A Performance Evaluation of Collective Communication Libraries

by

Subiksha Srinivasan
B.E., Anna University, 2023

A Project Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We acknowledge and respect the Lək^{w}əŋən (Songhees and X^{w}sepsəm/ Esquimalt) Peoples on whose territory the university stands, and the Lək^{w}əŋən and W̱ SÁNEĆ Peoples whose historical relationships with the land continue to this day.

A Performance Evaluation of Collective Communication Libraries

by

Subiksha Srinivasan
B.E., Anna University, 2023

Supervisory Committee

Dr. Kui Wu, Supervisor
(Department of Computer Science)

Dr. Jaya Prakash Champati, Departmental Member
(Department of Computer Science)

ABSTRACT

Collective communication operations such as AllGather and AlltoAll are fundamental to high-performance computing (HPC) and large-scale machine learning workloads. Their performance, however, is tightly constrained by network structure, link latency, and bandwidth availability across modern multi-GPU and multi-node systems. As systems scale and become increasingly heterogeneous, traditional collective scheduling approaches, which often assume unrealistic symmetry in latency and topology, become ineffective.

This project investigates Traffic Engineering for Collective Communication (TE-CCL), an optimization-based framework that formulates collective scheduling as a Mixed-Integer Linear Programming (MILP) problem. TE-CCL explicitly incorporates link-level latency (α) into its scheduling formulation, enabling more realistic modelling of heterogeneous multi-fabric GPU clusters. This project examines how varying α across links affects routing decisions, epoch schedules, and solver behaviour. By introducing heterogeneous α values—rather than assuming a fixed latency across all links—the model adapts its schedules to prioritize low-latency paths, reduce hop count where beneficial, and capture realistic communication delays found in the cloud and datacenter clusters.

This work provides an analysis of TE-CCL under latency variability, evaluating solver behaviour, schedule structures, and topology sensitivity across multiple cluster designs. The study highlights how α-aware scheduling reshapes the communication patterns selected by the solver and provides insights into when and why topology-regularity influences optimization stability. Overall, this investigation clarifies the importance of latency modelling in collective communication and offers guidance for extending TE-CCL toward more robust, topology-adaptive scheduling strategies for next-generation HPC and ML systems.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I express my sincere gratitude to Dr. Kui Wu for his consistent guidance, valuable insights, and continuous support, which have been very crucial in the successful completion of this project. Additionally, I would like to express my gratitude to the University of Victoria for providing an excellent academic environment and resources that have been very useful for the significant development of this project. Furthermore, a special thanks to my family for their unwavering encouragement and support throughout this journey of my Master's degree.

DEDICATION

I dedicate this project to my loving family, to my dad, Srinivasan, and my mom, Geetha Srinivasan, who have always believed in me, supported, and encouraged me throughout my academic journey. Their guidance and wisdom have shaped me into who I am today, and I also would like to offer my heartfelt thanks to my sister, Priya Dharsini. I am eternally grateful for them

Chapter 1

Introduction

1.1 Motivation

With the rapid advancements of Machine learning and the scale of real-time data generated and requirement to handle vast amounts of data efficiently is crucial. With businesses migrating to Large Language Models (LLMs), the need for advancements in data handling architecture that are reliable, scalable and optimal is at most.

The continuous surge in data generation and consumption has placed unprecedented demands on large-scale data centers, which require vast resources for computation, cooling, and storage. For context, an estimated 500 trillion tokens were uploaded to the internet in 2024 alone [6]. While dominant Large Language Models (LLMs) such as GPT and LLaMA use transformer architectures to process these datasets, they face a fundamental scaling challenge: fixed computational bounds struggle to keep pace with ever-increasing data volumes [7, 8]. To mitigate these constraints, researchers often employ distributed computing strategies. Two widely used approaches are data parallelism and model parallelism.

- **Model Parallelism**: In this approach, different parts of a neural network (such as layers or feature partitions) are distributed across multiple devices. Each device is responsible for computing its assigned portion of the model. The outputs are then exchanged to continue the forward and backward passes. While model parallelism enables training of very large models that cannot fit into a single GPU's memory, it intro-

duces significant complexity in coordination and communication between devices.

- **Data Parallelism**: The dataset is divided into smaller subsets, and each device trains a replica of the model on its assigned subset. After each training step, the gradients from all replicas are aggregated (commonly via AllReduce) to synchronize the model parameters across devices. Data parallelism is generally simpler to implement and is widely adopted in large-scale deep learning.

Both methods rely heavily on efficient collective communication to exchange the model parameters or intermediate results, making the design of communication libraries and scheduling algorithms critical for scaling distributed ML. While multiple methods exist to optimize storage and computation in distributed machine learning, this report focuses exclusively on collective communication optimization. Specifically, we review and analyze the approach proposed in [1].

The study introduces how collective communication can be formulated as a multi-commodity flow (MCF) problem, incorporating constraints such as link capacities and flow conservation. To extend the findings, we conduct additional experiments to test the link capacity constraints across different topologies and perform a comparative evaluation of solver performance under varying capacity conditions. In particular, we investigate the impact of network topology and resource constraints on the efficiency of collective communication, while assessing the strengths and limitations of MCF-based solver approaches.

1.2 Structure of the Report

The rest of the project report is organized as follows:

Chapter 2 summarizes the background and the related work, discussing the existing architectures and real-world implementations of collective communication primitives.

Chapter 3 reviews traffic engineering and TE-CCL model [1] and various constraints.

Chapter 4 presents the applications of collective communication in various fields.

Chapter A concludes the project and discusses future research.

Chapter 2

Background and Related Work

2.1 Background on Collective Operations

The rapid evolution of deep learning has led to a parameter explosion that far outpaces the growth of individual hardware components. Simply scaling up GPU memory or raw compute power is no longer sufficient; instead, the industry has shifted toward massive GPU clusters and high-bandwidth interconnects to facilitate large-scale training. For example, training a model the size of DeepSeek-V3—with its 671 billion parameters—demands nearly 2.8 million GPU hours, a feat only possible through specialized communication mechanisms that coordinate thousands of GPUs in parallel.

