all 9 GPUs ((Figure. 2.10(c)) requires 18 iterations, resulting in a per-link communication volume of:

$$16 \times \frac{N}{9}.$$

The two approaches can therefore be compared as follows:

1. Since
$$\frac{16N}{9} > \frac{4N}{3},$$
the communication load on the slower Ethernet links grows more rapidly in a single-ring AllReduce as the number of GPUs increases.

2. In the single-ring design, the AllReduce operation traverses the slower Ethernet dimension 8 times, whereas the torus-based multi-dimensional AllReduce accesses the slower dimension only 2 times. Consequently, multi-dimensional AllReduce can achieve significantly better performance under bandwidth-constrained or congested network conditions.

The simulator built in this work supports multi-dimensional collective topologies including ring, torus, and binary tree, allowing the communication patterns described above to be faithfully modeled across different parallelism configurations and cluster layouts.

2.1.5 Synchronization Barriers

Collective operations do not happen in isolation. They are embedded within a training loop that has strict ordering requirements (i.e.) a GPU cannot begin the next forward pass until the gradient synchronization from the current iteration is complete, and it cannot apply a weight update until all GPUs have contributed their gradients. These ordering requirements are enforced through synchronization barriers. Barriers are necessary to maintain consistency and without them, some GPUs could begin the next iteration while others are still computing, causing weight divergence or incorrect gradients.

Barriers occur at two levels. At the model level, shown in Figure 2.11, all GPUs must complete the full backward pass and finish gradient synchronization before any GPU begins the next forward pass. These communications are called as **Blocking communication**. This is the coarsest barrier and it determines the wall-clock time of each training iteration – the job can only move forward at the speed of the slowest GPU. At the collective level, shown in Figure 2.7, a finer-grained barrier is enforced within each collective operation, where every GPU must finish its current communication phase before any

GPU proceeds to the next phase of the collective. This ensures that partial results are not consumed before all contributions have been received.

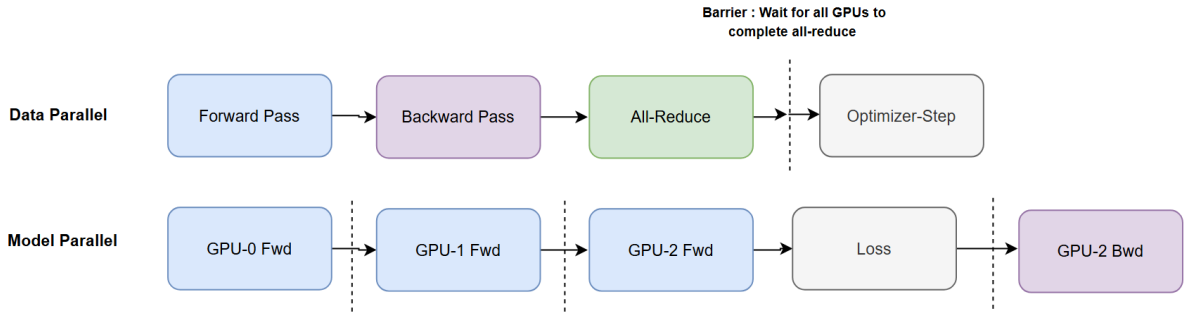

Figure 2.11: Synchronization barriers at model level during AI training.

These barriers are what makes network latency directly visible in training throughput. Any delay in collective communication (whether caused by network congestion, an inactive OCS circuit), holds up every GPU in the job at the barrier, adding directly to the iteration time. This tight coupling between network performance and training speed is the fundamental reason why the design of the network fabric and its scheduling strategy matters so much in distributed training.

The communication patterns described so far place concrete demands on the network fabric connecting the GPUs. How well those demands are served depends directly on the topology and switching architecture of the datacenter network. The following section examines the network topologies commonly used in GPU clusters and the trade-offs they present for distributed training workloads.

2.1.6 Datacenter Network Topologies

A modern datacenter is organized as a collection of servers grouped into racks, with racks connected through a multi-tier switching fabric. In AI training clusters, the network is typically divided into two distinct planes. The *frontend-network* handles external-facing traffic like user requests, storage access, and management traffic and is designed for general-purpose, latency-sensitive workloads. The *backend-network*, also called the training fabric or the GPU interconnect, is dedicated exclusively to the communication between GPUs during training as shown in Figure. 2.12. This includes gradient synchronization, activation passing, and all-to-all transfers. The backend network is where the performance of distributed training is ultimately determined, and it is the focus of this

thesis.

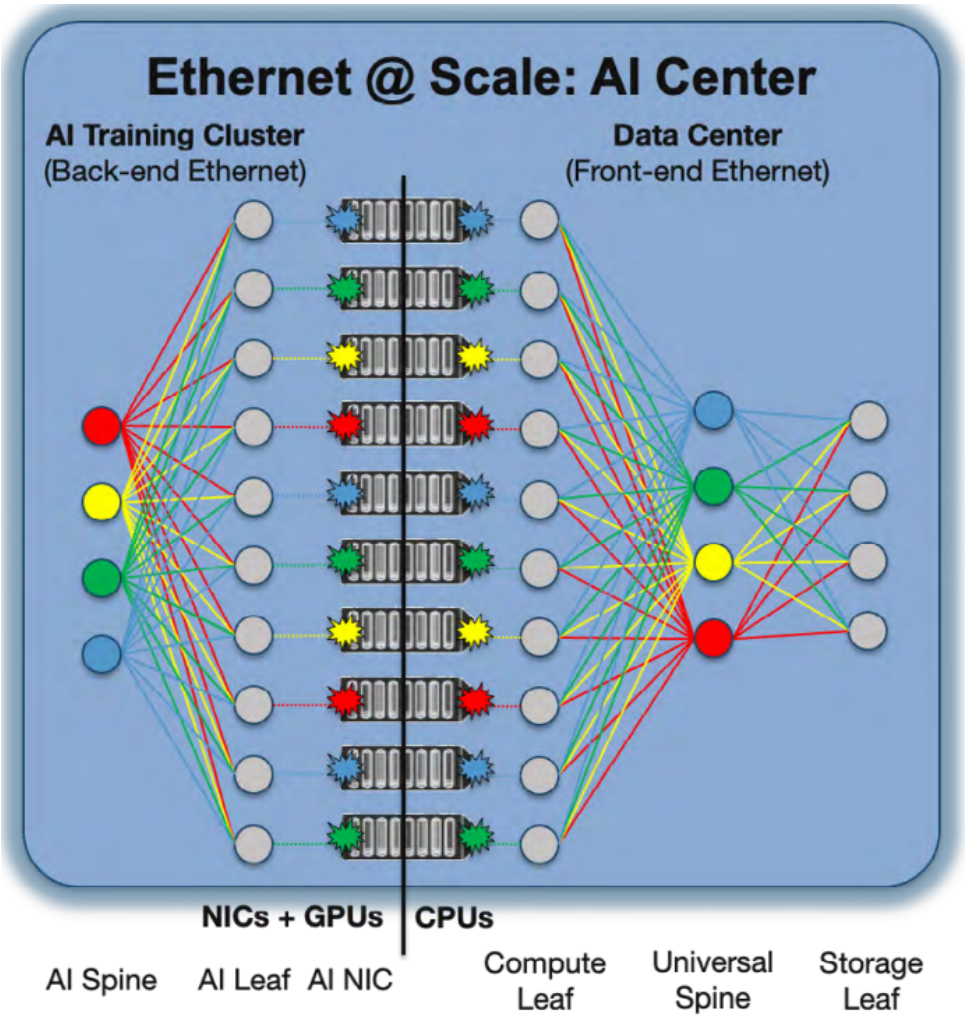

Figure 2.12: Illustration of Datacenter front-end network and AI backend network [32].

Within the backend network, servers are further grouped into racks, each connected to a Top-of-Rack (ToR) switch. ToR switches are then connected upward through one or more layers of aggregation and core switches that provide inter-rack connectivity. The topology of this fabric determines the bandwidth, latency, and scalability of the training network. Datacenter networks are built around different topology designs depending on the workload they are expected to serve. Popular choices include All-to-All, Dragonfly [9], and Fat-Tree [12], each offering a different trade-off between connectivity, cost, and scalability.

All-to-All provides full connectivity between every pair of switches, giving a diameter and average hop count of just one. While this is ideal from a latency and bandwidth perspective, the cabling complexity and cost grow quadratically with the number of nodes,

making it impractical at real deployment scales. Dragonfly reduces cabling costs by grouping switches into super-node clusters connected via high-radix inter-group links, achieving low hop counts at scale. However, it is prone to traffic imbalance and congestion under adversarial or asymmetric traffic patterns, which are common in distributed training. Fat-Tree, also known as the Spine-Leaf topology, is the most widely adopted topology for AI training workloads. It provides full bisection-bandwidth [2] and excellent load balancing through a multi-rooted tree structure, though it requires a large number of switches and careful traffic engineering to fully realize its theoretical bandwidth. Each of these topologies uses static electrical packet switching, meaning the network topology is fixed once deployed and cannot be changed to adapt to traffic demands. This is the fundamental limitation that motivates reconfigurable network architectures. Inspired by prior work on reconfigurable topologies [13, 34], this work adopts a hybrid packet-circuit architecture featuring a rail-optimized spine-leaf topology where OCS switches are integrated within the spine layer, as illustrated in Figure 2.13. This design retains the structural benefits of the Fat-Tree while adding the ability to reconfigure inter-rack connectivity on demand, enabling the network to be tailored to the traffic demands of the training jobs running on the cluster.

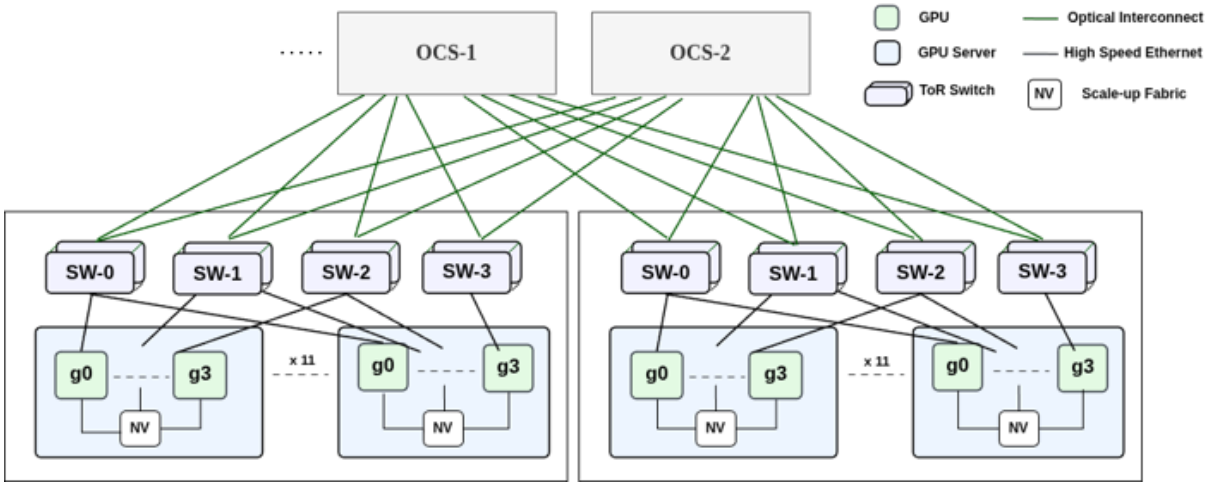

Figure 2.13: Rail-Optimized design with OCSes in the Spine Layer.

2.2 Primer on Reconfigurable Fabrics

An OCS provides reconfigurable, circuit-based connectivity between optical ports by establishing dedicated light paths. Unlike electrical packet switches, which dynamically

multiplex packets across shared buffers, an OCS operates without buffering or queuing and relies on a fixed switch configuration during each circuit interval. At any given time, each ingress port can be connected to at most one egress port, and vice versa, enforcing a one-to-one circuit mapping.

Unlike an electronic packet switch, which processes signals in the electrical domain, a Micro Electro Mechanical System (MEMS)-based OCS operates entirely in the optical domain. As shown in Figure 2.14, instead of an electronic switching core, an OCS contains an array of micro-mirrors (tiny mechanical mirrors fabricated using MEMS technology). When light arrives at an input port (aka ingress-port), these micro-mirrors steer the light towards the corresponding output port (aka egress-port) The connection between an ingress port and an egress port is therefore a direct optical path with no electrical conversion, no buffering, and no protocol processing at the switch itself.

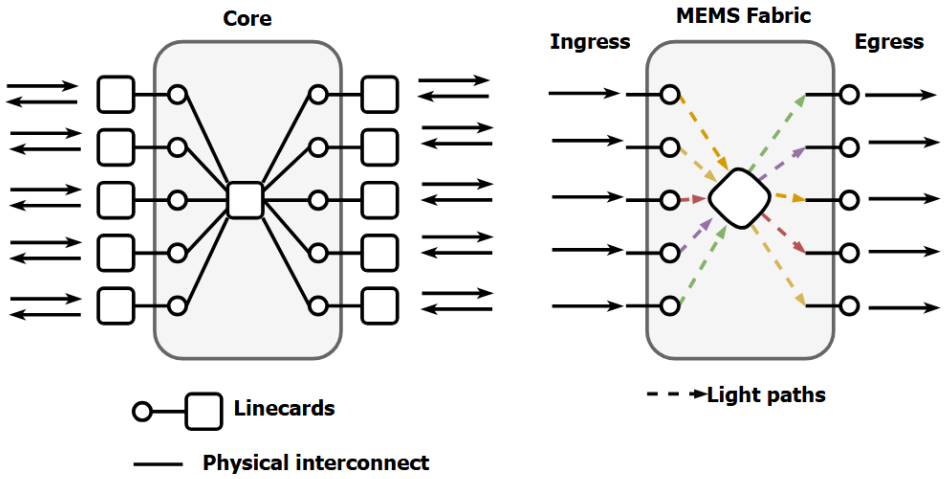

Figure 2.14: Illustration of difference between a packet switch and reconfigurable OCS switch.

Because the OCS operates entirely in the optical domain with no buffering or protocol processing, data traversing an OCS circuit experiences only propagation delay — the time for light to travel the physical distance between the two ends of the circuit. This makes OCS-based inter-rack links effectively transparent to the traffic they carry, behaving like a direct wire between the two connected ToR switches for the duration of the time-slice.

This property creates an important opportunity. By reconfiguring the OCS to connect different pairs of ToR switches at different times, the logical topology seen by the training workload can be made to approximate more connected structures such as All-to-All or Dragonfly, even though the underlying physical infrastructure is a simpler Spine-Leaf

architecture. Figure 2.15 illustrates this idea. The left side shows a conventional Spine-Leaf topology where ToR switches are connected through static spine switches. The middle shows an idealised All-to-All OCS interconnects where every ToR switch is directly connected to every other (like in [21]). The right side shows how OCS switches realize a similar effect but for AI workloads, where the dotted lines between ToR switches represent optical interconnection circuits, which are direct and bufferless like physical wires but time-varying, existing only during the time-slices for which they are scheduled. The dotted representation captures both of these properties: the near-zero processing overhead of an optical circuit and the dynamic nature of its availability.

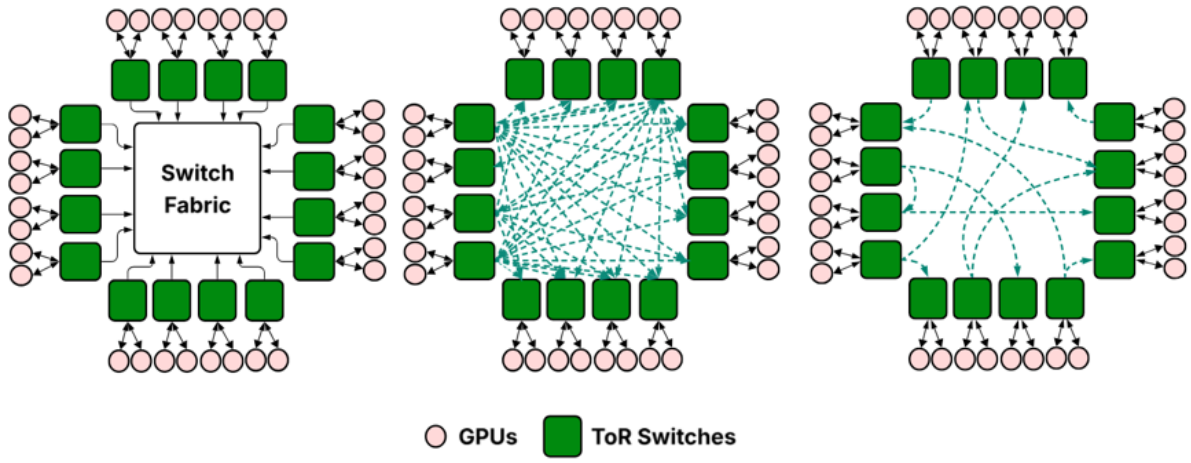

Figure 2.15: Approximating high-radix topologies using optical circuit switches over a Spine-Leaf architecture.