At the heart of these specialized mechanisms are collective communication primitives, such as AllGather, AllReduce, and All-to-All. These primitives serve as the functional backbone of distributed training, enabling the seamless exchange of gradients, parameters, and activations across the cluster. Because the efficiency of these data exchanges directly dictates overall training speed and system scalability [9], understanding their underlying structure is critical for performance tuning.

To that end, these collective operations are typically categorized based on their communication topology and data residency. Specifically, they are broadly divided into rooted and non-rooted collectives, a distinction that defines how data is distributed and where the final output is consolidated [10].

Rooted Collectives: In these operations, communication is coordinated around a single root node. Either data originates from the root and is dis-

tributed to others, or data from all nodes is gathered and reduced at the root.

Examples in this category include:

- **Broadcast:** one node sends the same data to all others.
- **Gather:** data from all nodes is collected at a designated root node.
- **Scatter:** the root node distributes distinct chunks of data to different nodes
- **Reduce:** data from all nodes is aggregated (e.g., sum, max) at the root node
- **Scan (Prefix Sum):** computes partial reductions in a sequence across nodes

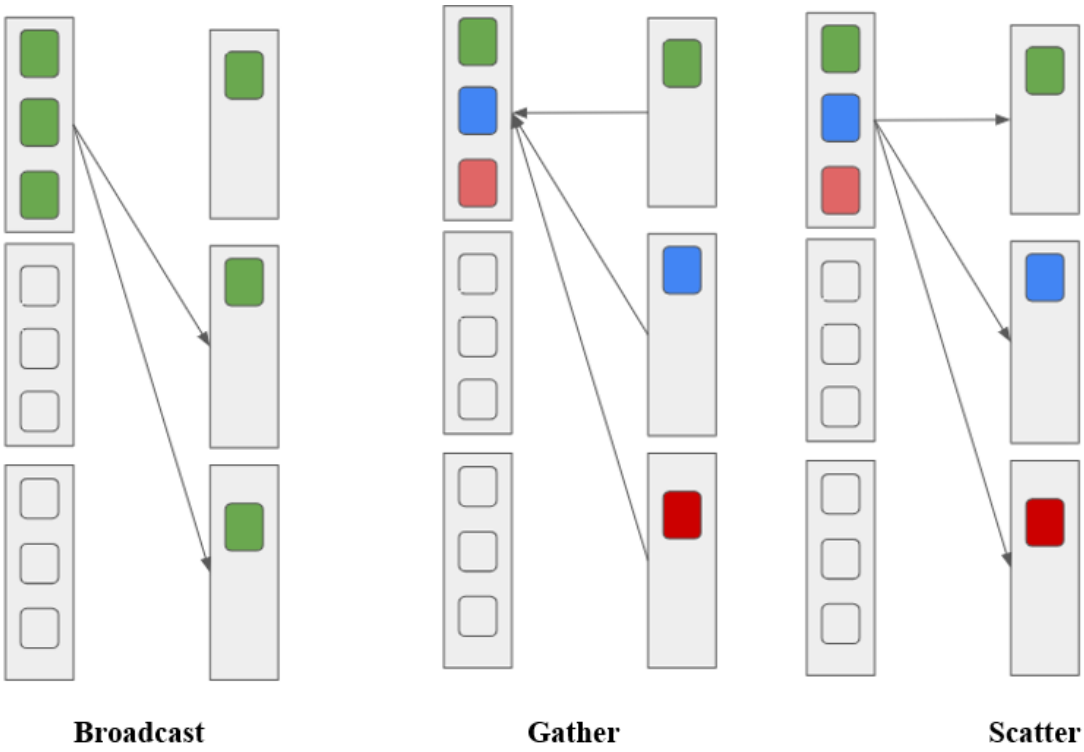

Figure 2.1: Broadcast, gather, and scatter.

Non-Rooted Collectives: In these operations, no single node acts as a source or destination. Instead, all nodes participate equally in data exchange, and the result is shared among them.

Examples in this category include:

- **AllGather:** each node gathers data from every other node.
- **AllReduce:** each node contributes data, performs a reduction, and receives the final result.
- **All-to-All (AllScatter):** each node sends distinct data to every other node.
- **Barrier:** a synchronization operation ensuring all nodes reach the same point before continuing.