However, reconfigurable fabrics come with a notable drawback: reconfiguration delays. Changing the topology requires tearing down existing optical links and establishing new ones, which takes a significant amount of time. At training scale, this can cause packet losses and added waiting time if not handled carefully. The lack of buffers in optical switches further amplifies the risk of packet loss, which can stall subsequent collectives due to synchronization barriers, a fundamental property of collective communication. Recently, Google [30] adopted these switches in their datacenters, motivating the optical networking community to develop strategies for minimizing reconfiguration delays.

2.2.1 Reconfiguration Schedule

As discussed in the previous section, optical circuit switches provide network reconfigurability. To effectively utilize this capability, the network reconfiguration events must be

carefully coordinated through a process known as reconfiguration scheduling. Broadly, reconfiguration scheduling can be categorized into two approaches: online scheduling and offline scheduling.

Online Scheduling. (also called traffic-aware or on-demand scheduling) is a reactive approach in which routing and topology configurations are determined dynamically as traffic demands arise or network conditions change. Rather than pre-planning the network state, the scheduler observes incoming traffic patterns in real time and triggers reconfigurations to align the network with current demand. This makes online scheduling well-suited for environments with unpredictable or bursty workloads, such as general-purpose cloud networks, where traffic flows are highly variable and cannot be anticipated in advance.

Offline Scheduling. (also called traffic-oblivious scheduling) takes the opposite approach: all topology configurations are computed once, entirely before the network begins operating. The network is divided into a sequence of pre-defined time slices as shown in Figure.2.16, each representing a fixed topology configuration that remains stable for a designated time window. These slices repeat periodically, and routing decisions are baked into each slice ahead of activation. Because the full schedule is known in advance, load balancing can be performed anticipatorily by mapping flows to the slices that will carry them most efficiently, and avoiding disruptive transitions at reconfiguration boundaries.

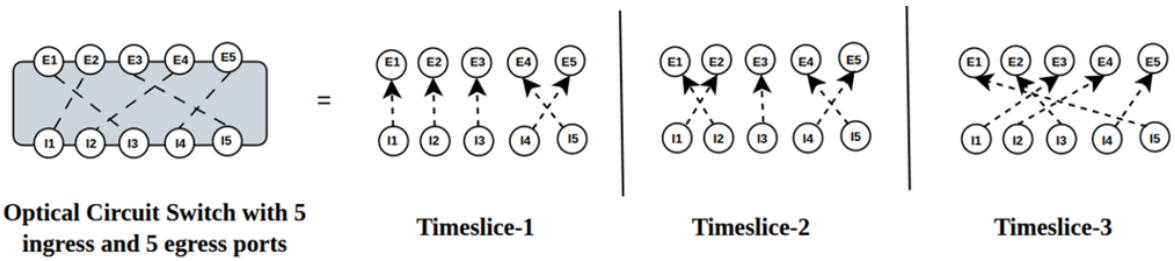

Figure 2.16: Illustration of offline reconfigurable schedule.

AllReduce operations, gradient synchronizations, and other collective communication patterns in AI training are large, long-lived, periodic, and exhibit predictable timing relative to training iteration boundaries. Online scheduling accumulates multiple additive, non-deterministic overheads per reconfiguration event including hardware switching delay, runtime algorithm latency, and forwarding plane revalidation that are fundamentally incompatible with the tight synchronization barriers of distributed training. For traffic that is known in advance, paying this per-decision overhead on every reconfiguration is

unjustifiable.

Offline scheduling directly leverages this predictability. By pre-computing topology slices that align with the periodic structure of training iterations, collective operations can be mapped to favorable topology windows with bounded, deterministic reconfiguration timing. This eliminates the routing instability and non-deterministic latency inherent in reactive approaches, reduces stragglers caused by unpredictable reconfiguration events, and allows load balancing decisions to be made globally and optimally before training begins - rather than greedily and reactively under load.

2.2.2 Gaps & Challenges

Despite prior work on handling general network traffic in reconfigurable networks such as OPERA [20], Helios [3], Rotornet [21], Sirius [1], etc. we found no work at the software level addressing AI-specific optimization for reconfigurable networks. In particular, there is a gap in efficient routing and optical scheduling strategies that leverage the unique, predictable communication patterns of AI training workloads [14]. This gap exists largely due to a lack of infrastructure, expertise, and purpose-built simulators at the intersection of AI and reconfigurable networking. Prior simulation frameworks for OCS networks include NS-3 extensions, OMNeT++ [31], and custom simulators. However, none of them integrate a full DNN training workload model with OCS scheduling and collective communication at the granularity required to evaluate the design space explored in this thesis. AstraSim [26] is the closest related simulator, modeling collective communication over abstract network backends, but does not model OCS reconfiguration physics or temporal routing. This motivated us to build a simulator for reconfigurable networks specifically targeting AI workloads.

2.3 Gaps in Job-Scheduling

GPU cluster schedulers such as Tiresias [4], Optimus [23], and Themis [19] allocate GPUs to training jobs subject to performance, fairness, and locality constraints. They are designed for networks that uses traditional switches which assumes circuits are available all the time. However, this assumption breaks in case of reconfigurable networks where circuits are available only for certain durations of time. Sending the data without the

awareness of this circuit unavailabilities causes severe packet loses and congestions in the network. Our work addresses this gap by proposing reconfiguration aware job scheduling by also leveraging the AI training semantics. The proposed schedulers works under the assumption that each GPU is assigned explicitly to any one of the model and vice versa.

2.4 Summary of Related works and their limitations

OCSes in Distributed DNN Training. Recent work has demonstrated the strong potential of optical circuit switching (OCS) for accelerating distributed DNN and LLM training by exploiting the predictability and repetitiveness of collective communication patterns [42, 24, 40, 18]. However, the evaluation largely focuses on deployed architectures and operational insights, rather than systematic exploration of alternative optical schedules, reconfiguration granularities, or training-aware optimization strategies.

Existing work on OCS-augmented AI data centers can be categorized by how optical reconfiguration is integrated into the distributed training process. One-shot reconfiguration approaches provision the optical topology once per job using offline analysis to optimize for dominant communication patterns over the entire training run [34, 7, 18, 8]. This avoids reconfiguration overhead and performs well for isolated, stable workloads, but provides limited adaptability to phase-level dynamics or shared-cluster execution. In contrast, in-workload reconfiguration approaches dynamically adapt optical circuits at selected communication phase boundaries to better accommodate data-, model-, and pipeline-parallel traffic [36, 35]. However, these studies typically abstract away the packet-switched components of optical–electrical architectures and assume purely optical fabrics, thereby overlooking the impact of ToR buffering and its interaction with reconfiguration dynamics. Moreover, they largely consider single-job execution, ignoring the additional complexity introduced by concurrent training workloads.

While this strategy exposes additional performance gains by aligning network connectivity with phase-specific demands, existing studies primarily focus on single-job execution and abstract away multi-job interference and control-plane contention. As a result, the implications of in-workload reconfiguration in shared AI clusters and the resulting design trade-offs remain insufficiently explored.

TopoOpt [34] represents a significant step toward co-optimizing network topology and

DNN parallelization strategy using reconfigurable optical interconnects. By exploiting the predictable structure of AllReduce traffic, TopoOpt shows that direct-connect optical fabrics can significantly reduce training iteration time compared to fat-tree networks. It adopts a single-shot reconfiguration which is effective for isolated training jobs with stable parallelization strategies, it becomes increasingly restrictive in shared and dynamic training environments. By relying on a fixed topology optimized offline, single-shot approaches assume that heterogeneous communication phases such as data-parallel, model-parallel, and pipeline-parallel exchanges can be jointly accommodated without temporal adaptation. This assumption holds when data-parallel traffic can be reshaped to fit the static topology, but breaks down under multi-job execution where overlapping communication phases from different jobs compete for the same optical resources. In such settings, the flexibility of data-parallel communication is limited by cross-job interference, leading to contention, reduced fairness, and suboptimal utilization. Moreover, static configurations cannot respond to runtime variability arising from stragglers, batch-size changes, or job arrivals and departures. These limitations motivate the need for simulation-driven exploration of time-aware and adaptive optical reconfiguration strategies that explicitly account for multi-job dynamics and phase-level communication diversity in shared AI clusters.

MixNet [16] targets communication inefficiencies in large-scale Mixture-of-Experts (MoE) training by exploiting the structure of expert routing and the sparsity of all-to-all communication between GPUs. It focuses on optimizing expert placement and routing-aware communication to reduce congestion and improve utilization on conventional packet-switched networks, primarily addressing who communicates with whom during MoE execution. While MixNet effectively reshapes communication patterns at the training-system level, it assumes a fixed underlying network fabric and does not consider reconfigurable network resources such as optical circuit switches or time-varying connectivity. In contrast, our work treats the network fabric itself as elastic and explores how optical reconfiguration parameters such as circuit duration, reconfiguration timing, and resource partitioning interact with multi-job training workloads. Rather than optimizing a specific training architecture like MoE, our simulator enables systematic design space exploration across diverse training strategies, revealing network-level challenges and solutions that are invisible under static-fabric assumptions.

ACTINA [36] introduces an analytical framework for evaluating optical circuit–switched (OCS) AI networks by modeling DNN communication graphs and studying in-workload reconfiguration strategies under different latency and bandwidth regimes. However, ACTINA focuses on architecture-level analysis using idealized models and does not capture system-level dynamics such as time-sliced circuit scheduling, inter-job contention, or training-aware scheduling decisions.

At the datacenter scale, Google's Jupiter architecture and Lightwave fabrics highlight the growing role of MEMS-based optical circuit switches combined with SDN control to support predictable, long-running ML workloads [42, 24, 40, 18]. The work emphasizes that ML traffic is repetitive and pre-analyzable, making it well-suited for topology reconfiguration and demand-aware design.

Multiple training jobs. Systems such as Tiresias [4], HiveD [39], and [41] address multi-job interference primarily through GPU placement, job scheduling, and congestion-aware deployment over a fixed packet-switched network fabric. These approaches assume a static underlying network. In contrast, our work explores an orthogonal design space at the network control layer, examining how optical circuit reconfiguration parameters and fabric scalability impact multi-job distributed AI training, using simulation to uncover challenges and solutions that static-fabric approaches cannot capture.

Extending this direction, TOPO-X [5] introduces a more holistic framework that jointly optimizes flow scheduling, optical topology, and training parallelism, formulating the problem as a resource-constrained project scheduling problem (RCPSP). While TOPO-X demonstrates further performance gains over TopoOpt, it relies on solver-driven optimization and abstracted network models, leaving limited insight into how low-level optical parameters—such as circuit time-slice width and switching delay—affect convergence and scalability

These limitations motivate the need for a dedicated simulator that explicitly models DNN training phases, collective communication semantics, OCS reconfiguration latency, and multi-job scheduling on shared optical fabrics, enabling systematic design-space exploration across circuit time-slice width, reconfiguration delay, and scheduling strategies—capabilities that are essential for translating promising OCS-based designs into practical, deployable AI training infrastructures.

2.4.1 DNN Training Simulators.

Evaluating OCS-augmented training designs without physical infrastructure requires simulation tools that can faithfully model both the network reconfiguration dynamics and the training-loop semantics that determine end-to-end performance. Several existing simulators address parts of this problem, but none yet spans the full design space.

Compute and Communication Simulators. AstraSim [26] is the most widely adopted framework for distributed DNN training simulation. It models the full co-design stack across three abstraction layers — workload, system, and network — enabling systematic exploration of parallelization strategies, collective communication algorithms, and network topology choices. It accepts workload descriptions in the Chakra format [38] and supports plug-and-play network backends ranging from analytical models to cycle-accurate NS-3 [27] simulation. AstraSim's breadth makes it well-suited for identifying performance bottlenecks across the compute-communication stack, but its network models assume static connectivity. It cannot represent time-varying circuit schedules, reconfiguration dead times, or the interaction between OCS switching events and in-flight collective operations. This limits its applicability to the reconfiguration-aware design space that OCS-augmented training demands.

Network Level Simulators. OMNeT++ [31] and HPCC [15] provide complementary fidelity at the network level. OMNeT++ is a discrete-event simulation framework that supports detailed switch models, congestion control protocols, and packet-level traffic dynamics, making it suitable for studying how OCS reconfiguration events interact with ToR buffer occupancy and in-flight flows. HPCC addresses the congestion dimension specifically, providing the high-precision per-hop telemetry needed for near-zero-queue congestion control in RDMA networks. Together, they capture the packet-level consequences of OCS design choices that higher-level simulators abstract away, but they do not natively model training-loop semantics, collective communication structure, or the compute-communication scheduling that governs when traffic is generated.

Emulation-Based Approaches. NeuronaBox [17] occupies a different point in the fidelity spectrum. Rather than modeling the network abstractly, it executes the actual training workload on a small subset of physical nodes while emulating the networked execution environment for the remaining nodes, intercepting NCCL (NVIDIA Collective Communication Library) calls and substituting delay-modeled network behavior.

This approach achieves emulation accuracy within 1% of real system measurements, far exceeding analytical simulation for workloads where framework overhead, memory bandwidth, and the implementation details of collective algorithms matter. NeuronaBox is well-suited for faithfully reproducing training behavior over a given static fabric, but it does not model reconfigurable network control, circuit scheduling, or time-varying connectivity. Its role is complementary: it provides ground-truth validation for simulation models and enables realistic framework-level experiments that purely analytical tools cannot replicate.

Chapter 3

Reconfigurable DCN Design-Space

OCS networks introduce new design dimensions that must be aligned with the dynamics of distributed training workloads. Our simulator enables systematic exploration of these interacting design dimensions to evaluate their impact on distributed DNN training.

Time Slice Width (t). OCS fabrics typically operate in a slotted manner, where a circuit configuration remains active for a fixed time slice t. This parameter controls the temporal granularity at which the network can adapt to changing communication demands. For distributed DNN training, synchronization phases often occur at well-defined iteration boundaries but may vary in duration across jobs and across training stages. A smaller t allows the OCS controller to respond quickly to new synchronization demands and reduces waiting time for jobs that reach a collective phase slightly out of phase with others. However, frequent slicing increases control overhead and amplifies the relative cost of reconfiguration delay. Conversely, a larger t amortizes control costs but introduces inefficiencies as the jobs wait (and thus corresponding GPUs sit idle) for the next slice. The simulator explores this trade-off to identify time-slice widths that minimize synchronization delay without destabilizing the control plane or wasting optical capacity.

Reconfiguration Delay ($T_{\mathbf{reconf}}$). Practical optical switches incur a non-negligible reconfiguration delay due to hardware settling time, guard intervals, and control signaling. During this period, no data can be transmitted over the reconfigured circuits. This duration can range from tens of nanoseconds [1] to several microseconds [21] and even milliseconds [18], depending on the underlying technology and the port count of the OCS switch. It measures how long a communication chunk is forced to wait because the

optical circuit it needs is not currently available. In distributed training, reconfiguration delay reduces the effective bandwidth available for synchronization and can dominate performance when circuits are frequently torn down and rebuilt, as it induces stalls during the communication phases of collective operations [36]. This is particularly problematic for workloads with many small synchronization phases. By explicitly modeling T_{reconf}, the simulator quantifies the break-even regimes where OCS provides net benefit over packet switching and identifies acceptable hardware and control-plane targets required to sustain efficient DNN training.

Optical Scheduling Algorithms. Given the demand matrices of training jobs, the OCS controller schedules circuit matchings (port-to-port connectivity) across time slices, allocating different configurations to different jobs. We support multiple scheduling approaches: (i) ILP-based formulations that yield optimal schedules and serve as upper bounds (discussed in §4.2), and (ii) custom policies that can exploit repetitive training structure, such as recurring AllReduce patterns. Scheduling quality directly impacts collective synchronization time and bandwidth utilization. Poor schedules unnecessarily serialize communication, while well-designed schedules overlap independent collectives and fully exploit optical capacity. Our simulator enables direct comparison of optimal and practical schedulers under realistic training dynamics.

Routing and Temporal Connectivity. In multi-hop optical topologies or packet–optical fabrics, establishing end-to-end connectivity requires routing decisions that adapt over time. This arises from the fact that OCS connectivity changes with time, resulting in a network composed of time-varying links. We model routing strategies based on locality-aware decision-making. Our primary approach is a modified A* algorithm that considers both the hop distance to the destination and the reconfiguration window of the immediate next link (discussed in detail in §4.5). However, routing decisions are taken locally at each node without global coordination, and thus the solution is not globally optimal.

Significant opportunities remain for improving routing in OCS-enabled networks tailored for DNN training. In particular, globally coordinated routing strategies that explicitly reason about the full network state and reconfiguration events across all time-varying links represent an open research direction. An effective routing algorithm should be reconfiguration-aware and proactively avoid paths that are likely to incur higher delays or congestion due to circuit transitions along any time-varying link. The simulator

captures how routing algorithms influence scheduling and slicing decisions to improve synchronization efficiency.

Job Preemption Policies. In the presence of multiple concurrent training jobs, optical circuits must be scheduled across jobs to ensure fair and efficient resource utilization. Specifically, preemption defines how an active job may be paused so that another concurrently running job can resume execution using the optical circuit. In hard preemption, the ongoing training job is immediately halted, stopping both computation and communication. In contrast, soft preemption leverages the compute–communication structure of collective operations to transition circuits with minimal disruption. Within soft preemption, we consider two modes: (i) Coarse-Grained [6] and (ii) Fine-Grained preemption (detailed explanation in §4.4). By modeling these options, the simulator quantifies the trade-offs between responsiveness, efficiency, and training stability. It is important to note that job preemption arises from contention for network resources, not GPU compute. We assume that each GPU is assigned to at most one training job, and GPU allocation remains unchanged throughout the job's execution.

Inter-Job Scheduling. Modern training clusters often run multiple training jobs concurrently, each with its own synchronization phases [10, 25]. Inter-job scheduling determines how these jobs are mapped onto the OCS fabric across time slices. *Serial scheduling* minimizes interference but can lead to long waiting times for smaller jobs. *Interleaved scheduling* improves fairness but causes frequent job preemptions and therefore increases completion time for large collectives. Custom, application-aware policies can prioritize jobs approaching synchronization boundaries to reduce straggler effects [37], which are known to disproportionately increase iteration time in synchronous training. Our simulator evaluates these policies in terms of throughput, fairness, and tail iteration latency, which is critical for understanding multi-job training environments.

OCS Scaling. The number of OCSes is a critical design-space parameter for distributed AI training in shared clusters, particularly under concurrent multi-job execution. Increasing the number of OCSes effectively expands the available switching radix and the number of concurrent circuit realizations, enabling better support for overlapping collective communication phases across jobs [35]. From a provider's perspective, varying the OCS count represents a practical trade-off between capital cost, power consumption, fault-domain size, and scheduling flexibility. While scaling can be approximated by de-

ploying a single large-radix OCS, this abstraction hides important differences in control complexity, reconfiguration contention, and fragmentation that arise in multi-job settings [36]. Consequently, our simulator treats the number of OCSes as a design variable, enabling systematic exploration of how OCS deployment choices interact with other system- and job-level parameters.

Chapter 4

Proposed Approaches

The challenges identified in Section 1.1 requires a coordinated approach to designing and managing reconfigurable networks for AI training workloads. This chapter presents a unified framework that combines offline scheduling, intelligent circuit management, and fairness-aware resource allocation. Rather than optimizing network reconfiguration in isolation, the proposed approach jointly considers the interactions among workload characteristics, optical circuit scheduling, job placement, and temporal resource sharing to improve both performance and fairness.

Our design consists of six interconnected components. We first construct bandwidth demand matrices from workload characteristics and network topology information. These demands are then used to generate optical circuit schedules that determine how OCS configurations change over time. Next, we apply job pinning strategies to allocate network resources fairly based on workload requirements.

To support multi-job execution, we introduce mechanisms for inter-job scheduling and OCS sharing, which coordinate how multiple workloads access the shared switching fabric over time also accounting to reconfiguration overheads associated with circuit switching. Finally, we develop temporal routing algorithms that efficiently route traffic across the dynamically changing network topology while respecting time-varying connectivity constraints.

4.1 Demand Matrix Construction

As established in Chapter 2, the communication patterns of distributed DNN training are predictable and periodic where the same set of GPUs exchange the same volumes of data in the same pattern every iteration. This predictability makes it possible to characterize the full communication demand of a training job before it begins, and to use that characterization to pre-program the OCS fabric with a fixed schedule rather than reacting to traffic at runtime (detailed discussion in §2.2.1).

The demand matrix is the data structure that captures this characterization. It encodes, for every pair of ToR switches (switches facing the OCS) in the cluster, whether they want to communicate or not. This serves as the primary input to all subsequent scheduling decisions.

Constructing the demand matrix is not straightforward. Although the communication pattern is structured, the exact traffic between any two racks depends on several factors: how the GPUs participating in the job are placed across racks, which parallelism strategy the job uses, and which logical topology (refer to §2.1.3) governs each collective operation. A job using data parallelism with a ring AllReduce generates a very different inter-rack traffic pattern than one using tensor parallelism with an All-to-All, even if both jobs use the same number of GPUs. The demand matrix construction procedure must account for all of these factors to produce an accurate picture of the job's network footprint.

Since the demand matrices depend on GPU placement, parallelism strategy, and collective logical topology, they are specific to each workload and system configuration. Since the demand matrices depend on GPU placement, parallelism strategy, and collective communication topology, they are specific to each workload and system configuration. Unlike conventional operating system tasks, AI training workloads typically run for days, weeks, or even months. Therefore, we assume that workload configurations and GPU placements are known prior to scheduling. The simulator supports two ways of specifying demand. Users may either provide demand matrices directly (if prior traffic distribution data is available and accessible), or supply a workload description together with a collective implementation strategy, in which case the demand matrices are derived automatically. When derived internally, communication demand is first computed at the GPU level based on the model placement and the logical topology of each collective. Under the assumption of a rail-optimized topology (shown in Figure.2.13), these GPU-level

demands are then aggregated onto the corresponding ToR switches to produce the final demand matrix used for scheduling and routing.

Demand Matrix Representation. A demand matrix is an nxn binary matrix where rows and columns correspond to ToR switches, and each element $DM[i][j]$ indicates whether communication occurs between ToR_i and ToR_j during training, Specifically $DM[i][j] = 1$ means that ToR switches i and j will communicate atleast once during the training job, while $DM[i][j] = 0$ indicates no direct communication. For example, consider a system with 8 ToR switches arranged in a rail-optimized topology as shown in 2.13. The corresponding demand matrix can be :

$$\mathbf{DM} = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

In this example, ToR_0 and ToR_1 communicate (indicated by $DM[0][1] = 1$), while ToR_0 and ToR_3 do not ($DM[0][3] = 0$. Note that ToR_3 shows no communication demands with any other switch, suggesting that the GPUs connected to ToR_3 do not participate in inter-ToR communication during this training phase – they either want communicate among themselves (becomes on-rail communication) or communicate intra-server.

The topology in this work is deliberately restricted to two tiers. In a three-tier design, each pod would contain its own leaf and spine switches, with an OCS layer sitting on top connecting spine switches across pods. To schedule the OCS in such a design, a demand matrix at the spine layer would be needed. However, deriving this from GPU-level communication demand is fundamentally difficult because during training, packets from the same collective may travel through different spine switches depending on load balancing decisions made at runtime. This means there is no reliable way to determine in advance how much traffic each spine switch will carry, and therefore no principled basis for constructing an offline schedule at the OCS layer. Even if a schedule were constructed, a poor mapping between the assumed demand

and the actual traffic would lead to severe under-utilization of some OCS circuits while others become bottlenecks. In a two-tier design, the OCS sits directly above the ToR switches, which means the traffic at the OCS layer maps directly and predictably to the GPU-level communication. This direct correspondence is what makes the demand matrix accurate and the offline schedule valid. Restricting to two tiers is therefore not a limitation of scope but a deliberate design choice that preserves the integrity of the scheduling approach. Nevertheless, extending the approach to three-tier topologies remains possible if prior estimates of network-level traffic distribution are available, enabling OCS scheduling to structure communication more effectively at the network layer.

4.2 Creation of Optical Circuit Schedule

Once the demand matrices between ToRs are obtained for each training job, the next step is to map these demands onto the available OCS circuits. Since OCS connections are directional and form one-to-one ingress–egress mappings at a particular time, scheduling must respect asymmetric traffic demands and enforce matching constraints when allocating circuit slots.

For generating optical circuit connections between ingress and egress ports, we require a matrix which is *conflict-free* with respect to both rows and columns as as defined in Eq. (4.1) and Eq. (4.2).

$$\sum_{i=1}^{p} M[i][j] = 1 \quad \forall\, j = 1, 2, 3, \ldots, p \tag{4.1}$$

$$\sum_{j=1}^{p} M[i][j] = 1 \quad \forall\, i = 1, 2, 3, \ldots, p \tag{4.2}$$

where $M[i][j]$ defines the connectivity between port i and j for an OCS switch with p ports. These constraints enforce a *one-to-one mapping* between ingress and egress ports, ensuring that each ingress connects to at most one egress and each egress receives traffic from at most one ingress.

Since the demand matrices (represented by DM) for individual jobs may not satisfy the conflict-free conditions, each demand matrix is decomposed into multiple sub-matrices

to ensure conflict-free operation as defined in Eq. (4.3).

$$\mathrm{DM} \leq M_1 + M_2 + M_3 + \cdots + M_K, \tag{4.3}$$

where $\leq$ denotes the element-wise inequality.

Existing OCS scheduling approaches for reconfigurable data center networks [21, 1, 20] mainly focus on unpredictable, general-purpose traffic with all-to-all communication patterns by constructing a sequence of perfect bipartite matchings whose union forms complete All2All connectivity. A simple example for a 4×4 network is shown in Eq.(4.4) :

$$\underbrace{\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}}_{\text{All2All Connectivity}} = \underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}}_{M_1} + \underbrace{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}}_{M_2} + \underbrace{\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{M_3} \tag{4.4}$$

In contrast, AI workloads exhibit structured and persistent communication between GPUs during DNN training. An example of configuration with AI specific workload would be as shown in Eq. (4.5) :

$$\underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\text{Structured AI Communication}} = \underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{C_1} + \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}}_{C_2} \tag{4.5}$$

As observed, applying the All2All decomposition in Eq. (4.4) to AI workloads requires three circuit configurations, whereas the structured communication pattern in Eq. (4.5) requires only two configurations. Consequently, blind rotation-based scheduling, as used in traditional networks, provisions many unnecessary connections, resulting in wasted bandwidth, higher energy consumption, and excessive OCS reconfigurations. This ultimately leads to lower throughput and higher communication latency in AI clusters.

4.2.1 Matrix-Decomposition Strategy.

The decomposition of demand matrix into conflict free sub-matrices is equivalent to edge-coloring the bipartite graph induced by DM, where each color represents a feasible OCS matching. By Konig's theorem, the minimum number of required matchings equals the maximum degree of the graph. However, in our case, minimizing the number of matchings K (which is equivalent to the number of decompositions) alone is insufficient. In practice, the sequence in which the configurations $M_1, \ldots, M_K$ are applied plays a critical role in determining overall training performance. Therefore, we jointly consider the following objectives:

1. **Minimizing the number of sub-matrices.** Reconfiguration of the OCS incurs a penalty during training. Therefore, the number of sub-matrices, denoted by K, is kept as small as possible.

2. **Minimizing reconfiguration events.** To further limit OCS reconfigurations, the generated sub-matrices are reordered such that the number of switchovers (i.e., connection flips) between consecutive slices is minimized.

3. **Maximizing port utilization.** If a slice contains idle ingress or egress ports, a decision is made on whether to connect them based on the configuration of the subsequent slice, with the goal of minimizing the total number of reconfigurations.

There exists no single standalone algorithm that can solve this decomposition problem while simultaneously satisfying all constraints and optimization objectives, making the problem NP-hard. Therefore, we formulate it as an ILP problem that searches for the optimal solution exhaustively.

ILP Formulation

Given a binary traffic demand matrix $DM \in \{0,1\}^{p \times p}$, the objective is to decompose it into a minimum number of conflict-free submatrices $M_1, M_2, \ldots, M_K$, while simultaneously minimizing the number of reconfiguration flips between consecutive submatrices.

Decision Variables:

- $x_{i,j,k} \in \{0,1\}$: Indicates whether entry (i,j) is assigned to submatrix k.

- $y_k \in \{0, 1\}$: Indicates whether submatrix k is chosen.
- $z_{i,j,k} \in \{0, 1\}$: Indicates whether entry (i, j) changes between submatrices k and $k + 1$.
- $s \in \mathbb{Z}^+$: Minimum submatrix size (number of ones in the best-used submatrix).

Constraints:

Zero entries remain zero

$$x_{i,j,k} = 0 \quad \forall (i, j) \text{ such that } DM_{i,j} = 0, \ \forall k. \tag{4.6}$$

Row and column conflict constraints

$$\sum_{j=1}^{p} x_{i,j,k} \leq y_k \quad \forall i, k, \tag{4.7}$$

$$\sum_{i=1}^{p} x_{i,j,k} \leq y_k \quad \forall j, k. \tag{4.8}$$

Demand coverage

$$\sum_{k=1}^{K} x_{i,j,k} \geq 1 \quad \forall (i, j) \text{ such that } DM_{i,j} = 1. \tag{4.9}$$

Sub-matrix usage ordering

$$y_k \geq y_{k+1} \quad \forall k \in \{1, \ldots, K - 1\}. \tag{4.10}$$

Flip variable linearization

$$z_{i,j,k} \geq x_{i,j,k} - x_{i,j,k+1} \quad \forall i, j, k, \tag{4.11}$$

Minimum sub-matrix size

$$\sum_{i=1}^{p} \sum_{j=1}^{p} x_{i,j,k} \ \geq \ s - \text{BigM} \cdot (1 - y_k) \quad \forall k. \tag{4.12}$$

Objective Function:

$$\min \; W_1 \sum_{k=1}^{K} y_k \; + \; W_2 \sum_{k=1}^{K-1} \sum_{i=1}^{p} \sum_{j=1}^{p} z_{i,j,k} \; - \; s, \tag{4.13}$$

where $W_1 \gg W_2 \gg 1$ are large positive weights that prioritize minimizing the number of sub-matrices, followed by minimizing the number of reconfiguration flips between consecutive sub-matrices, while encouraging larger sub-matrix utilization.

Eq. (4.7) and (4.8) enforce conflict-free mappings between ingress and egress ports of the OCS by ensuring that each active configuration forms a valid one-to-one matching. Eq. (4.9) guarantees that every demand entry is realized at least once across the set of generated configurations (submatrices), thereby ensuring complete coverage of the demand matrix as discussed in Eq. (4.3).

Eq. (4.10) serves as a strengthening constraint that improves solver convergence by tightening the feasible region without altering optimality. Eq. (4.11) linearizes the reconfiguration cost and enforces minimization of reconfiguration events by penalizing link breakages (1-¿0 transitions) between consecutive configurations. Eq. (4.12) promotes maximal utilization of OCS ports within each active configuration, reducing idle ports and improving slice efficiency. Finally, Eq. (4.13) jointly minimizes the number of reconfiguration transitions by also maximizing the port utilization.

The ILP is solved once offline before training begins using a state-of-the-art MILP solver (Gurobi in our implementation) with a 1% MIPGap (relative optimality gap) to balance solution quality and runtime. Under this configuration, a 60-ToR topology solves in 581 seconds, while a 128-ToR topology requires approximately 3600 seconds.

Limitation

Currently, the demand matrix construction and ILP formulation do not account for bandwidth heterogeneity, collective-specific communication volumes and objectives to minimize the collectives explicitly. Incorporating these factors would make the schedule online. Extending the framework to support such dynamic and heterogeneous workload characteristics remains an important direction for future work.

4.3 Inter-Job Scheduling

When multiple training jobs run concurrently on the same cluster, the OCS fabric must be shared across all of them. Each job has its own demand matrix and its own circuit schedule, and the scheduler must decide which jobs share each OCS switch, how much time each job receives, and in what order their configurations are executed. To address this, we propose a *job pinning* strategy.

4.3.1 Job-Pinning Strategy

When there are multiple jobs to be trained in a shared cluster, we assign each job to a specific OCS switch for scheduling purposes (i.e.) its circuit schedule is computed and programmed onto that switch alone. However, at the routing level, a job's traffic is not restricted to its pinned switch. Chunks are free to traverse any reachable OCS switch in the fabric, allowing the routing layer to exploit available circuits on other switches when the pinned switch is reconfiguring or serving a different matching. This separation between scheduling responsibility and routing flexibility simplifies the schedule construction problem without sacrificing the connectivity offered by the full fabric.

With N_{OCS} available OCS switches, jobs are distributed across them as follows. When $N_{OCS} = 1$, all jobs are pinned to the single available switch. When $N_{OCS} > 1$, jobs are paired based on their pairwise demand overlap ($DM_i \bigoplus DM_j$) and assigned such that jobs with the highest mutual overlap are pinned to the same OCS switch. The intuition is that jobs sharing similar inter-rack communication patterns can reuse the same circuit configurations, reducing the total number of distinct matchings the OCS must serve and minimising reconfiguration cost across job boundaries. For example, with two OCS switches and four jobs as shown in Figure.4.1, where W_1 is have maximal overlap with W_3, and W_2 have maximal overlap with W_4, W_1, W_3 are pinned to first switch and W_2, W_4 to the second.

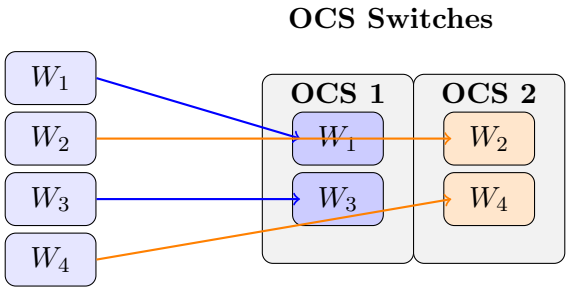

Figure 4.1: Job pinning with two OCS switches and four jobs.