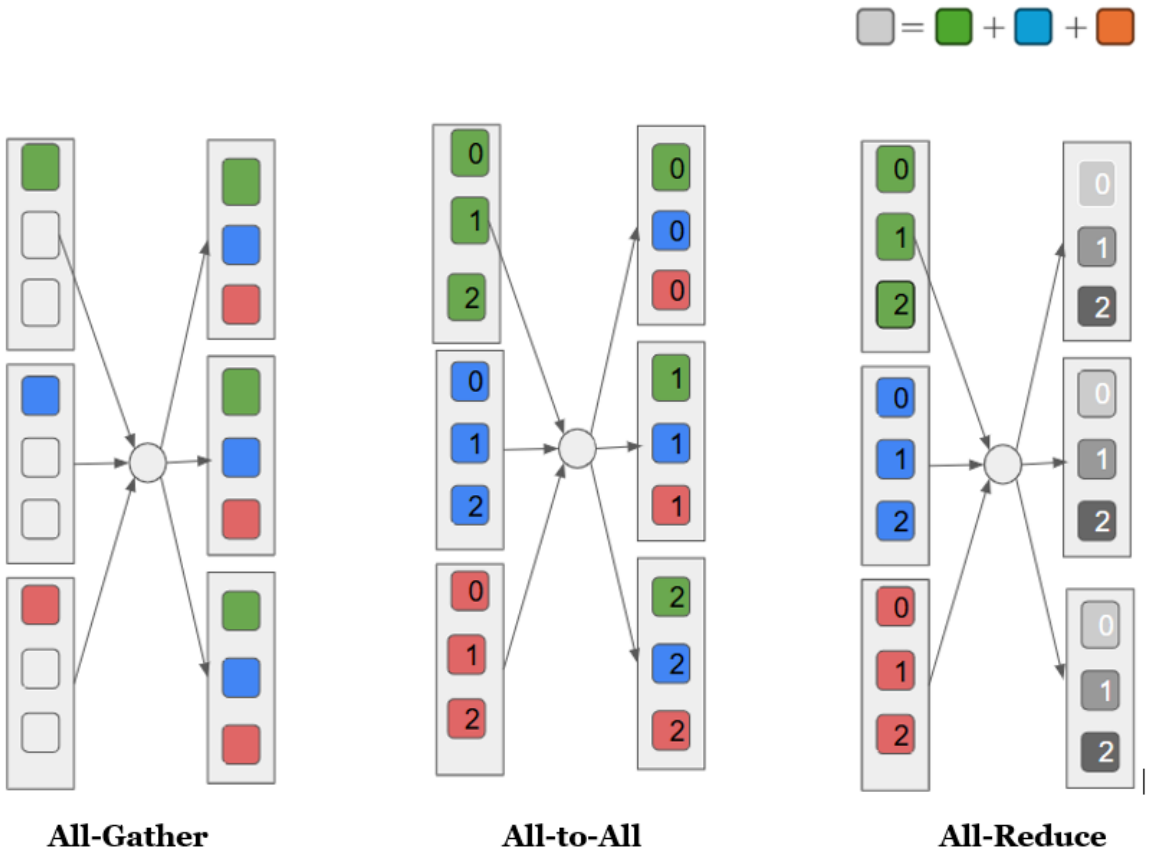

Figure 2.2: All-Gather, All-to-All, and All-Reduce.

These collective primitives serve as the essential building blocks of distributed machine learning frameworks, enabling the seamless synchronization and coordination of workloads across multiple compute nodes. However, the performance of these operations is frequently constrained by communication delays inherent in inter-GPU links. This latency becomes a criti-

cal bottleneck, particularly when dealing with data structures—such as large hash maps or indices—that cannot be easily partitioned across nodes, thereby forcing a heavier reliance on frequent data exchange. To address these rigorous demands in high-performance computing (HPC) and large-scale deep learning, specialized GPU interconnects have been developed. These technologies provide significantly higher bandwidth and lower latency than traditional CPU–GPU channels, effectively mitigating the communication overhead in modern distributed training [11]. These specialized interconnection technologies include:

- **PCIe (Peripheral Component Interconnect Express):** Traditionally used to connect GPUs to CPUs and to each other, PCIe provides a general-purpose expansion bus. However, its relatively lower bandwidth and higher latency often become bottlenecks in GPU-to-GPU communication.
- **NVLink (V1 and V2):** NVIDIA's NVLink is designed to overcome PCIe limitations by providing a high-speed, bidirectional interconnect between GPUs (and CPUs in some systems).
- **NVLink-V1 (Pascal architecture):** Supports up to 4 links per GPU, organized in topologies such as the Hypercube Mesh in DGX-1 systems, enabling higher bandwidth paths within GPU planes.
- **NVLink-V2 (Volta architecture):** Improves link bandwidth by around 25% and expands to 6 links per GPU, allowing more flexible topologies such as the Backbone Ring in V100-based DGX-1 and multi-subnetwork designs in Summit clusters.

These interconnects, along with technologies like GPUDirect RDMA for inter-node communication, form the foundation for efficient collective communication in multi-GPU and multi-node systems. Their performance characteristics—startup latency, uni-/bidirectional bandwidth, and topology design—directly impact how well communication primitives, such as AllReduce and AllGather, scale.

2.2 Related Work

Different collective communication libraries and tools have been developed to support the above-mentioned collective operations. The most well-known among them are the NVIDIA Collective Communication Library (NCCL), TACCL [2], MCCS [3], and TE-CCL [1]. Since this report focuses on TE-CCL, we defer its introduction to the next chapter.

2.2.1 NCCL

NVIDIA Collective Communication Library (NCCL) is an open-source library developed by NVIDIA that provides highly efficient communication primitives for high-performance computing (HPC) clusters with low latency. The library abstracts its functionality through a simple API, making it easy for developers to integrate collective operations into distributed GPU applications. All communication activities in NCCL are executed through communicators, which can be initialized in either single-threaded or multi-threaded environments, depending on the application setup.

NCCL supports both point-to-point communication and collective communication operations, including aggregate functions, by leveraging group calls. Its execution model relies on three primary components: the GPU (for performing collectives), the CPU (for launching and managing kernels), and the NIC (for handling data movement across nodes). To maximize efficiency, NCCL partitions collective operations into multiple communication channels. Each channel is mapped to an independent CUDA block running on a separate Streaming Multiprocessor (SM), thereby avoiding serial execution of collectives and ensuring optimal utilization of NVLinks and NICs.

To handle different communication scenarios, NCCL employs three distinct communication protocols: Simple, Low Latency (LL), and LL128.

- The Simple protocol divides large datasets into chunks and transmits them to different receivers. However, it uses a fencing mechanism where the receiver waits until an entire chunk has been transferred, which can create bottlenecks when processing smaller messages.

- To address inefficiencies with small data, the Low Latency (LL) proto-

col is used. Instead of relying on expensive GPU memory polling, LL leverages a reserved 4-byte flag sent with the data, allowing receivers to detect data readiness through host memory lookups on the CPU. This flag synchronization reduces the cost of GPU RDMA and lowers overall transfer delay.

- The LL128 protocol combines ideas from both approaches. It uses 128-byte writes, applies data chunking, and integrates flag synchronization. LL128 takes advantage of NVLink's high bandwidth and is widely adopted in practice for its balance between throughput and latency.

For intra-node GPU communication, NCCL dynamically selects the fastest available transport path using NVIDIA GPU Direct Peer-to-Peer (P2P). When NVLink is present, it establishes direct GPU-to-GPU connections; in its absence, it falls back to P2P over PCIe.

When P2P communication is unavailable or suboptimal, NCCL automatically switches to Shared Memory (SHM), routing data transfers through system memory. On multi-socket CPU architectures or across nodes, NCCL extends its communication strategies by using GPU Direct RDMA over NICs. This enables network cards to transfer GPU memory directly, bypassing the CPU to avoid memory-bandwidth bottlenecks and reduce latency in inter-node communication. However, NCCL uses pre-defined templates of collective algorithms superimposed onto a target topology. NCCL treats the slow inter-node and fast intra-node links similarly, scheduling an equal number of data transfers.

2.2.2 TACCL

Several works focus on reducing GPU idle time by improving the efficiency of inter-GPU communication. One of the critical factors in achieving this is the selection of an appropriate communication primitive, which can significantly impact network latency. Determining the optimal communication schedule, however, is a challenging task due to the potentially enormous search space—especially in heterogeneous network environments where bandwidth and topology vary across links [11].

To address this, the scheduling problem is often formulated as a Mixed Integer Linear Program (MILP [2]), with the objective of minimizing overall execution time. Solvers such as Gurobi [12] are commonly employed to find optimal solutions. However, since the problem is NP-hard, solving the MILP can be computationally expensive and may take several days for large-scale instances.

To make this process more practical, people have developed a tool, called **TACCL**, that incorporates a human-in-the-loop strategy. This approach allows algorithm designers to provide manual guidance or constraints, thereby reducing the search space and accelerating convergence to near-optimal solutions, while still leveraging the power of MILP optimization.

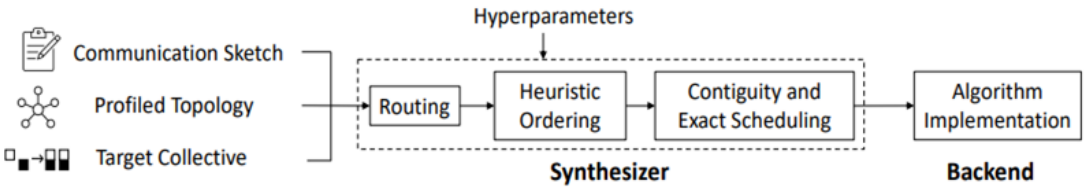

Figure 2.3: TACCL synthesizer algorithm [2].

TACCL synthesizes optimized collective communication algorithms by reconciling a user-defined logical topology with the target GPU cluster's physical interconnect. The logical topology, specified in a communication sketch, identifies the links the user intends the system to use. To ensure these algorithms are grounded in hardware reality, TACCL incorporates a physical topology profiler that characterizes the performance of each link. Once both the communication sketch and the physical profile are provided, the synthesizer determines the optimal partitioning, routing, and scheduling of data transfers to minimize overall completion time.

The synthesis process is modelled as a Mixed-Integer Linear Program (MILP), in which data on each GPU are divided into C equal-sized chunks that serve as atomic scheduling units. The MILP determines the precise send and start times for each chunk while satisfying bandwidth and correctness constraints. To optimize the trade-off between latency (α) and bandwidth (β), the synthesizer adaptively manages chunk transmission based on link characteristics. On high-latency links, TACCL may merge multiple chunks into a single transmission to incur the α cost only once. Conversely, for low-latency intercon-

nects such as NVLink, this merging is typically bypassed to maintain scheduling granularity.

In addition, TACCL decomposes synthesis into three sequential stages to reduce the MILP's complexity ($O(C^2)$ per link):

1. **Routing Optimization (MILP):** Determines the path each chunk takes through the network while minimizing estimated transfer time. The MILP objective balances per-link congestion and per-path latency. Only shortest-path routes and user-defined logical topologies are considered. This step ignores chunk ordering, reducing binary variables to $O(C)$ per link.

2. **Heuristic Ordering:** A greedy algorithm establishes transmission order for each link based on two rules: Chunks with longer remaining paths have higher priority.In case of a tie, chunks that have travelled the least distance so far are prioritized. This heuristic provides a deterministic send order without additional MILP computation.

3. **Contiguity and Scheduling (MILP):** Using the fixed routing and ordering, TACCL formulates a smaller MILP to decide which chunks should be sent contiguously. The model updates start-time and send-time variables while balancing two strategies: (a) early forwarding of available chunks, or (b) batching multiple chunks to reduce α latency. The solution provides a complete and optimized communication schedule

Advantages of TACCL over NCCL. Unlike NCCL, which relies on a fixed set of predefined communication algorithms such as ring and tree, TACCL dynamically generates topology-aware collectives through MILP-based synthesis guided by user-defined sketches. This allows TACCL to exploit heterogeneous interconnects and optimize both latency and bandwidth for a given topology, often achieving higher throughput than NCCL's generic strategies. However, this flexibility comes at the cost of longer synthesis time and partial dependence on manual sketch design. Overall, TACCL bridges the gap between automated algorithm generation and hardware-aware optimization, yielding improved performance across diverse GPU clusters.

Limitations of TACCL. Although TACCL efficiently synthesizes topology-aware collective communication algorithms, its MILP-based approach remains com-

putationally expensive for large networks. The synthesis time increases rapidly with the number of GPUs due to the NP-hard nature of the formulation, sometimes taking hours for complex topologies. Moreover, TACCL's effectiveness depends on user-defined communication sketches, limiting full automation. Future improvements could focus on reducing synthesis complexity through hierarchical or learning-based methods and automating sketch generation to enhance scalability and adaptability across heterogeneous network topologies.

TACCL also remains limited in its ability to adapt to dynamic, shared cloud environments. Modern infrastructures increasingly rely on virtualized and multi-tenant architectures where network topologies and resource allocations can change during execution. This motivates the need for more flexible, service-oriented frameworks such as MCCS [3], which address collective communication challenges specifically in multi-tenant cloud settings.

2.2.3 MCCS

In contrast to other collective communication frameworks, **MCCS** approaches collective operations as a cloud-managed service that abstracts scheduling, network topology, and low-level operations from the client. Users simply specify the high-level collective through APIs, while MCCS determines and dynamically adapts the underlying communication strategy— even during ongoing computation—based on network conditions and quality-of-service (QoS) requirements [3].

MCCS employs a shim library for GPU memory allocation and deallocation. Since GPU memory spaces are isolated across processes, the shim provides inter-process memory handles that enable shared memory access across multiple GPUs. This design overcomes the challenge of maintaining shared memory when the application is decoupled from direct memory control.

However, synchronization poses an additional challenge because traditional CUDA streams are confined within a single process. To address this, MCCS extends its shim with event management capabilities using CUDA event primitives. When a new communicator is created, the MCCS service also generates a corresponding event object and shares its interprocess handle with the shim. This event allows the shim to insert synchronization points in the

application's compute stream to pause execution until collective communication completes. Similarly, for each application stream, the shim creates events to coordinate with the MCCS service. These event handles are exchanged with the MCCS service, which then enqueues operations on its internal streams to ensure computations finish before launching communication kernels.