Combining per-job schedule

When the number of jobs exceeds the number of OCS switches as per the birthday paradox, multiple jobs will be pinned to the same switch. So now, one need to decide how jobs share the OCS switch – how much time each job receives on its switch, and in what order jobs are served – with the goal of minimizing cross-job interference, keeping reconfiguration overhead at job boundaries low, and ensuring fair access to the fabric across all co-located jobs.

Assume n jobs $\{W_1, W_2, \ldots, W_n\}$ are pinned to the same OCS switch. Each job W_i has an individual schedule:

$$S_i = \{M_{i,1},\ M_{i,2},\ \ldots,\ M_{i,K_i}\} \tag{4.14}$$

where K_i is the minimum number of conflict-free matrices as defined in Section 4.2. When all n jobs are assigned to the same switch, the OCS must execute a combined schedule formed by merging $\{S_1, S_2, \ldots, S_n\}$.

4.3.2 OCS Sharing Policy

When multiple jobs are pinned to the same OCS switch, their matching sequences must be merged into a single schedule as discussed in Section 4.3.1. The manner in which these individual schedules are merged directly determines the execution order of configurations and the total number of reconfiguration events the OCS must perform. We propose two policies for merging in this work : serial and interleaved.

Serial Schedule

If multiple workload schedules are concatenated serially on a single OCS, the resulting OCS schedule takes the form:

$$S_{\text{OCS}} = \{M_{i,1}, M_{i,2}, \ldots, M_{i,n_1},\ M_{j,1}, M_{j,2}, \ldots, M_{j,n_2},\\ M_{l,1}, M_{l,2}, \ldots, M_{l,n_3}\}, \tag{4.15}$$

where n_1, n_2, and n_3 denote the number of sub-matrices in the schedules of workloads W_i, W_j, and W_l, respectively.

This execution order repeats periodically in the sequence $W_i \to W_j \to W_l \to W_i \to \cdots$ until the prescribed training interval is completed.

Interleaved Schedule

In this approach, we arrange the slices into a single sequence such that the matchings alternate between the jobs - one from the job-1, then one from the job-2, and so on.

If the schedules are interleaved, then the workloads are scheduled as follows:

$$S_{\text{OCS}} = \{M_{i,1}, M_{j,1}, M_{l,1}, M_{i,2}, M_{j,2}, M_{l,2}, \ldots\}, \tag{4.16}$$

This execution pattern repeats periodically in a round-robin fashion until the training interval is completed.

However, naively interleaving the matchings of co-located jobs can significantly increase the total number of reconfiguration events, since consecutive matchings from different jobs are likely to differ in many port connections. To minimise this cost, we frame the interleaving problem as a graph problem. Each matching in a job's schedule is treated as a node, and nodes belonging to different jobs are connected by edges whose weight is the flip cost between the two corresponding matchings – the number of port connections that must change when transitioning from one to the other. The goal is to find an interleaving of matchings across all jobs such that the total flip cost across all consecutive transitions is minimised, while ensuring that the resulting sequence alternates fairly between jobs.

To solve this, we apply the Hungarian algorithm at each round of the schedule. Consider n jobs pinned to the same OCS, each with a sequence of matchings. At any given

round r, each job contributes one matching to the schedule. The question is: in what order should these n matchings be arranged within round r such that the transition from the last matching of round r to the first matching of round $r+1$ costs as few reconfigurations as possible? To answer this, a cost matrix C of size $n \times n$ is constructed at each round boundary, where $C[i][j]$ is the flip cost of transitioning from the matching currently assigned to position i in round r to the matching of job j that will appear in round $r+1$. The Hungarian algorithm is then applied to C to find the assignment of jobs to positions in round $r+1$ that minimizes the total flip cost across all n transitions simultaneously. This gives a new ordering of jobs for round $r+1$. The same process is repeated at every subsequent round boundary, and the result is a complete interleaved schedule where the ordering of jobs is re-optimized at every step, keeping the cumulative reconfiguration cost as low as possible throughout the entire schedule.

Workloads assigned to different OCSes can be executed in parallel. For example, if workload W_p is assigned to OCS_1 and workload W_q is assigned to OCS_2, then W_p and W_q can be executed in parallel.

4.4 Multi-Job Modeling

As jobs are pinned to specific OCS switches, multiple jobs may share the same switch, but only one job's circuit schedule can remain active during a given time slice. Consequently, jobs assigned to the same OCS execute in a time-multiplexed manner (discussed in § 4.3.2), where each job utilizes the OCS for one slot before yielding it to the next job. This sharing mechanism makes preemption necessary. At the end of its allocated slot, a job must pause its activity so that the next queued job can use the network.

The question is what exactly should be paused. The simplest approach is to suspend the entire job by stopping all GPU activity and resuming execution only when the next slot arrives. However, this is inefficient. A DNN training job alternates between compute phases, where GPUs process activations and gradients locally, and communication phases, where data is exchanged over the network. Since compute phases do not require the OCS, pausing them unnecessarily leaves GPUs idle during inter-slot periods and introduces additional dead time on top of the existing OCS reconfiguration overhead.

A better approach is to suspend only the network-dependent portions of the job while allowing computation to continue. This forms the basis of **soft preemption**. When a job's slot expires, the job does not halt completely. Instead, compute phases continue normally, while communication events are deferred until the job's next allocated slot begins. This allows GPUs to remain productive during inter-slot periods and reduces the performance impact of time-multiplexing on overall training throughput.

Soft preemption can be applied at different levels of granularity within the training loop, and the choice of granularity affects both the correctness and the efficiency of the preemption mechanism. We implement two variants: *Coarse-Grained preemption*, which draws the preemption boundary at the collective level, and *Fine-Grained preemption*, which draws it at the individual chunk level within a stream.

4.4.1 Coarse-Grained Preemption

Coarse-Grained preemption pauses a job by deferring the generation of new collective communications during another job's quantum, without interrupting the already in-flight collectives, as illustrated in Figure. 4.2. In the figure, the blue regions denote communication phases, the green regions denote computation phases, and the yellow regions represent compute–communication overlap and dashed line region is to tell whoes job slot it is.

While the job is paused, its computation phases may continue to progress; however, no new communication operations are issued until the job resumes in its next allocated slot. By avoiding the suspension of in-flight network operations, Coarse-Grained preemption maintains a simple and clean pause–resume mechanism with minimal runtime complexity.

The primary trade-off is that a job may occupy its OCS time-slot slightly longer than the assigned quantum duration, since preemption occurs only at communication boundaries after the currently active collective operation completes.

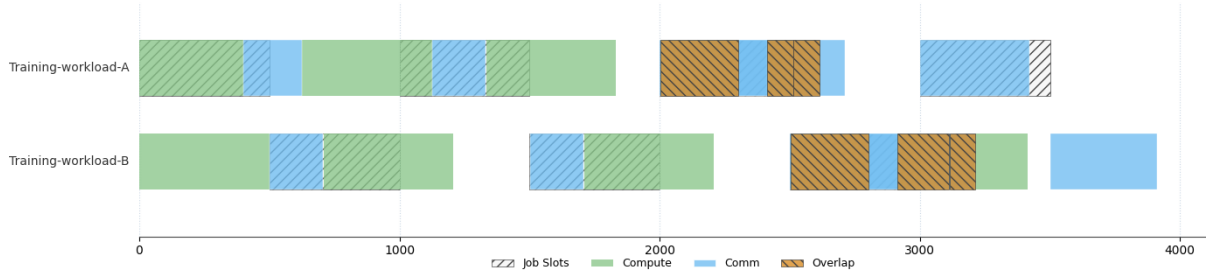

Figure 4.2: Illustration of Coarse-Grained preemption.

4.4.2 Fine-Grained Preemption

In contrast to CG, Fine-Grained preemption allows preemption of job at collective granularities leveraging the multi-phase nature of the collectives (detail discussion in § 2.1.2). Fine-Grained preemption allows to pause mid-collectives at the collective phase boundaries as illustrated in Figure. 4.3. Any stream that are in the mid-collective will be saved into a paused-streams list. When the job is resumed in its next time-slot, the scheduler resumes the paused streams and start from where they got left off and continues transmitting the remaining chunks. This leads to more precise time-sharing.

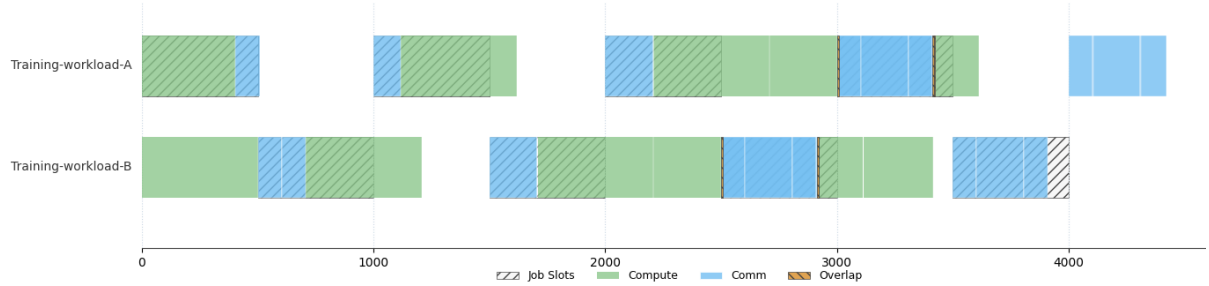

Figure 4.3: Illustration of Fine-Grained preemption

The choice between the two strategies affects how tightly the system-level job scheduling is coupled to the network-level OCS schedule. Coarse-Grained preemption aligns job boundaries with collective boundaries, which is simpler and avoids partial transmissions mid-slice. Fine-Grained preemption gives the scheduler finer control over GPU time allocation, which matters more when jobs have very different communication intensities and strict fairness is required.

4.5 Temporal Routing

Routing in an OCS-based network is fundamentally different from routing in a static network. In a static network, links are continuously available, and routing primarily involves selecting an efficient path through a fixed topology. In contrast, an OCS-based network changes its connectivity with every reconfiguration event, meaning that a circuit available during one time slice may not exist in the next. As a result, if the chunk arrives at an intermediate node before the corresponding outgoing circuit becomes available, it must wait in the ToR queue until the next active window opens. This waiting time directly increases the end-to-end communication latency.

To handle this, we model the network as a temporal graph in which each edge carries a set of active intervals representing the time windows during which the corresponding OCS circuit is live. Figure 4.4 illustrates a temporal graph constructed within the interval [0,25] with the dotted lines representing the temporal connectivity. The connection between the ToR switches and the OCS ports exists for [0,25] as they are static whereas, the connections between the OCS ports exists only at certain intervals of time (determined by the OCS-schedule). The edges are also weighted by the traversal times of the chunk as chunks of different size takes different time to travel as defined by Eq. (4.17)

$$LinkWeight\ (ns) = \ LinkLatency\ (ns)\ +\ \frac{ChunkSize\ (in\ bits)}{Bandwidth\ (ns)} \tag{4.17}$$

Routing in this setting is fundamentally different from routing in a static network. The cost of a path is not just a function of its hop count or its static latency, it depends on when the chunk is injected, how long it waits at each intermediate node, and which OCS time-slices are available along the way. We formalize this as the foremost path problem on a temporal graph: given a source, a destination, and a departure time, find the path that delivers the chunk to the destination as early as possible.

We propose two routing algorithms formulated on the temporal graph representation of the network. The first solves the foremost path problem by minimizing arrival time, while the second jointly balances hop count and immediate waiting time at intermediate nodes. These algorithms offer different trade-offs between routing quality and computational complexity, as discussed in the following subsections.

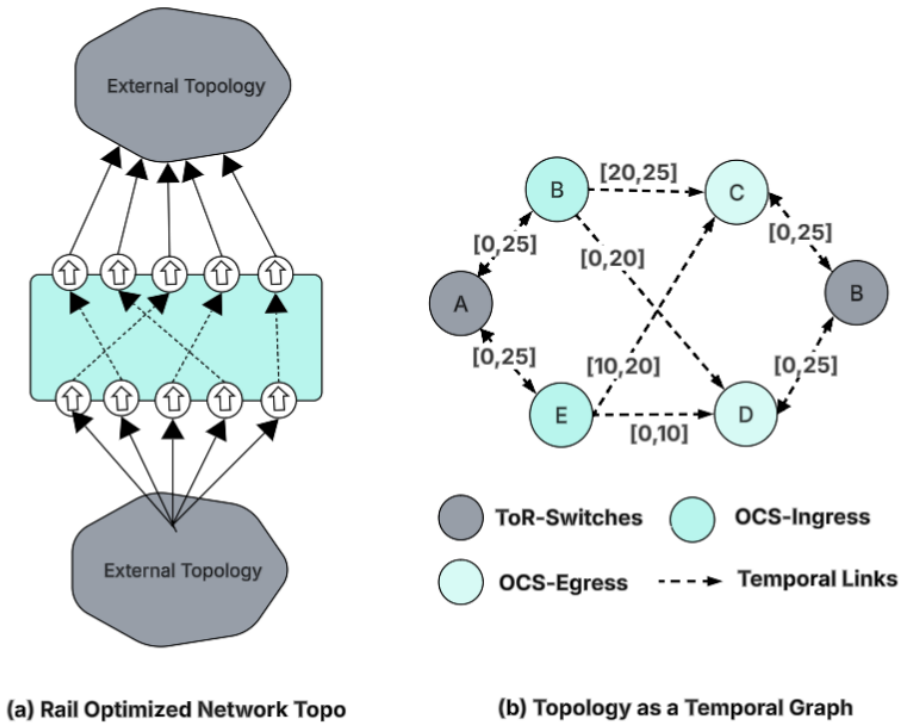

Figure 4.4: Reconfigurable network as a Temporal-Graph.

4.5.1 Multi-path Routing Algorithm

A fundamental inefficiency in OCS-based networks is the backpressure that builds up at ToR switches when required circuits are inactive. Chunks arriving during these inactive intervals must wait at the ToR switches until the next time slice begins, and when many chunks from the same collective contend for the same circuit, queuing delays accumulate and increase communication time. We propose multi-path routing as a solution to mitigate these backpressures by evenly distributing the traffic by also accounting to circuit unavailabilities and reachability to the destination. Since a source and destination ToR are often connected through multiple OCS switches with different reconfiguration schedules, chunks can dynamically choose among several valid paths instead of waiting for a single inactive circuit. This increases the likelihood of finding an immediately available path, thereby reducing queuing delays and improving communication efficiency.

The algorithm finds k paths with the earliest arrival times from a source node to a destination node in a temporal graph. It exploits two precomputed acceleration matrices — matrix A and matrix N to trace the first foremost path in $O(\text{hops})$ time and then systematically explore diverging paths using a min-heap.

Key Definitions

Before describing the algorithm, we define the key terms used throughout.

- **Foremost path.** A path from source s to destination d that minimizes the arrival time at d, given that the chunk departs s at time t_0. Along this path, a chunk may wait at intermediate nodes for the next active interval of the outgoing link.

- **Out-edge.** An edge incident on a node in the first foremost path that was not chosen by the greedy next-hop selection. Out-edges represent divergence points — alternative continuations of the path that may lead to different foremost paths.

- **Divergence point.** A node on the current path from which at least one out-edge exists. The divergence number is the index of this node in the path sequence, indicating where the new path deviates from the previous one.

- **Matrix $A[v][t]$.** For each node v and discrete time step t, $A[v][t]$ stores the index of the augmented multigraph node that corresponds to being at node v at time t. This enables $O(1)$ lookup of the correct augmented node for any (node, time) pair encountered during traversal.

- **Matrix $N[v][t]$.** For each node v and time t, $N[v][t]$ stores the index of the neighbour that a greedy foremost-path algorithm would visit next from v at time t. N is computed by a backward sweep from the destination over the time-expanded graph, encoding the globally optimal next-hop at every (node, time) pair.

- **Candidate tuple.** An entry in the min-heap of the form
  (T_{arr}, prefix_path, waits, t_{instant}, next_node), where T_{arr} is the estimated arrival time at the destination via this candidate, prefix_path is the sequence of nodes visited so far, waits is the list of waiting times at each hop, t_{instant} is the current time at the divergence point, and next_node is the first node of the unexplored branch.

Algorithm Description

The algorithm proceeds in two phases.

Phase-1: First Foremost Path. The A and N matrices are constructed from the temporal graph modified to reflect the chunk's injection time t_0. Starting from the source node at time t_0, the first foremost path is traced greedily: at each step, the next node is looked up in N[current_node][current_time]. The waiting time at each hop is the gap between the chunk's arrival time and the start of the earliest active interval on the outgoing link. During this traversal, every out-edge encountered — edges from nodes on the path to neighbours not selected by N — is pushed onto the min-heap as a candidate tuple, recording the divergence point and the estimated arrival time if the path were to follow that out-edge instead.

Phase-2: Remaining $k-1$ Paths. The algorithm repeatedly pops the candidate with the smallest estimated arrival time from the heap. From the divergence point recorded in the candidate, the algorithm resumes greedy tracing using N until the destination is reached or no valid next hop exists. If the destination is reached, the resulting path is added to the result set. New out-edges encountered during this extension are pushed onto the heap. This continues until k paths are found or the heap is exhausted.