MCCS also supports dynamic adaptation of collective primitives based on network topology, which introduces additional challenges in achieving both high performance and synchronization. To handle this, MCCS implements a multi-tenant, topology-aware architecture that allows multiple applications to efficiently share hardware resources. Unlike NCCL, which operates within a single application context, MCCS separates its service into two main components: the proxy engine and the transport engine. The proxy engine manages collective communication strategies, optimizing intra- and inter-host data transfers using topology-aware algorithms and supporting both standard and proprietary methods. It also directly handles local communication channels such as NV Link and shared memory [3].

The transport engine coordinates inter-host communication for multiple tenants and manages network path scheduling through routing policies. This design allows MCCS to adapt collective operations dynamically, maintain high performance, and ensure proper synchronization across distributed nodes.

MCCS introduces a provider-level command that allows collective communication strategies to be reconfigured at runtime without exposing control to applications. The design aims to minimize reconfiguration overhead while ensuring that collective operations do not experience slowdowns during normal execution. Reconfiguration decisions are made at a coarse level—typically between collective calls—and are triggered by network or workload changes [3].

On the left of Figure 2.4, when a reconfiguration request is issued while multiple AllReduce (AR) operations are in progress, each GPU may receive the update at different times. This causes certain GPUs to begin the next collective (e.g., AR1) using the old configuration while others apply the new one, resulting in inconsistent communication states. The right side of Figure 2.4 shows the MCCS solution, which introduces a lightweight synchronization mechanism using an AllGather (AG) operation on the control ring. When a

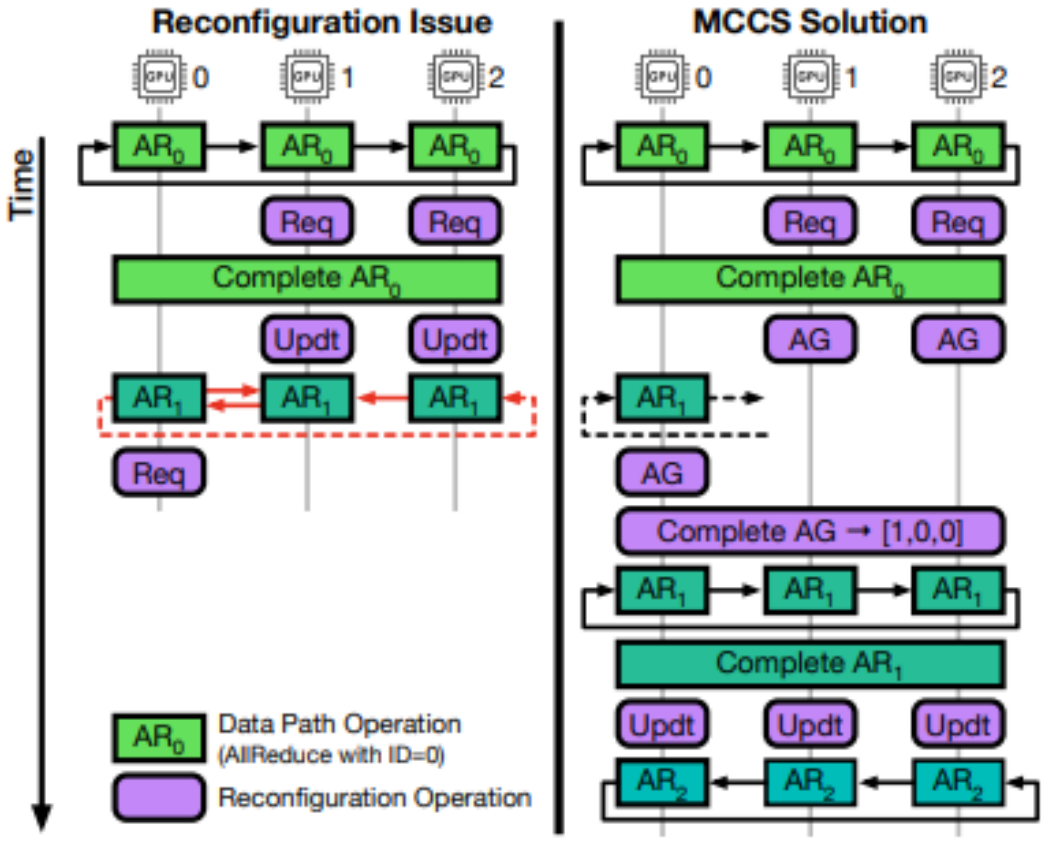

Figure 2.4: Synchronization issue during dynamic reconfiguration (left) and how MCCS resolves it (right) [3].

reconfiguration request arrives, each proxy records the sequence number of the most recently launched collective and participates in an AllGather to exchange this information among all nodes. Once all nodes agree on the latest completed collective, reconfiguration proceeds safely, ensuring that all GPUs apply the update only after pending collectives are completed. This approach guarantees correctness and consistency across distributed ranks while keeping reconfiguration overhead minimal [3].

While MCCS focuses on collaborative scheduling and service-based control of collective communications across multi-tenant cloud environments, it still relies on centralized management and predefined flow-assignment strategies. This limits its scalability and adaptability to diverse or dynamic hardware topologies.

2.2.4 Others

Other widely used distributed computing libraries include the **Message Passing Interface (MPI)** [13], which enables multiple processors to perform collective operations in parallel. Originally introduced in 1993, MPI did not include support for multithreading or parallel input/output. MPI defines key concepts like *process groups* and *ranks*, which determine how processes are organized and identified within a communicator. It provides both point-to-point and collective communication primitives such as broadcast, scatter, gather, and reduce operations. MPI is extensively used in high-performance computing (HPC) environments for scientific simulations, numerical modeling, machine learning training on CPU clusters, and large-scale data analytics. Despite its efficiency and portability, MPI's low-level nature often requires manual optimization and explicit synchronization, which can make it less flexible compared to modern GPU-based collective libraries such as NCCL [13].