Algorithm 1 Construction of Matrices A and N

Require: Temporal graph G, destination d, graph lifetime T

Ensure: Matrix A, Matrix N

$A[v][t] \leftarrow -1, \quad N[v][t] \leftarrow -1 \quad \forall\, v, t$

$A[d][t] \leftarrow t \quad \forall\, t$

// Precompute feasible edges at each time step

for each edge (u, v, w) with active intervals **do**

    for $t = 0$ to T **do**

        if $t + w$ falls within an active interval **then**

            Add (u, v) to *timeEdges*$[t]$

        end if

    end for

end for

// Backward sweep

for $t = T$ down to 0 **do**

    for each (u, v) in *timeEdges*$[t]$ **do**

        $t_{\text{arr}} \leftarrow t + w(u, v)$

        if $A[v][t_{\text{arr}}] \neq -1$ **and** $A[v][t_{\text{arr}}] < A[u][t]$ **then**

            $A[u][t] \leftarrow A[v][t_{\text{arr}}], \quad N[u][t] \leftarrow v$

        end if

    end for

end for

// Fill gaps by propagating forward

for each node v **do**

    for $t = T - 1$ down to 0 **do**

        if $A[v][t] = -1$ **then**

            $A[v][t] \leftarrow A[v][t + 1], \quad N[v][t] \leftarrow N[v][t + 1]$

        end if

    end for

end for

return $A,\ N$

Algorithm 2 Find-KForemost-Paths Algorithm

Require: Source s, destination d, injection time t_0, number of paths k

Ensure: k foremost paths ordered by arrival time

Compute matrices A and N from the temporal graph at boundary t_0
result $\leftarrow [\,]$, $\quad \mathcal{H} \leftarrow$ empty min-heap
// Phase 1: find the first foremost path
$v \leftarrow s$, $\quad t \leftarrow t_0$, $\quad$ path $\leftarrow [\,]$
while $v \neq d$ **do**
    $u \leftarrow N[v][t]$ ▷ globally optimal next hop
    if no valid u **then break**
    end if
    Push all out-edges from v (neighbours $\neq u$) onto $\mathcal{H}$ with their estimated arrival times
    Advance: $t \leftarrow t + \text{wait}(v, u, t) + \text{latency}(v, u)$, $\quad$ path.append(u)
    $v \leftarrow u$
end while
if path reaches d **then** append to result
end if
// Phase 2: find remaining $k-1$ paths from heap
while |result| $< k$ **and** $\mathcal{H}$ not empty **do**
    Pop the candidate with the smallest arrival time from $\mathcal{H}$
    Resume greedy traversal from the divergence point using N
    Push any new out-edges encountered onto $\mathcal{H}$
    if path reaches d **then** append to result
    end if
end while
return result

Working Example

Consider the temporal graph in Figure 4.5, consisting of four nodes A, B, C, D and edges with the following active intervals:

- $A \to B$: active at $(0, 1)$ and $(2, 3)$
- $A \to C$: active at $(2, 4)$
- $B \to C$: active at $(1, 2)$
- $B \to D$: active at $(3, 5)$
- $C \to D$: active at $(2, 3)$ and $(4, 5)$

The source is A, the destination is D, and the injection time is $t_0 = 0$.

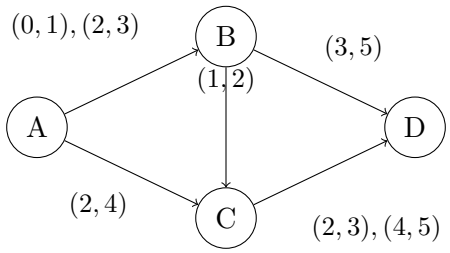

Figure 4.5: Temporal graph example with four nodes and time-varying edge availability.

Phase 1 — First foremost path.

Starting at A, $t = 0$:

- $N[A][t{=}0] \to B$ (earliest departure: $t = 0$, arrives at B at $t = 1$). Out-edge: $A \to C$. Push $(T_{\text{arr}},\ [A],\ [0],\ t{=}0,\ C)$ onto $\mathcal{H}$.
- $N[B][t{=}1] \to C$ (earliest departure: $t = 1$, arrives at C at $t = 2$). Out-edge: $B \to D$. Push $(T_{\text{arr}},\ [A, B],\ [0, 0],\ t{=}1,\ D)$ onto $\mathcal{H}$.
- $N[C][t{=}2] \to D$ (earliest departure: $t = 2$, arrives at D at $t = 3$).

First foremost path: $A \to B \to C \to D$, arrival time $= 3$.

Phase 2 — Second and third foremost paths.

Heap contains: $\{(4,\ [A, B],\ [0, 0],\ t{=}1,\ D),\ (T,\ [A],\ [0],\ t{=}0,\ C)\}$.

- Pop $(4,\ [A, B],\ \ldots,\ D)$: resume from $B \to D$ at $t = 1$. Earliest active interval for $B \to D$ is $(3, 5)$, so chunk waits until $t = 3$, arrives at D at $t = 4$. Path: $A \to B \to D$, arrival $= 4$. This is the 2nd foremost path.

- Pop $(T,\ [A],\ \ldots,\ C)$: resume from $A \to C$ at $t = 0$. Earliest active interval for $A \to C$ is $(2, 4)$, chunk waits until $t = 2$, arrives at C at $t = 3$. From C, $N[C][t{=}3]$ has no active interval at $t = 3$ (next is $(4, 5)$), so waits until $t = 4$, arrives at D at $t = 5$. Path: $A \to C \to D$, arrival $= 5$. This is the 3rd foremost path.

The three foremost paths in order are:

1. $A \to B \to C \to D$, arrival time $= 3$
2. $A \to B \to D$, arrival time $= 4$
3. $A \to C \to D$, arrival time $= 5$

This matches the paths shown in Figure 4.6.

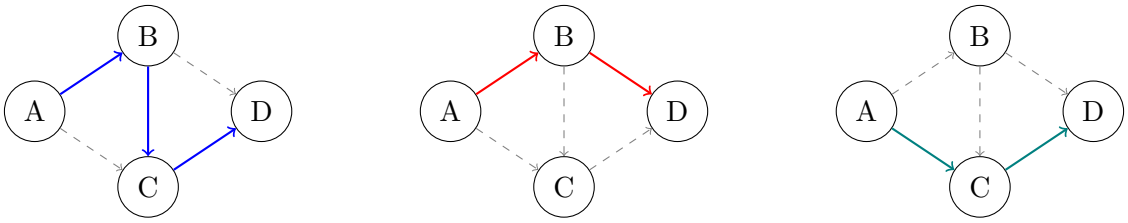

Figure 4.6: The three foremost paths: (i) 1st Foremost (blue), (ii) 2nd Foremost (red), (iii) 3rd Foremost (teal). Coloured edges show the path taken; dashed grey edges are alternatives not chosen.

4.5.2 Heuristic-Routing Algorithm

The K-Foremost path algorithm with $k = 1$ computes the earliest-arriving path between a source and destination. However, in datacenter networks, the probability of a chunk encountering queuing increases with the number of hops along the path. Consequently, a longer path that appears faster in offline computation may in practice experience higher cumulative queuing delays across its additional hops than a shorter path would incur. Therefore, minimizing waiting time alone is not always optimal; in many cases, preferring a shorter path with slightly higher waiting can lead to lower overall communication latency.

To address these limitations, we propose a new routing algorithm based on A* search framework. Rather than optimizing solely for the theoretical arrival time, this algorithm balances two competing objectives at each hop: the immediate waiting time at the current node, which reflects the actual delay the chunk will experience before its next circuit

opens, and the minimum hop count from the candidate neighbour to the destination, which serves as a proxy for the congestion risk of the remaining path. A balance factor α controls the trade-off between these two objectives, allowing the algorithm to be tuned between favouring low latency and favouring short paths.

At each intermediate node u reached at time t, the algorithm evaluates all neighbors. For each neighbor v, it identifies the earliest feasible departure time by finding the first active interval $[l, r]$ of edge (u, v) such that the $max(t, l) + w_{uv} <= r$, yielding a waiting time of $w_v = max(t, l) - t$. To guide the routing process to the destination, the algorithm precomputes the minimum hop-count distance $H(v)$ from each node v to the destination node (using BFS). This hop count serves as an admissible lower bound estimate for the remaining routing cost. Consequently, for each candidate neighbor, the algorithm computes a normalized composite score as Eq. (4.18)

$$score(v) = \alpha * \hat{w}_v + (1 - \alpha) * \hat{H}_v, \tag{4.18}$$

where $\hat{w}_v$ and $\hat{H}_v$ are the waiting time and heuristic estimate normalized by their respective maxima across all candidates at the current step, and $\alpha \in [0, 1]$ is a tunable parameter. Setting $\alpha = 0$ yields a purely topology-driven search that minimizes remaining hop count, while $\alpha = 1$ reduces the algorithm to a greedy minimum-wait strategy. The algorithm selects the neighboring node v with the lowest *score* as the next hop and forwards traffic through it to reach the destination.

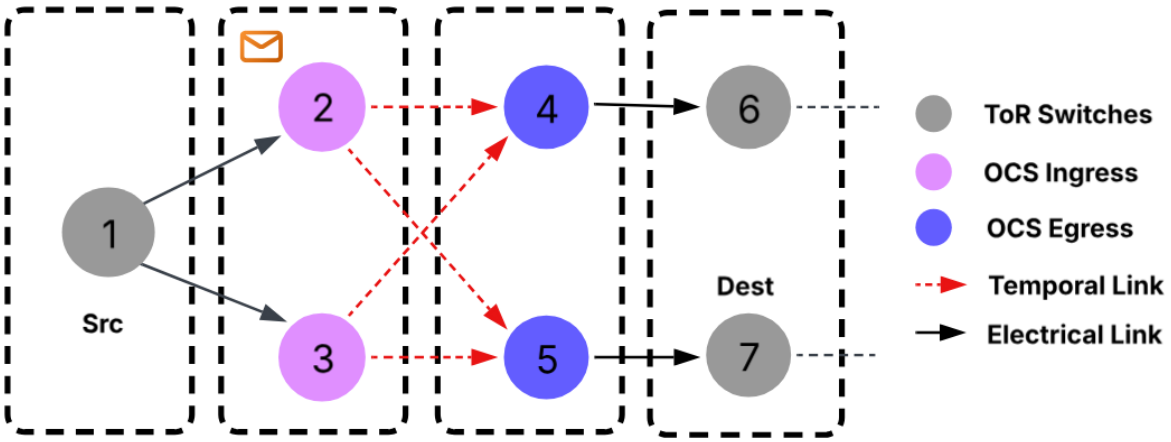

Figure 4.7: RDCN as a temporal graph

For example, in Figure 4.7, consider a packet originating at node ① and destined

for node ⑦, currently located at intermediate node ②. From node ②, there are two candidate next hops, nodes ④ and ⑤. Suppose the path via node ⑤ has fewer remaining hops to the destination, but the optical link from node ② to node ⑤ becomes active later than the link to node ④. If the waiting delay on link (2,5) is sufficiently large, the combined score (Eq. (4.18)) may exceed that of routing through node ④, even though node ④ leads to a longer spatial path. In this case, the algorithm selects node ④ to reduce the immediate delay. Conversely, if the waiting time on link (2,5) is small, the shorter-hop advantage is likely to dominate (based on the score) and node ⑤ is selected. This behavior illustrates how the heuristic balances both the earliest arrival and minimizing hop count.

Chapter 5

Simulator Architecture

5.1 Simulator Design

ClusterSIM, a simulator that is built in this work is built upon ASTRA-sim [26], an open-source execution-driven framework for distributed deep learning training, and extend it with over 5,000 lines of new C++ code to support in three dimensions that it does not natively address: scale-out across reconfigurable optical topologies, multi-job cluster scheduling with preemption, and a temporal routing co-simulation layer. ClusterSIM is structured as three distinct layers, each with a clearly defined responsibility, communicating with the layers adjacent to it through well-defined interfaces. This layered design makes it possible to study the interactions between training workloads and network behavior without either layer needing to know the internal details of the other.

The three layers are the **AstraSim Layer**, the **Network Layer**, and the **Control Plane**. Figure 5.1 illustrates the overall structure and how the layers relate to each other.

The simulator has a dedicated entrypoint that bootstraps the entire system. It reads the workload and network configuration files, initializes the Network Layer with the topology parameters, computes the GPU and ToR demand matrices and passes them to the Network Layer which internally creates the reconfiguration schedules and initializes the temporal graph. Once the network is fully initialized, the entrypoint brings up the AstraSim Layer and the Control Plane and triggers the start of training.

AstraSim Layer decides when a gradient is ready to be communicated, how it is split into chunks, and which collective algorithm governs its transmission. It does not know or care about the physical network underneath. When it needs to send a chunk from one

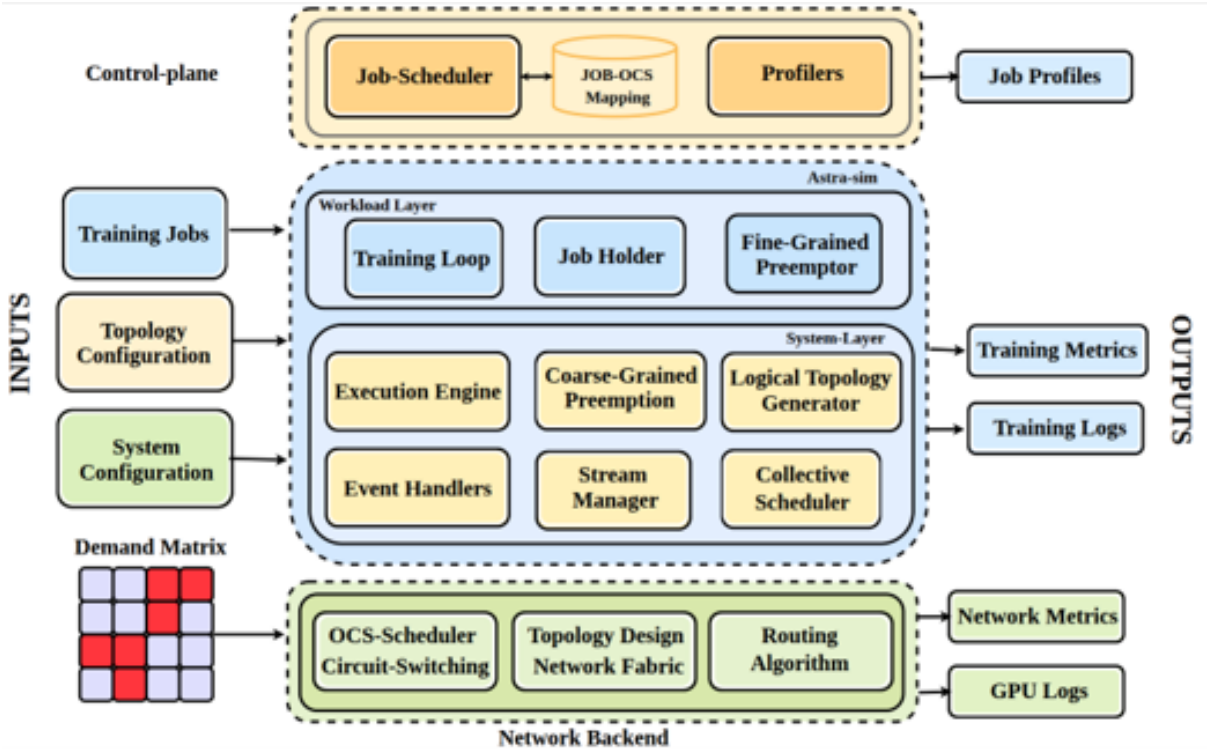

Figure 5.1: Simulator Architecture.

GPU to another, it simply calls `sim_send` and waits for a callback telling it the chunk has arrived.

The Network Layer sits at the bottom and is responsible for everything that happens to a chunk once it leaves a GPU and before it arrives at its destination. It models the physical devices in the cluster that includes GPUs, NVSwitches, ToR switches, and OCS switches along with the links that connect them and the queuing behavior at each device. It receives chunks from the AstraSim Layer through the `sim_send` interface, routes them through the topology according to pre-computed or on-demand paths, enforces the timing constraints imposed by the OCS schedule, and fires a receive callback when a chunk arrives at its destination GPU. The Network Layer knows nothing about the training loop above it; it only moves chunks from source to destination as faithfully as possible.

The Control Plane sits above both layers and is responsible for the global coordination of the simulation before it begins. During the simulation, the Control Plane also manages the job scheduler deciding which job gets access to the OCS at each time quantum, handling preemption, and recording completion times.

Together, these three layers cover the full system stack from GPU compute down to optical circuit scheduling. The following sections describe each layer in detail.

5.1.1 Network Backend

The network backend is the layer of the simulator responsible for modeling everything that happens to a chunk of data after it leaves a GPU and before it arrives at its destination GPU. It models the physical devices in the cluster, the links connecting them, the rules governing when data can be transmitted over each link, and the queuing behavior at intermediate devices. The system layer above it only knows that it called `sim_send` — everything that happens after that is handled entirely by the network backend.