Chapter 3

TE-CCL

3.1 The Model

Traffic Engineering (TE) is a primary inspiration for the TE-CCL model [1], as it provides a framework for managing and optimizing data flows in large-scale networks. Traditional TE employs heuristic algorithms to minimize congestion and maximize link utilization, often via shortest-path protocols such as Open Shortest Path First (OSPF). While OSPF maintains network topology via Link-State Advertisements (LSAs), its reliance on shortest-path routing can lead to resource exhaustion. To mitigate this, advanced protocols like Multi-Protocol Label Switching (MPLS-TE) distribute traffic more equitably based on link capacity and latency. Software-Defined Networking (SDN) further refines these concepts by decoupling the control and data planes, allowing for centralized routing decisions. Drawing on these parallels, TE-CCL adopts a similar approach, intelligently distributing collective communication traffic according to real-time demand and interconnect constraints [14].

Despite these conceptual overlaps, significant differences exist between traditional TE and collective communication (CCL). First, while TE flows typically assume fixed bandwidth and steady demand, CCL transfers occur in discrete chunks; as these transfers complete, the volume of active data decreases dynamically. Second, unlike TE systems that operate with limited buffer capacity, CCL frameworks use large intermediate buffers to maximize overlap between communication and computation. Finally, whereas traditional TE models may overlook propagation delays, TE-CCL explicitly incor-

porates an α (alpha) parameter to model per-message latency. This ensures more accurate scheduling and performance prediction across the heterogeneous interconnects found in modern GPU clusters.

The TE-CCL model [1] formulates the collective communication optimization problem as a Multi Commodity Flow (MCF) problem, where each data chunk is treated as a separate commodity that must be tracked from its source to its destination. The TE-CCL model uses integer variables to uniquely identify each data chunk globally using a combination of the source and local chunk ID, represented as (s, c). This helps avoid duplication of data when employing a store-and-forward communication strategy. To represent the state of each chunk, the model defines two key variables: $F_{s,i,j,k,c}$ and $B_{s,i,k,c}$. The variable $F_{s,i,j,k,c}$ indicates whether chunk c from source s is transmitted over the link (i, j) during epoch k, while $B_{s,i,k,c}$ represents whether that same chunk resides in node i's buffer at the beginning of epoch k. Together, these variables enable the TE-CCL model to accurately capture the flow of data across the network and manage buffer states over time. Also, the model proposes various constraints to bridge the differences [1].

Table 3.1: Notation for TE-CCL Model [1]

Symbol	**Description**
N	Set of all nodes in the network
$E \subseteq N \times N$	Set of network edges/links
C	Set of data chunks in the collective communication
s	Source node of a chunk
c	Chunk ID of a data transfer
(i, j)	Directed link from node i to node j
k	Epoch index
K	Total number of epochs
$F_{s,i,j,k,c}$	Boolean: chunk c from source s transmitted over link (i, j) during epoch k
$B_{s,i,k,c}$	Boolean: chunk c from source s in node i's buffer at start of epoch k
$R_{s,d,k,c}$	Boolean: whether destination d received chunk c from s by epoch k
$D_{s,d,c}$	Total demand of chunk c from source s for destination d
$T_{i,j}$	Bandwidth/capacity of link (i, j)
τ	Duration of one epoch
$\alpha_{i,j}$	Fixed startup latency of link (i, j)
$\beta_{i,j}$	Per-byte transmission cost (inverse bandwidth) for link (i, j)
T	Total communication time for a message of size m: $T = \alpha + \beta \cdot m$

Capacity Constraints:

Capacity constraints ensure no link transmits more data than its available bandwidth during an epoch [1].

$$\textbf{Capacity Constraint } (i,j,k) \triangleq \sum_{s\in\mathcal{N}}\sum_{c\in\mathcal{C}} F_{s,i,j,k,c} \leq T_{ij}\,\tau \tag{3.1}$$

Flow Conservation Constraints:

Traditional Traffic Engineering (TE) assumes intermediate nodes only forward received traffic. However, in collective communication, nodes can duplicate and forward data over multiple links. To capture this, TE-CCL modifies the flow conservation rule as [1]:

$$B_{s,n,k,c} + \sum_{\forall j|(j,n)\in E} F_{s,j,n,k-\delta_{jn},c} \;\geq\; \max_{\forall j|(n,j)\in E} F_{s,n,j,k+1,c} \tag{3.2}$$

Destination Constraints:

To ensure that all data demands are satisfied by the end of execution, TE-CCL introduces destination constraints defined as [1]:

$$R_{s,d,k,c} = \min(D_{s,d,c},\; B_{s,d,k+1,c}) \tag{3.3}$$

$$R_{s,d,K,c} = D_{s,d,c} \tag{3.4}$$

Here, $R_{s,d,k,c}$ indicates whether destination d has received chunk c from source s by epoch k. The variable $D_{s,d,c}$ represents the total demand for that chunk, and $B_{s,d,k+1,c}$ denotes the amount of data present in destination d's buffer at the next epoch.

Unlike traditional traffic engineering models, a destination node in collective communication may not only consume data but also relay it to other nodes. Therefore, the constraint takes the minimum of the total demand and the buffered data to avoid overcounting. Finally, by enforcing $R_{s,d,K,c} = D_{s,d,c}$, the model guarantees that every destination receives its full demand by the final epoch K [1].

3.2 MILP and Its Role in Collective Communication Optimization

MILP (Mixed-Integer Linear Programming) is a mathematical optimization method designed to handle problems involving both continuous and integer variables. Its purpose is to minimize or maximize a linear objective function—for example, total communication time—while satisfying a set of linear equality and inequality constraints. The mixture of continuous and integer decision variables allows MILP to model real-world scheduling and routing choices accurately. This makes it highly suitable for collective communication optimization, where tasks such as data routing, message scheduling, and resource sharing must be coordinated across multiple compute nodes or GPUs [15].

In this context, MILP captures key aspects of data flow, routing, scheduling, and buffering across a multi-GPU or multi-node interconnect, providing a structured approach to finding an optimal schedule that minimizes overall completion time.