Physical Topology and Device Hierarchy

The network backend represents the cluster as a hierarchy of physical devices. At the lowest level are the GPUs themselves. GPUs within the same server are connected to a shared NVSwitch, which provides high-bandwidth intra-server connectivity. Each GPU is also connected to a Top-of-Rack switch, referred to as a ToR switch, through an uplink. The ToR switch aggregates traffic from all servers in its rack and connects upward to one or more OCS switches through ingress and egress port devices. This four-level hierarchy – GPU, NVSwitch, ToR, OCS – is the physical skeleton of the simulated cluster.

Each device type has a distinct role and a distinct set of behavioral rules. A GPU or NVSwitch device has a per-destination output queue and a single link per destination. When a chunk arrives at a GPU destined for another device, it is placed into the appropriate output queue. If the outgoing link is free, the chunk is immediately scheduled for transmission. If the link is busy serving a previous chunk, the new chunk waits in the queue until the link-free event fires. This models head-of-line blocking at the GPU level in a straightforward way.

Links: Static and Temporal

Every connection between devices is modeled as a link object. There are two types of links. A static link, used for intra-server and intra-rack connections, is always available. When a chunk is placed on a static link, the link computes the serialization delay — the time it takes to put all the chunk's bits onto the wire, given the link's bandwidth — and schedules a chunk-arrival event at the destination device after serialization delay plus propagation latency. The link marks itself as reserved for the duration of the serialization, and when that period ends a link-free event fires, allowing the next queued chunk to be

transmitted. The serialization delay for a chunk of size B bytes over a link of bandwidth W is given by Eq. (5.1):

$$T_{\text{serialization}} = \frac{B \times 8}{W} \tag{5.1}$$

where W is expressed in bits per second. The total time for a chunk to traverse one static hop is therefore:

$$T_{\text{hop}} = T_{\text{serialization}} + T_{\text{latency}} \tag{5.2}$$

A temporal link is used for OCS-backed inter-rack connections. Unlike a static link, a temporal link carries a list of active intervals — time windows during which the OCS circuit it represents is live. Transmitting a chunk over a temporal link when the circuit is inactive is a hard error and triggers an assertion failure in the simulator. When a chunk does arrive on an active temporal link, no latency is modeled at the OCS ingress-to-egress hop — the OCS is treated as something that simply forwards the bits at the speed of light, so only delay that the chunk experience here is electrical-optical conversion modeled as a serialization delay. This is intentional: the OCS itself has no buffers (no queuing), offers seamless transfers because it does not process bits electronically.

The ToR Switch and the Calendar Queue

The ToR switch is the most behaviorally complex device in the network backend specially in these kind of reconfigurable networks. Unlike a GPU or an NVSwitch, a ToR switch must coordinate between the timing of incoming chunks and the timing of the OCS schedule. When a chunk arrives at a ToR switch destined for a remote rack, the ToR cannot simply forward it immediately — the OCS circuit to that destination may not be active yet.

To handle this, each ToR switch maintains a calendar queue. The calendar queue is a per-uplink-port, per-destination data structure that stores the active intervals of the OCS circuits reachable through each port. When a chunk arrives, the ToR performs a lookup: it identifies which OCS ingress port leads toward the destination rack, consults the calendar queue to find the earliest active interval for that (ingress port, destination ToR) pair that is no earlier than the current time, and computes the waiting time as the

gap between now and the start of that interval. The chunk is then placed into a Virtual Output Queue (VoQ) with an optional support for Push-In First-Out scheduling.

The calendar queue entries are not taken directly from the OCS schedule. They are adjusted by subtracting the worst-case traversal time — the time a chunk needs to travel from the ToR through the OCS and arrive at the destination ToR before the circuit expires. Formally, for each active interval $[t_{\text{start}},\ t_{\text{end}}]$ in the OCS schedule, the effective deadline stored in the calendar queue is given by:

$$t_{\text{deadline}} = t_{\text{end}} - T_{\text{worst-case traversal}} \tag{5.3}$$

This conservative adjustment ensures that a chunk is never released toward the OCS just as the circuit is about to close, which would cause it to traverse an inactive link.

The OCS Switch: Ingress and Egress Ports

The OCS switch itself is not a forwarding device in the usual sense. It is modeled as a collection of port devices — ingress ports and egress ports — connected internally through temporal links. Each ingress port is a lightweight device object identified by a name of the form `OCS_i_INGRESS_j`, and each egress port is similarly named `OCS_i_EGRESS_j`, where i is the OCS switch index and j is the port index corresponding to the connected ToR switch. For the purpose of packet-level simulation, when the OCS schedule is initialized, the `pair_internal_connection` method connects the ingress port of one ToR to the egress port of another ToR through a temporal link whose active intervals correspond to the time-slices assigned to that particular matching. Crucially, no buffering is modeled at OCS ingress ports. The simulator explicitly enforces this with a hard assertion: a chunk that arrives at an OCS ingress port must find the circuit active at that exact moment. If it does, it is forwarded immediately over the temporal link and only the propagation latency is charged. All buffering and waiting responsibility belongs to the ToR switch below, not the OCS above. This design reflects the physical reality of optical switching – an OCS port has no electronic memory and cannot hold a chunk while waiting for a circuit to open.

In contrast, for a flow-based simulation, the circuit timings are directly installed into the calendar queues of the ToR switches thereby completely abstracting away the OCS devices.

End-to-End Chunk Transmission

To make the interaction between all these components concrete, consider a chunk traveling from GPU u on rack r_1 to GPU v on rack r_2. The chunk carries a pre-computed route — an ordered sequence of device identifiers it will follow from source to destination. The chunk is first injected at GPU u, which places it into the per-destination output queue toward the next device in its route and forwards it to the ToR switch T_{src} of rack r_1 over a static intra-rack link. Upon arrival at T_{src}, the chunk triggers a calendar queue lookup to determine when the appropriate OCS circuit will be active. The chunk is then placed into the Virtual Output Queue $VoQ[p_{\text{in}}][T_1]$, where p_{in} is the OCS ingress port leading toward the destination and T_1 is the destination ToR switch identified from the route sequence. Rather than modeling each chunk as an independent packet-level event, the simulator employs a flow-based abstraction between T_{src} and T_1. When the chunk is enqueued, the simulator computes its expected arrival time at T_1 by accounting for bandwidth sharing among all flows currently occupying the same VoQ. A chunk arrival event is scheduled at that computed time. If a new chunk arrives at the same VoQ before the in-flight chunk has reached T_1, a flow completion time update is triggered for all in-flight flows in that queue to reflect the revised bandwidth allocation. As each flow completes its traversal and reaches T_1, if T_1 is the T_{dest} to which the destination GPU v is connected to, the chunk is placed onto the static downlink and forwarded directly to GPU v. If not, the same process will repeat until the chunk reaches T_{dest} and then sent to the destination GPU v. UpOn arrival at v, a receive callback fires, which advances the collective operation that was waiting for this chunk. So, the total end-to-end latency of the chunk is as shown in Eq. (5.4)

$$T_{\text{e2e}} = T_{(u->T_{src})} + T_{(T_{src}->T_1)} + ... + T_{(T_{dest}->v)} \tag{5.4}$$

Note that, $T_{(u->T_{src})}$ accounts for queuing delays at GPU-NIC and $T_{(T_i->T_{i+1})}$ accounts for queuing at T_i.

Event Priorities and Simulation Correctness

Every transition between devices is modeled as a discrete event. The event queue is the central scheduler that drives the entire simulation forward by processing events in

chronological order. When multiple events are scheduled at the same timestamp, they are processed according to a sorted ordering: link-free events are processed first (highest priority), followed by chunk-arrival events (medium priority), and metric tracking events are processed last (lowest priority). This ordering ensures that network state — specifically, which links are free and which chunks have arrived — is always updated before any tracking decisions are made at the same timestamp, preserving causal correctness throughout the simulation.

Integration of Routing Algorithm

Throughout the simulator design, the integration of the temporal routing algorithm proved to be the most challenging component. This stage exposed the gap between the theoretical efficiency of the algorithm and the practical constraints of large-scale simulation environments. In particular, practical considerations such as network scale and Fine-Grained temporal resolution introduced significant computational and system-level challenges that had to be carefully addressed.

Aspect-1 : Scalability The K-foremost path algorithm described in §4.5 computes the k fastest paths between a single source and a single destination by operating at discrete time instants. This introduces a significant computational challenge at scale. In practice, OCS time-slice widths are in the range of milliseconds ($10^6 ns$) to seconds ($10^9 ns$), while the simulator operates at nanosecond granularity. This means the time axis of the temporal graph spans a very large number of discrete steps, making the construction of matrices A and N expensive both in time and memory. As a result, the algorithm is far too slow to be used for online route computation, where paths would need to be computed on the fly as chunks are injected into the network.

The only practical option is therefore to compute all routes offline, before the simulation begins, and reuse the pre-computed paths during the run by simply looking them up. However, this requires computing paths for every source-destination ToR pair in the cluster. In a real datacenter with hundreds of ToR switches, the number of pairs grows quadratically. For example, a cluster with 80 ToR switches has ${}^{80}C_2 \approx 3200$ source-destination pairs, and a cluster with 160 ToR switches has nearly 12,800 pairs. Computing the full A and N matrices and running the K-foremost algorithm for each of

these pairs offline is a significant but one-time cost that is amortized across the entire training run.

Aspect-2 : Time-Sensitivity There is a further challenge related to the timing of individual chunks. The matrices A and N are computed with respect to a specific boundary time (i.e.) the time at which the chunk is assumed to depart from the source. When the matrices are computed at $t = 0$, the resulting paths are valid only for chunks that are injected into the network at $t = 0$. In practice, chunks are injected at arbitrary times throughout the training run, determined by when each layer finishes its backward pass and when the scheduler gives the job its next time quantum. A chunk injected at time $t = \tau$ will encounter a completely different state of the OCS schedule than one injected at $t = 0$. The active intervals it sees at each hop are shifted by τ relative to the ones the pre-computed matrices assumed, which means the pre-computed paths may route the chunk through circuits that are inactive at the time it actually arrives. The offline paths computed at $t = 0$ are therefore not directly reusable for chunks that depart at arbitrary times.

This time-sensitive nature of the algorithm and network also introduced challenges at the forwarding plane: if a chunk misses its scheduled time-slice due to upstream congestion, simply holding it in the output queue blocks all subsequent chunks behind it, while rerouting it requires the switch to perform a fresh lookup at runtime. To address this, the ToR switch ports are synchronized with the OCS schedule through calendar queues [11]. A chunk at the head of a queue is released only when its scheduled circuit is active, and if it misses a slot it is rescheduled to the next valid entry in the calendar rather than blocking the queue indefinitely.

Aspect-3 : Design-Specific Routing This problem arises specifically in the locally-aware algorithm. In a general temporal graph, temporal and non-temporal links are distributed arbitrarily along a path (i.e.) a chunk may traverse two or three static links followed by a temporal link, or vice versa, in no predictable order. However, the network topology used in this work is highly structured: the links from a ToR switch up to the OCS ingress ports are always static (exists for the whole period), and only the link from an OCS ingress port to its corresponding egress port is temporal. This structure has an important consequence for the locally-aware algorithm.

Consider a ToR switch connected to two OCS switches. From this ToR, there are two outgoing paths, each going through a different OCS switch. Both of these first hops are static links with zero waiting time (i.e.) the chunk can depart immediately on either of them. The locally-aware algorithm selects the next hop greedily based on the composite score in Eq. (4.18), which at this point is identical for both candidates since neither incurs any immediate waiting. The algorithm therefore always resolves this tie in favour of the same OCS switch, effectively ignoring the other path entirely. The temporal nature of the two routes only becomes visible one hop later, at the OCS ingress-to-egress link, by which point the routing decision has already been committed. As a result, all traffic from this ToR is consistently steered toward one OCS switch while the other remains underutilized. This systematic load imbalance cannot be corrected by the locally-aware algorithm because it does not look ahead past the immediate next hop, and so it never sees the difference between the two paths until it is too late to act on it.
The three aspects above motivated the following abstractions and design compromises, which together make the routing approach practical at scale.

1. **Graph-Conversion.** The raw network topology contains multiple OCS switches connecting the same pair of ToR switches, creating a multigraph, a graph where two nodes can be connected by more than one parallel edge. Standard shortest-path algorithms cannot distinguish between these parallel edges, meaning all traffic between a ToR pair would be routed through a single OCS switch regardless of load, causing the imbalance described in Aspect-3. To resolve this, we apply a two-step graph conversion shown in Figure 5.2. In the first step (Figure 5.2a), all OCS ports are replaced by parallel links between the corresponding ToR pairs. For example, in the figure, ToR-0 has two egress ports ($E1, E2$) and one ingress port (I0), and two separate OCS circuits – OCS-0 carrying flow $I_0 \rightarrow E_1$ and OCS-1 carrying flow $I_0 \rightarrow E2$ – connect ToR-0 to ToR-1 and ToR-2 respectively, forming parallel edges in the multigraph. In the second step (Figure 5.2b), the multigraph is converted into a simple graph by inserting a virtual intermediate node on each parallel edge. In the figure, the two parallel paths from ToR-0 are split through virtual nodes **0** and **1**, where node **0** corresponds to the OCS-0 path toward ToR-1 and node **1** corresponds to the OCS-0 path toward ToR-2. This transformation makes every OCS-switched path topologically distinct, allowing the routing algorithm to

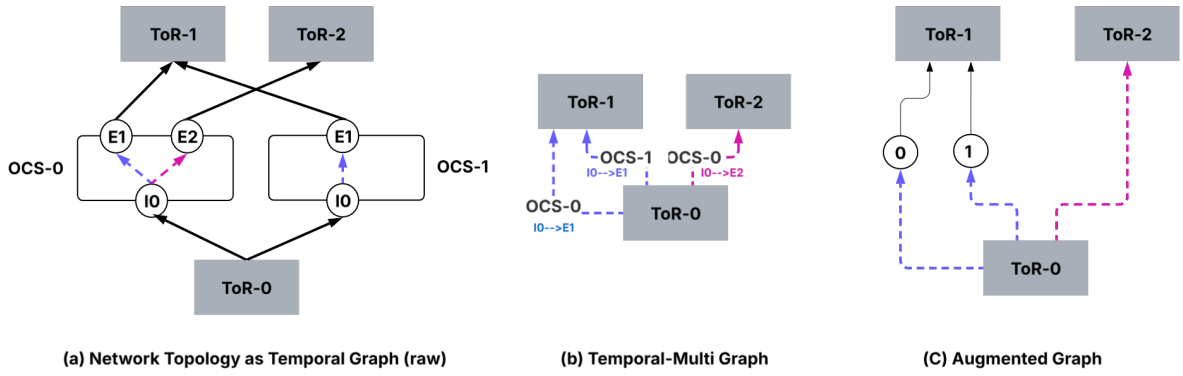

Figure 5.2: Temporal Graph Augmentation

independently evaluate and select paths through different OCS switches, thereby enabling load-aware routing and resolving the parallel-edge ambiguity.

2. **Normalized traversal time.** To keep the time axis tractable and to simplify normalization, we assume that the traversal time between any two adjacent nodes is exactly one time-slice, corresponding to one OCS time-slice width which means traveling from one ToR to another ToR takes 1 time-slice once the circuit is available (for the routing algorithm). This worst-case assumption bounds the number of discrete time steps the algorithm must reason about and makes the normalization of the time axis straightforward, since all edge weights are now uniform and divide evenly into the time axis.

3. **Paths at all normalized time instants.** To address the problem of chunks being injected at arbitrary times, paths are pre-computed at every normalized time instant on the reduced graph. Since the OCS schedule is periodic, there are only as many distinct time instants as there are time-slices in one schedule period. At runtime, a chunk injected at time τ simply looks up the pre-computed path corresponding to the normalized time instant that matches τ modulo the schedule period, ensuring that the retrieved path is approximately valid for the actual network state the chunk will encounter.

Note

For the problem of a chunk missing its scheduled time-slice, no perfect solution currently exists. Prior work [11] addresses this challenge by leveraging the multi-stage processing pipeline available in programmable switches, which typically support 12–

20 sequential lookup stages before making the final forwarding decision. At each stage, the switch examines the occupancy state of a candidate output port and determines whether sufficient buffer headroom exists to accommodate the incoming chunk without delaying it beyond the next available time-slice. The admissible buffer threshold is precomputed through offline network engineering based on expected traffic characteristics. If the buffer occupancy remains below this threshold, the chunk is forwarded through that port; otherwise, the pipeline advances to the next lookup stage and evaluates an alternative output port. This threshold-driven multi-stage lookup mechanism enables missed-slot recovery to be handled entirely within the switch pipeline without stalling the forwarding process.

5.1.2 The AstraSim Layer

The AstraSim Layer is the topmost layer of the simulator. It is responsible for modeling everything that happens inside the GPUs: the training loop, the collective communication operations that arise from it, and the lifecycle of every gradient from the moment a gradient is ready to the moment the last chunk of that gradient is acknowledged as received. This layer has no knowledge of how the network physically works. It expresses all communication as calls to `sim_send` and `sim_recv`, trusting the Network Layer below to deliver chunks correctly and call back when they arrive.