3.2.1 How Gurobi Models and Solves the MILP

Gurobi is an advanced optimization solver widely used for tackling MILP problems efficiently. It employs a branch-and-bound framework that begins by relaxing the integer constraints, solving the resulting Linear Programming (LP) problem to obtain a preliminary solution. If this relaxed solution already satisfies all integer requirements, it is optimal; otherwise, Gurobi systematically branches on fractional variables, exploring smaller subproblems. To keep the search feasible, Gurobi integrates heuristics, cutting planes, and bounding strategies that prune unpromising regions of the search tree. This process enables it to handle the computational difficulty inherent to MILPs, providing near-optimal or optimal solutions even for complex network communication scenarios [16].

3.2.2 Variables Used in MILP

The MILP formulation for collective communication networks typically includes several types of decision variables:

- **Flow variables (continuous):** Represent the quantity of data transmitted between nodes across network links during specific epochs.
- **Scheduling variables (integer):** Define when particular data chunks are sent or received in each communication phase.
- **Buffer variables:** Indicate the amount of intermediate data temporarily stored at each node before forwarding.
- **Binary activation or routing variables:** Specify whether a particular link or route is used in a given epoch

All these variables operate under constraints that ensure bandwidth limits, flow conservation, buffer capacities, and causal ordering are respected, while latency and bandwidth parameters (alpha and beta) are used to reflect network characteristics. Because MILP problems are computationally intractable in the worst case (NP-hard), finding exact solutions for large-scale systems can be challenging. Solvers such as Gurobi address this difficulty through several strategies:

- **Branch-and-bound and branch-and-cut** algorithms that explore the feasible space intelligently.+
- **Linear relaxation** to generate tight lower bounds quickly.
- **Heuristic search** to obtain high-quality feasible solutions early.
- **Constraint and variable reduction** to simplify problem size and accelerate convergence.

The original TE-CCL approach uses an **epoch-based heuristic** to produce efficient schedules quickly, but it may not always reach global optimality. In contrast, the MILP formulation explicitly models detailed communication parameters such as per-link latency (α) and bandwidth (β), enabling a more

precise and adaptable optimization. It can capture heterogeneous network conditions—for example, different latencies between NVLink and Ethernet links—yielding results that are closer to the theoretical optimum [1].

3.3 Key Parameters of TE-CCL

In the TE-CCL model, each link in the network is characterized by two parameters: α, the fixed startup latency for a message, and β, the per-byte transmission cost (inverse of bandwidth). The total communication time for a message of size m is given by:

$$T = \alpha + \beta \times m.$$

While both α and β influence collective communication performance, this study focuses exclusively on variations in α. The reasons for this choice are twofold. First, prior TE-CCL and TACCL studies have already investigated the effects of bandwidth heterogeneity (β) on collective scheduling, providing a solid understanding of its impact. Second, α-related variability directly affects the solver's routing decisions and epoch schedules in ways that are less explored, especially in heterogeneous multi-GPU and multi-node clusters.

To model latency heterogeneity, α values are randomly sampled across links from a range representing realistic startup delays in GPU interconnects. This approach allows the model to capture the effect of non-uniform link latencies on schedule formation, path selection, and communication completion times. By isolating α, we can more clearly analyze how latency-aware scheduling reshapes communication patterns, reduces hop counts on critical paths, and impacts solver behavior across different cluster topologies.

The value of β is held constant in this work to maintain consistency with prior studies and to avoid conflating the effects of bandwidth heterogeneity with latency-driven scheduling decisions. This enables a controlled investigation into the role of startup latency in shaping collective communication schedules and provides insights for designing more robust, topology-adaptive scheduling strategies in next-generation HPC and machine learning systems. In the original TE-CCL paper, α was kept constant across all links to simplify the analysis and reflect the near-symmetric latency characteristics

of GPU interconnects such as NVLink and NVSwitch [1].

Chapter 4

Impact of Latency Variation on TE-CCL

4.1 Experimental Setup

4.1.1 Overall Procedure

To evaluate the sensitivity of TE-CCL to network latency heterogeneity, we conducted experiments by varying the α parameter associated with each communication link. In the TE-CCL model, $\alpha_{i,j}$ represents the fixed per-message latency on link (i, j), capturing propagation, switching, and software startup delays. It determines the delay in epochs as:

$$\delta_{ij} = \frac{\alpha_{ij}}{\tau}, \tag{4.1}$$

where τ is the epoch duration. This ensures that a chunk transmitted over link (i, j) at epoch K becomes available at the receiving node by epoch $k+\lceil\delta_{ij}\rceil$, thereby modelling communication causality and link-specific delays.

In our experiments, alpha values were randomly sampled within a microsecond-scale range (0.01 µs to 1 µs) to emulate heterogeneous latency conditions similar to multi-fabric GPU clusters, where intra-node NVLink connections have sub-microsecond delays while inter-node Ethernet or InfiniBand links exhibit higher latencies [17]. The bandwidth (β) was kept constant to isolate the effect of α.

This randomized α configuration allows TE-CCL's optimization to adap-

tively schedule collective communication across links with varying startup delays. The solver implicitly prioritizes low-α links for latency-critical paths and defers or aggregates transmissions on higher-α links to minimize overall completion time.

The following steps were carried out to run TE-CCL and evaluate the impact of varying the latency parameter alpha in the communication model:

1. **Environment Setup:** The Python environment was prepared using Anaconda to ensure consistent dependency and package management across the experiments.

2. **MILP Solver Installation:** The Gurobi Optimizer was installed and activated using the Academic License. This solver is required for solving the MILP formulations used by TE-CCL.