The Kernel

The central object of the AstraSim Layer is the `Kernel`. Each GPU in the simulated cluster is represented by one `Kernel` instance. The Kernel owns everything that belongs to that GPU: its training model, its stream scheduler, its logical topology definitions, its local event queue, and its connection to the Network Layer through a `NetworkAPI` object.

When a Kernel is constructed, it reads a system configuration file that sets key behavioral parameters. It then constructs a `Model` object for the training workload assigned to this GPU, a `SchedulerUnit` that manages virtual queues for active streams, and four `GeneralComplexTopology` objects — one each for AllReduce, ReduceScatter, AllGather, and AllToAll — that define the logical communication structure for each collective type.

One GPU per job is designated as the master GPU. The master GPU is responsible

for logging trace events — recording when each compute phase starts and ends, when each stream is created, blocked, resumed, or completed, and when the job is paused or resumed by the scheduler.

The Model and the Training Loop

The `Model` class represents the DNN workload assigned to a GPU. It holds an array of `Layer` objects — one per layer of the neural network — and drives the training loop by iterating through them. The training loop is a state machine with three repeating states: `FORWARD_PASS`, `INPUT_GRAD`, and `WEIGHT_GRAD`. The loop begins at the first layer in the forward pass state and advances through layers and states in sequence, repeating for a configured number of passes.

At each state transition, the Model calls the `call()` method, which dispatches to one of the parallelism-specific iteration functions depending on the configured policy: `iterate_hybrid_parallel_data_model()`, `iterate_data_parallel()`, `iterate_model_parallel()`. Each of these functions implements the correct sequence of compute phases and collective operations for the corresponding parallelism strategy.

Within each iteration function, the sequence is the same in structure. First, the function checks whether the blocking collective (refer to Section §2.1.5) from the previous state has finished. If it has not, the function returns immediately and waits for a callback. If it has, it loads the compute time for the current phase and registers a timed event using `try_register_event`, which will fire after the specified number of simulation cycles. When the event fires, the compute phase is marked as complete. Then, depending on the parallelism policy and the preemption state of the GPU, the collective for this phase is either issued immediately or deferred into a `pending_comms` queue to be issued upon job resumption.

The Model tracks simulation progress through a pass counter and a layer index. When all passes are complete and all streams have finished, it calls `workload_finished()`, which notifies the CentralController that this GPU's job is done.

Layers

Each `Layer` object encapsulates the communication specification of one neural network layer. It stores, for each of the three communication phases (forward pass, input gradient,

weight gradient), the compute time, the collective operation type, the data size, the involved parallelism dimensions, and the barrier type (blocking or non-blocking).

When the Model reaches a layer's communication phase, it calls `issue_fwd_pass_comm()`, `issue_input_grad_comm()`, or `issue_weight_grad_comm()` on the Layer. Each of these methods calls into the Kernel's `generate_collective()` method, passing the collective type, data size. The Layer also tracks whether the collective has finished using a `Gradient` object, and exposes methods like `is_weight_grad_comm_finished_blocking()` that the Model polls to check whether it can advance to the next state.

Collective Generation and the Stream

When the Kernel's `generate_collective()` method is called, it performs the following steps. First, it determines the total data size and splits it into k chunks, where k is the configured number of dataset splits (the pipelining parameter). For each chunk, it creates a `StreamBaseline` object. Each stream is assigned a priority based on the scheduling policy and enqueued into the appropriate virtual queue in the `SchedulerUnit`.

A `StreamBaseline` carries a list of `CollectivePhase` objects — one phase per active dimension in the logical topology. Each `CollectivePhase` holds a pointer to an `Algorithm` object that implements the specific collective for that dimension. The Algorithm objects are: `Ring` for ring-based AllReduce, ReduceScatter, and AllGather; `AllToAll` for all-to-all operations; and `DoubleBinaryTreeAllReduce` for tree-based AllReduce. The inter-dimension scheduling policy determines the order in which dimensions are activated: with round-robin scheduling, the starting dimension is rotated for each successive stream, so that consecutive chunks begin on different dimensions and spread their traffic across different OCS time-slices.

The Stream Lifecycle

Once a stream is created and placed in the ready queue, the `SchedulerUnit` initiates it by calling `proceed_to_next_vnet_baseline()`. This function first checks whether the stream is ready to proceed by consulting the `synchronizer` map. The synchronizer tracks, for each stream number, how many GPUs across the collective have completed the current phase. A stream can only proceed to the next phase once all GPUs in

the collective have finished the current one — this enforces the collective synchronization barrier. If the stream is not yet ready because some other GPU in the collective is still finishing the current phase, the stream is moved into the `blocked_streams` list and waits. When the last GPU finishes and calls `declare_ready()`, the ready counter reaches the required count and all blocked streams for that stream number are simultaneously released and passed back to `proceed_to_next_vnet_baseline()`. When the stream is ready to proceed, `proceed_to_next_vnet_baseline()` advances it to its next `CollectivePhase`. Before doing so, it checks whether the owning GPU is currently paused by the job scheduler. What happens at this point depends on the configured preemption strategy. Under Coarse-Grained preemption, the stream does not check preemption state at this level — the preemption boundary is at the collective level in the Model, not inside the stream. Under Fine-Grained preemption, `consume_ready()` checks the `is_paused` flag on the GPU before advancing the stream. If the GPU is paused and the stream has not yet started any steps in the current phase, the stream is added to `BaseStream::paused_streams` to be restarted from the beginning of that phase when the job resumes. If the GPU is paused but the stream has already started some steps in the current phase, it is added to `BaseStream::pending_streams` to be resumed from the point of interruption. In both cases, the stream stops progressing until `Kernel::resume_pending_streams()` is called by the Control Plane when the job is given its next time quantum. If there are no more phases left in the stream, the stream is marked as dead and calls `Gradient::notify_stream_finished()`. When all k k streams of a gradient have finished, the Gradient notifies the owning Layer, which may unblock the next training phase. Each active phase is executed by its Algorithm object. The `Ring` algorithm models the ReduceScatter and AllGather steps of a ring collective. It tracks the number of remaining steps and at each step calls `sim_send()` to inject a chunk into the network and `sim_recv()` to register a receive expectation. The toggle flag in the Ring distinguishes ReduceScatter steps — which involve a local reduction and incur a `local_reduction_delay` — from AllGather steps, which are pure data copies. When all ring steps for the current phase are complete, the Algorithm calls `declare_ready()`, which updates the synchronizer for this stream and triggers the release of any blocked streams waiting on this stream number across all participating GPUs.

Interaction with the Network Layer

The boundary between the AstraSim Layer and the Network Layer is the `sim_send` and `sim_recv` interface. When `sim_send` is called, the Kernel either calls `NetworkAPI::sim_send()` immediately (if the delay is zero) or registers a timed event that will call it after the specified delay. The `NetworkAPI::sim_send()` method computes a flow hash from the source, destination, time, and tag, uses it to look up or compute a route through the topology, constructs a `Chunk` object with that route, and injects it into the network. On the receive side, `sim_recv()` registers a callback in the `CallbackTracker`. When the corresponding chunk arrives at the destination GPU, the Network Layer fires the callback, which the Kernel uses to advance the collective algorithm. The AstraSim Layer is completely unaware of whether the chunk waited in a ToR queue for an OCS time-slice, traversed one or three hops, or was delayed by a missed time-slot. All of that is handled transparently by the Network Layer.

5.1.3 The Control Plane

The Control Plane is the upper most conceptual layer of ClusterSIM. It does not model training behavior and it does not model network packet forwarding. Its role is global coordination: it owns the state that is shared across all GPUs and all jobs, makes all scheduling decisions, and is responsible for both the pre-simulation setup phase and the runtime orchestration of job execution. Concretely, the Control Plane is implemented as the `CentralController` class, which is a singleton accessible to every `Kernel` via the static pointer `Kernel::controller`.

Global State

The CentralController maintains several pieces of global state that no individual GPU or network component can own independently. It holds `sys_list_`, a flat array of Kernel pointers indexed by GPU ID, which allows any component to look up the Kernel of any GPU in constant time. It holds `job_to_mastergpu`, a map from job name to the ID of the master GPU for that job. It holds `jobs_to_participating_GPUs`, a map from job name to the list of all GPU IDs involved in that job, which is used by the synchronizer in the AstraSim Layer to know how many GPUs must finish a collective phase before it

can advance. It holds the preemption strategy, the training time budget, the number of OCS switches, and a reference to the network topology.

Job Lifecycle Management

The Control Plane manages the full lifecycle of a training job from submission to completion. Before training starts, jobs are submitted to the Control Plane via `add_job()`, which registers a `JobEventTracer` for the job (for trace logging) and adds the job to the job scheduler's upcoming queue. When `start_training()` is called, the Control Plane calls `job_scheduler->schedule()`, which admits all jobs into the ready queue, selects the first job for each OCS, and fires the first quantum expiry event. If a training time budget is configured, the Control Plane also schedules a termination event at `current_time + training_time_budget`, after which all jobs are forcibly stopped regardless of completion status.

When a GPU finishes its workload (all passes complete, all streams delivered), it calls `Kernel::workload_finished()`, which calls `CentralController::handle_job_end()`. This in turn notifies the job scheduler that the job is finished, which removes it from the ready queue and records its completion time.

The Job Scheduler and Round-Robin Policy

The Control Plane owns a `ClusterScheduler` instance that manages the time-multiplexing of training jobs onto OCS switches. The scheduler maintains, for each OCS switch, an upcoming queue (jobs not yet admitted), a ready queue (jobs currently eligible to run), and a current iterator pointing to the job that is presently running on that OCS.

The scheduling policy is implemented by a `RoundRobinPolicy` object. Under this policy, jobs take turns using the OCS in a circular order. The duration of each turn is called the slot duration, and it is determined by the policy name. Under the serial intra-OCS scheduling strategy, a job's slot duration is `num_slices` $\times$ `quantum`, meaning the job holds the OCS for all of its matchings in one contiguous block before yielding. Under the interleaved strategy, the slot duration is exactly one quantum — one time-slice width plus one reconfiguration delay — so that matchings from different jobs alternate at the finest possible granularity.

The quantum expiry mechanism drives the scheduler. When a job's slot expires,

the Kernel of the master GPU for that job fires a `JOB_SCHEDULE` event, which calls `ClusterScheduler::on_quantum_expiry()`. This function pauses the current job, removes any completed jobs from the ready queue, calls `RoundRobinPolicy::select_next_job()` to choose the next job, and then resumes that job by calling `runtime.resume(next_job)` and scheduling the next quantum expiry event. The `RunTimeController` object inside the ClusterScheduler is responsible for translating pause and resume decisions into actual calls to `Model::pause()` and `Model::resume()` on every Kernel participating in the job.

Preemption Handling in the Control Plane

When the scheduler decides to pause a job, the behavior depends on the configured preemption strategy, which the Control Plane communicates to the AstraSim Layer via `get_preemption_strategy()`. Under Coarse-Grained preemption, the `is_paused` flag is set on the GPU's Kernel, and the Model checks this flag before issuing each collective. If the flag is set, the collective is not issued immediately; instead, a `PendingComm` entry is pushed onto the `pending_comms` queue, recording the communication phase, the layer index, and the barrier type. When the job is resumed, `Model::resume()` drains the `pending_comms` queue, issuing each deferred collective in order, and then re-enters the training loop.

Under Fine-Grained preemption, pausing can happen mid-stream. When `consume_ready()` is called on a stream whose owner's GPU is paused, the stream does not advance to the next phase. Instead, if the stream has not yet started any steps in the current phase, it is added to `BaseStream::paused_streams` to be resumed from scratch. If it has already started some steps, it is added to `BaseStream::pending_streams` to be resumed from the point of interruption. When the job is resumed, `Kernel::resume_pending_streams()` is called, which iterates over all paused and pending streams and calls `proceed_to_next_vnet_baseline()` on each, continuing their execution exactly where they left off.

Termination and Reporting

When the simulation ends — either because all jobs have completed or because the training time budget has expired — the Control Plane's `handleTermination()` function is called. This function first clears all remaining events from the event queue to prevent

any further simulation activity. It then iterates over all completed jobs and all jobs that were still active at termination, calls `workload->report()` on each job's master Kernel to collect per-layer statistics (total compute time, exposed communication time, average chunk latency, average network message latency), and writes the results to CSV files. The output includes one row per job covering job completion time, percentage of training completed, parallelism type, number of streams per gradient, and a detailed backend report generated by the Network Layer covering per-link utilization, cross-job OCS traffic statistics, and queue depth traces.

Chapter 6

Experimental Evaluation

We present preliminary results to validate simulator correctness and study key trade-offs in reconfigurable optical fabrics for distributed AI training across multiple system and workload configurations. Specifically, we examine communication scalability, the effects of OCS reconfiguration, and multi-job scheduling behavior under a shared optical infrastructure.

Experimental Setup. All experiments are conducted on a simulated cluster of up to 512 GPUs (8 GPUs/server) organized across multiple pods, each consisting of 5–8 servers and 8 ToR switches interconnected via a shared OCS with a fixed reconfiguration delay of $500ns$ unless until explicitly mentioned. We assume a rail-optimized design [33], where GPUs within a server are distributed across different ToR uplinks to improve bandwidth utilization and communication efficiency. For *multi-job* experiments, we use six concurrently arriving training workloads, each having one transformer block. For *single-job* experiments, we use a small language model (SLM) workload consisting of 16 transformer blocks and distributed across 512 GPUs using an HP strategy (refer to Section § 2.1.1). All workloads are obtained from the ASTRA-sim repository [28]. We primarily report the mean Job Completion Time (JCT) and the spread in JCT across concurrent jobs.

Impact of time-slice width (TSW) on JCT. Achieving full alignment between the circuit schedule and runtime collective execution is fundamentally infeasible without per-flow timing information that is unavailable at scheduling time. TSW is therefore the primary tuning parameter that can influence this alignment. A fundamental question then is how sensitive training performance is to this choice, and whether a well-chosen

TSW can recover the alignment between communication pattern and circuit schedule. We characterize this trade-off by training a single model distributed across a 512-GPU cluster (8 pods, 1 OCS switch) while sweeping TSW over $\{2, 4, 6, 8, 10, 12, 20\}$ μs.

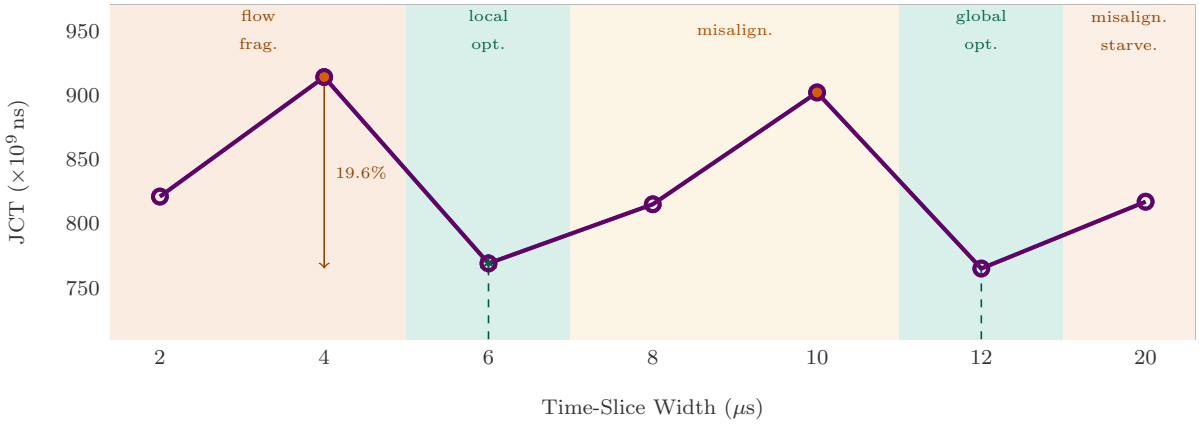

Figure 6.1: JCT versus TSW for a single training job.

Figure 6.1 reveals a *non-monotonic* relationship between TSW and JCT. The non-monotonic behavior arises from the interaction between the TSW and the per-chunk transmission time (t_{chunk}) of collective communication. When TSW $< t_{chunk}$, flows become fragmented across multiple circuit windows, causing repeated re-queuing and additional ToR contention. For example, at TSW$= 2\mu s$, fragmented flows recover relatively quickly due to shorter inter-slot gaps, whereas at $4\mu s$, flows still cannot complete within a single slot but experience longer waiting periods between slots, producing the first JCT peak.

As TSW increases beyond t_{chunk} but remains misaligned with chunk boundaries, residual idle periods emerge at the end of circuit windows. During these intervals, the circuit remains allocated, but subsequent communication phases cannot proceed because they require different circuit configurations. In our example, TSW of $6\mu s$ provides favorable alignment and produces a local minimum, whereas $8\mu s$ and $10\mu s$ are too large for one chunk but insufficient for two, creating idle tails that increase JCT and produce the second peak.

More generally, JCT minima occur when TSW $\approx k \cdot t_{chunk}$ for integer k, as circuit allocation aligns efficiently with workload communication structure. The global minimum at $12\mu s$ reflects the best alignment for this workload. Beyond this point, similar periodic behavior persists as more chunks fit within each slot, although sensitivity to TSW increases and small misalignments can rapidly negate the benefits. Consequently, the optimal TSW

is workload dependent and varies with message size, collective implementation, and link bandwidth, motivating adaptive TSW selection as future work.

Impact of Reconfiguration Delays. The single-job experiment above reveals that TSW selection is a schedule–communication alignment problem: too-short slices interrupt in-flight flows, while too-long slices leave a finished job's circuit idle, starving subsequent collective phases. This behavior is primarily a single-workload artifact because the starvation effect observed at large TSW values arises when there are no competing jobs available to utilize circuits that have been vacated by completed communication phases. In a multi-job setting, this changes, as a circuit released mid-phase by one job can be immediately claimed by another job whose traffic maps to the same ingress–egress pair, replacing what would be an idle period with productive transmission. In this experiment, we study how increasing TSW amortizes reconfiguration overhead and improves JCT through greater cross-job circuit reuse. We evaluate this on a 440-GPU cluster with a fixed 1 μs reconfiguration delay per circuit-switching event while sweeping TSW over $\{6, 10, 20, 40\}$ μs.

Figure 6.2 shows JCT decreasing monotonically with TSW. Two effects drive this. First, increasing TSW reduces the relative impact of the OCS reconfiguration delay (overhead), reducing the fraction of time lost to circuit reconfiguration. Second, and specific to the multi-job setting, larger TSWs increase the window during which a vacated circuit can be reused by a co-scheduled job, suppressing reconfiguration events that would otherwise be triggered by job-phase boundaries.

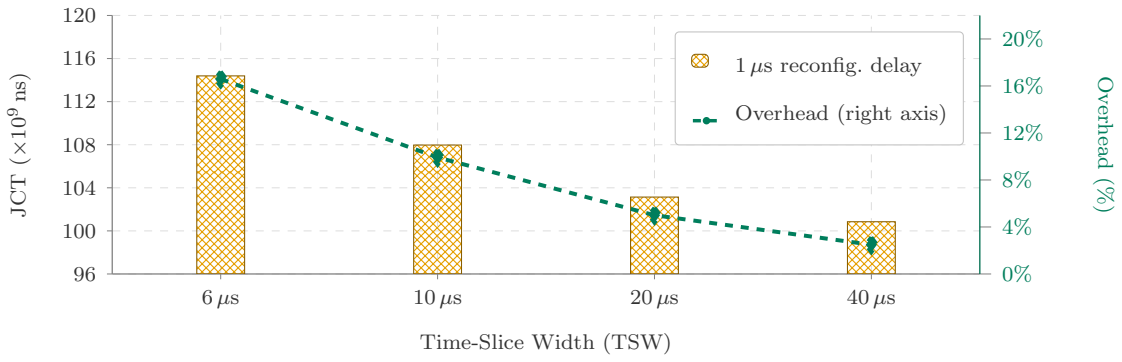

Figure 6.2: JCT (bars, left axis) and reconfiguration overhead (line, right axis) as a function of TSW under a 1 μs reconfiguration penalty.

Preemption policies under multi-job contention. Our next evaluation examines how different preemption strategies affect performance and fairness when multiple train-

ing jobs share the same OCS fabric. We evaluate fairness across different preemption policies by measuring job completion time distribution when six training jobs share the same OCS fabric while varying the number of OCSes from 1 to 6. OCS count forms an important design dimension in multi-job training, as it determines the switching capacity available for overlapping collective communication and influences contention, scheduling flexibility, and resource sharing efficiency [35]. Fairness is quantified using the spread in completion times across the jobs: a lower spread indicates a more balanced and fair allocation of network resources, whereas a higher spread reflects greater imbalance and poorer fairness under contention.

In this setting, competing jobs contend for the same OCS time slots, and the preemption policy determines how conflicts are resolved at quantum boundaries. We compare three policies: No-Preemption (NP), which allows jobs to retain circuits for an entire quantum and may cause monopolization, and Coarse-Grained (CG) and Fine-Grained (FG) preemption policies discussed in § 4.4.

Figure 6.3 shows the JCT spread across different preemption policies and OCS configurations. Overall, the JCT spread decreases as the number of OCSes increases due to reduced network contention and improved communication parallelism. However, in the 1-OCS configuration, the overhead introduced by preemption is higher than the contention overhead itself, leading to network underutilization and higher JCTs compared to the NP case. Among the evaluated approaches, FG consistently performs better than CG by providing lower JCTs and improved fairness. The benefit of FG is most significant in the 2-OCS configuration, where each OCS is effectively shared among three jobs, providing a *balanced* trade-off between network utilization and preemption overhead while allowing two jobs to progress in parallel. As the number of OCSes increases beyond two, jobs become more distributed across the available OCSes and gradually start receiving relatively exclusive OCS access, which reduces the number of preemptions experienced by certain jobs, inherently introducing allocation bias. Consequently, the improvement in the spread of the JCT gradually decreases with additional OCSes, indicating that increasing the number of OCSes beyond a point introduces an asymmetric job-to-OCS pinning that *hurts* fairness. The 2-OCS configuration with Fine-Grained preemption thus provides the best balance of performance, fairness, and cost efficiency for this workload.

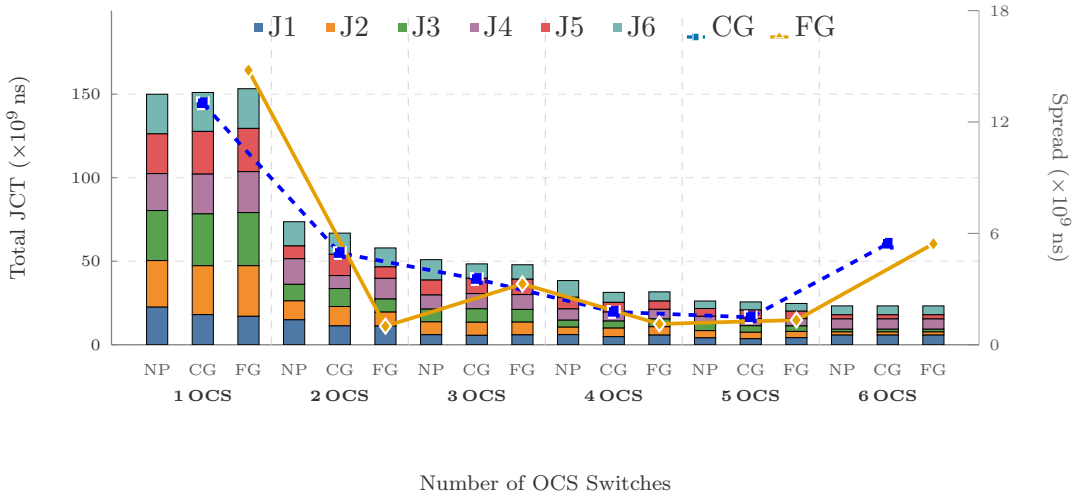

Figure 6.3: JCT per job (stacked bars, left axis) and JCT spread across jobs (lines, right axis) under different preemption strategies (NP, CG, FG) as OCS count scales from 1 to 6.

Effectiveness of temporal routing algorithms. We now evaluate our routing algorithm, which balances two competing objectives through a tunable parameter $\alpha \in [0, 1]$: the *waiting cost* at the ToR (i.e., time until the next circuit window becomes available) and the *hop count* to the destination. Setting $\alpha = 0$ reduces the algorithm to earliest-departure routing, while $\alpha = 1$ reduces it to static shortest-path routing that ignores circuit availability.

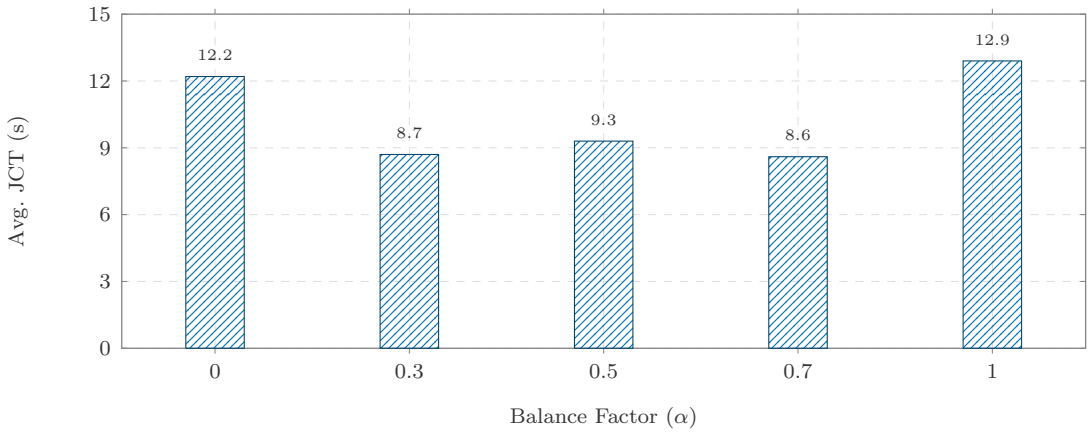

Figure 6.4: Average JCT as a function of the routing balance factor α.

Figure 6.4 shows that *both extremes are suboptimal*. Pure waiting-cost routing ($\alpha = 0$) achieves 12.2 s average JCT. It greedily picks the next available circuit window but may route through unnecessarily long paths, increasing propagation and serialization delay. Pure hop-count routing ($\alpha = 1$) performs even worse at 12.9 s, as it selects the shortest path without considering circuit availability, causing chunks to queue at ToR switches for future circuit windows that could have been avoided by taking a slightly longer path.

The optimal lies when $\alpha \in (0.3, 0.7)$ achieves a better balance between hop length and waiting cost. This confirms that a balanced choice of α yields a 33% JCT reduction over the pure hop-count baseline in a multi-job setting.

Chapter 7

Conclusions and Future Work

In this paper, we explored a previously unaddressed optical design space for distributed AI training over reconfigurable data center networks. Although reconfigurable optical data centers have been widely advocated for AI workloads, existing tools do not capture the interaction between collective communication patterns and optical circuit constraints. We therefore significantly extended a state-of-the-art distributed DNN training simulator to capture collective communication behavior, circuit reconfiguration dynamics, and multi-job contention in shared clusters. Using this framework, we analyzed the challenges that arise when integrating Optical Circuit Switches (OCS) into DNN training environments. To mitigate these issues, we propose several scheduling, routing, and circuit management mechanisms for OCS-based shared training clusters. Beyond individual algorithmic contributions, this work provided practical insights into performance behavior in OCS-enabled AI clusters. Several important research directions emerge from this work. First, more adaptive routing strategies can be developed to better exploit transient circuit availability in multi-job environments. Second, integrating realistic transport protocols and control-plane handshakes into the simulator would enable the study of protocol-level interactions with optical switching and improve deployment fidelity. Finally, we plan to analyze the impact of stragglers and heterogeneous workloads on collective performance under dynamic connectivity.

Bibliography

[1] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, et al. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 782–797, 2020.

[2] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[3] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 339–350, New York, NY, USA, 2010. Association for Computing Machinery.

[4] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[5] Yi-Xiang Hu, Han Tian, Yifang Zhao, Feng Wu, and Xiang-Yang Li. Topo-x: Co-optimize flow scheduling, topology, and ml training parallelism. In *2025 34th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–6, 2025.

[6] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. Elastic resource sharing for distributed deep learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 721–739, 2021.

[7] 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 *Proceedings of the 50th annual international symposium on computer architecture*, pages 1–14, 2023.

[8] Mehrdad Khani, Manya Ghobadi, Mohammad Alizadeh, Ziyi Zhu, Madeleine Glick, Keren Bergman, Amin Vahdat, Benjamin Klenk, and Eiman Ebrahimi. Sip-ml: high-bandwidth optical network interconnects for machine learning training. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 657–675, 2021.

[9] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. *SIGARCH Comput. Archit. News*, 36(3):77–88, June 2008.

[10] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. {ATP}: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761, 2021.

[11] Yiming Lei, Federico De Marchi, Jialong Li, Raj Joshi, Balakrishnan Chandrasekaran, and Yiting Xia. Openoptics: An open research framework for optical data center networks, 2025.

[12] Charles E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10):892–901, October 1985.

[13] Jialong Li, Federico De Marchi, Yiming Lei, Raj Joshi, Balakrishnan Chandrasekaran, and Yiting Xia. Unlocking diversity of fast-switched optical data center networks with unified routing. *IEEE Transactions on Networking*, 34:33–48, 2026.

[14] 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 *Proceedings of the 8th Asia-Pacific Workshop on Networking*, APNet '24, page 1–8, New York, NY, USA, 2024. Association for Computing Machinery.

[15] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Xudong Liao, Yijun Sun, Han Tian, Xinchen Wan, Yilun Jin, Zilong Wang, Zhenghang Ren, Xinyang Huang, Wenxue Li, Kin Fai Tse, et al. Mixnet: A runtime reconfigurable optical-electrical fabric for distributed mixture-of-experts training. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 554–574, 2025.

[17] Banruo Liu, Mubarak Adetunji Ojewale, Yuhan Ding, and Marco Canini. Towards a flexible and high-fidelity approach to distributed dnn training emulation. In *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 88–94, 2024.

[18] Hong Liu, Ryohei Urata, Kevin Yasumura, Xiang Zhou, Roy Bannon, Jill Berger, Pedram Dashti, Norm Jouppi, Cedric Lam, Sheng Li, et al. Lightwave fabrics: at-scale optical circuit switching for datacenter and machine learning systems. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 499–515, 2023.

[19] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: fair and efficient gpu cluster scheduling. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 289–304, USA, 2020. USENIX Association.

[20] William M. Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C. Snoeren, and George Porter. Expanding across time to deliver bandwidth efficiency and low latency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1–18, Santa Clara, CA, February 2020. USENIX Association.

[21] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. Rotornet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 267–280, New York, NY, USA, 2017. Association for Computing Machinery.

[22] Raul Puri Patrick LeGresley Jared Casper Bryan Catanzaro Mohammad Shoeybi, Mostofa Patwary. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[23] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[24] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, et al. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 66–85, 2022.

[25] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. {CASSINI}:{Network-Aware} job scheduling in machine learning clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1403–1420, 2024.

[26] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. Astra-sim: Enabling sw/hw co-design exploration for distributed dl training platforms. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 81–92. IEEE, 2020.

[27] George F. Riley and Thomas R. Henderson. The ns-3 network simulator. In *Modeling and Tools for Network Simulation*, pages 15–36. Springer Berlin Heidelberg, 2010.

[28] Sudarshan Srinivasan Saeed Rashidi, Srinivas Sridharan and Tushar Krishna. Astrasim: Distributed ai training network simulator. https://github.com/astra-sim/astra-sim/tree/ASTRA-sim-1.0/inputs/workload, 2020.

[29] Olatunji Ruwase Yuxiong He Samyam Rajbhandari, Jeff Rasley. Zero: Memory optimizations toward training trillion parameter models. *arXiv preprint arXiv:1910.02054*, 2020.

[30] Ryohei Urata, Hong Liu, Kevin Yasumura, Erji Mao, Jill Berger, Xiang Zhou, Cedric Lam, Roy Bannon, Darren Hutchinson, Daniel Nelson, Leon Poutievski, Arjun Singh, Joon Ong, and Amin Vahdat. Mission apollo: Landing optical circuit switching at datacenter scale, 2022.

[31] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques*, page 60. ICST, 2008.

[32] John Peach Vijay Vusirikala. Ai networking. White paper, Arista-Networks Inc., 2025.

[33] 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. In *2024 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 1–10. IEEE, 2024.

[34] 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 *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, 2023.

[35] Changbo Wu, Zhuolong Yu, Gongming Zhao, and Hongli Xu. Enabling reconfiguration-communication overlap for collective communication in optical networks. *arXiv preprint arXiv:2510.19322*, 2025.

[36] Zhenguo Wu, Benjamin Klenk, Larry Dennison, and Keren Bergman. Actina: Adapting circuit-switching techniques for ai networking architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1211–1222, 2025.

[37] Donglin Yang, Wei Rang, and Dazhao Cheng. Mitigating stragglers in the decentralized training on heterogeneous clusters. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 386–399. Association for Computing Machinery, 2020.

[38] Jinsun Yoo, William Won, Meghan Cowan, Nan Jiang, Benjamin Klenk, Srinivas Sridharan, and Tushar Krishna. Towards a standardized representation for deep learning collective algorithms. *IEEE Micro*, 2025.

[39] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis CM Lau, Yuqi Wang, Yifan Xiong, et al. {HiveD}: Sharing a {GPU} cluster for deep learning with guarantees. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 515–532, 2020.

[40] Anny Xijia Zheng, Remy Chang, and Leon Poutievski. Scaling ml workloads with google's evolving datacenter network architecture. In *2025 Optical Fiber Communications Conference and Exhibition (OFC)*, pages 1–3. IEEE, 2025.

[41] Ai Zhong, Gongming Zhao, Hongli Xu, Jin Fang, Jiawei Liu, and Peng Yang. Network resource-aware multi-job deployment in deep learning clusters. In *2025 34th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9. IEEE, 2025.

[42] 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 *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 761–774, 2024.