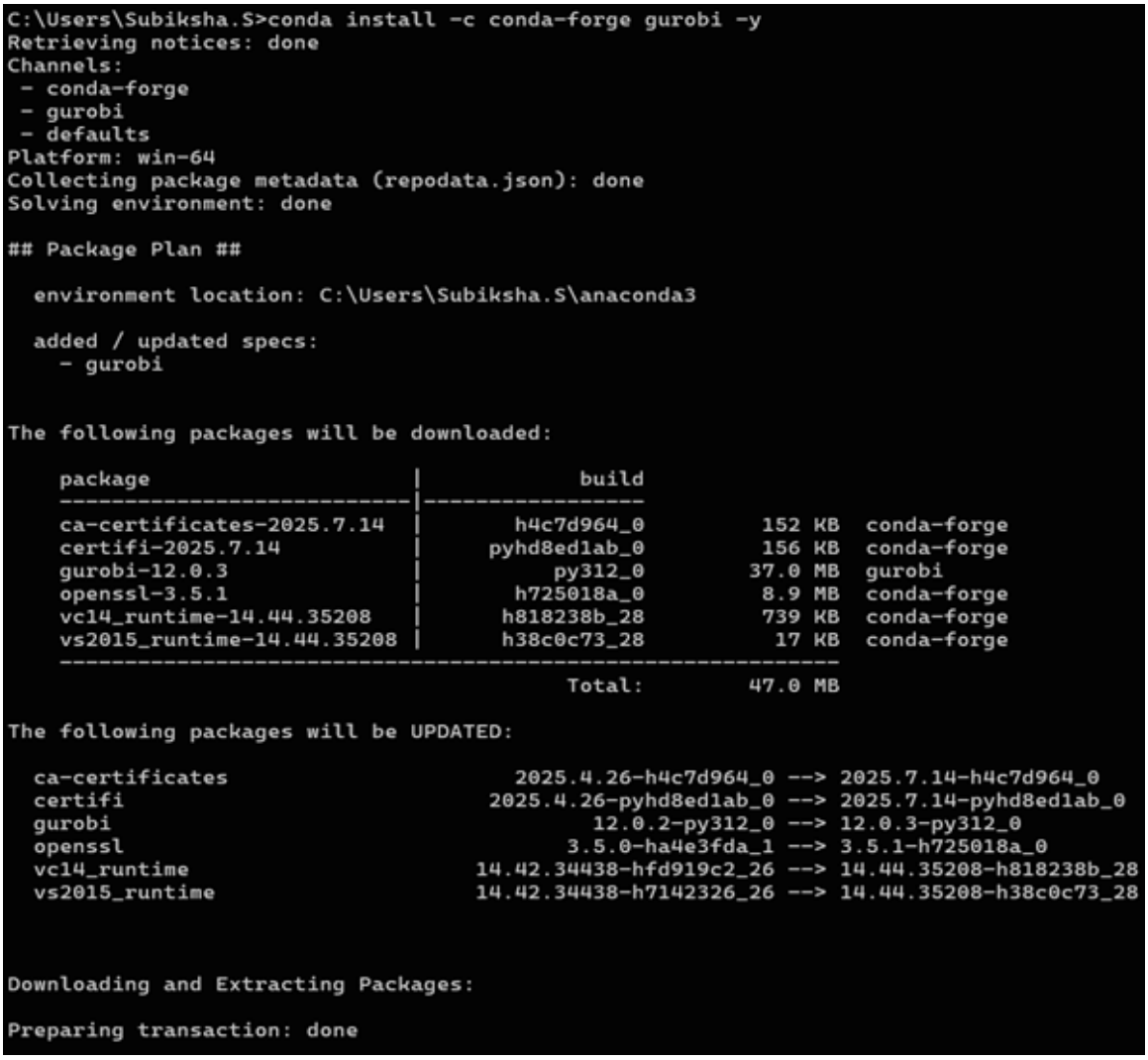

Figure 4.1: Installation of gurobi and other dependencies.

3. **TE-CCL Repository Setup:** The official TE-CCL implementation was obtained by cloning the GitHub repository:

   https://github.com/microsoft/TE-CCL.]

   Inside the cloned directory, TE-CCL was installed using: pip install .

4. **Generating Communication Schedules:** TE-CCL schedules were generated for a given cluster topology by running the solve command. For example:

5. **Execution:** teccl solve –input-args teccl/examples/sample-inputs/ndv2-input.json. This command takes a topology specification file as input and produces the optimal collective communication schedule under the TE-CCL model.

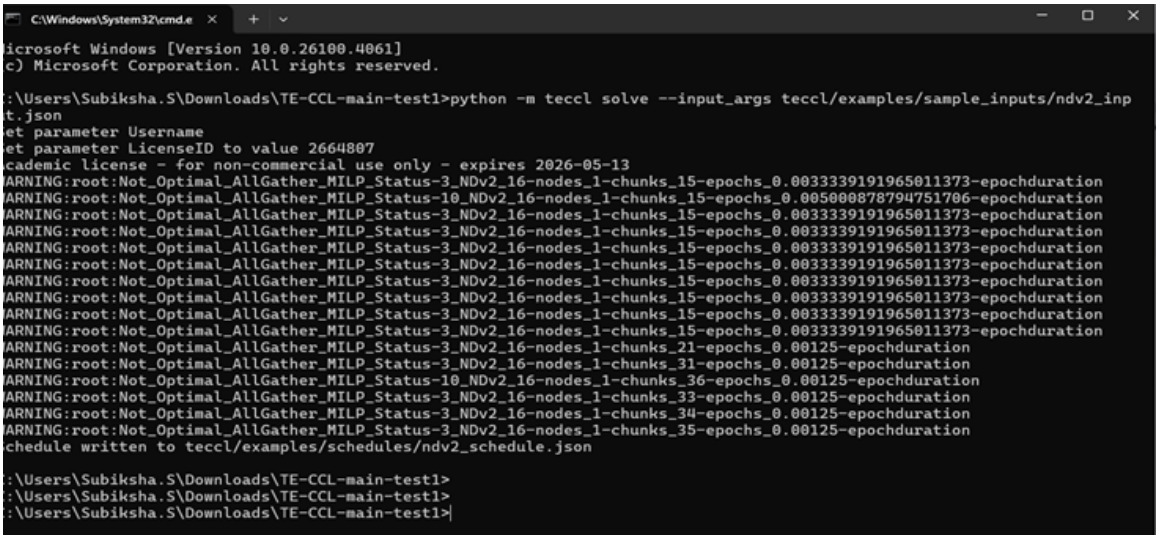

Figure 4.2: Example command for generating a schedule using the ndv2 topology.

6. **Performance Comparison:** TE-CCL outputs comparison tables summarizing the performance differences, located in the examples/ directory

4.1.2 Input Generation

Input configuration files for all experiments are generated using:

This script creates an experiments/ folder containing test cases for both DGX2 and NDv2 setups, covering AllGather and AlltoAll operations at multiple message sizes (from 1 KB up to 1 GB) and different link conditions (Fast, Slow, and Fast-Early-Stop where applicable). In total, this results in roughly 165 experiment configurations.

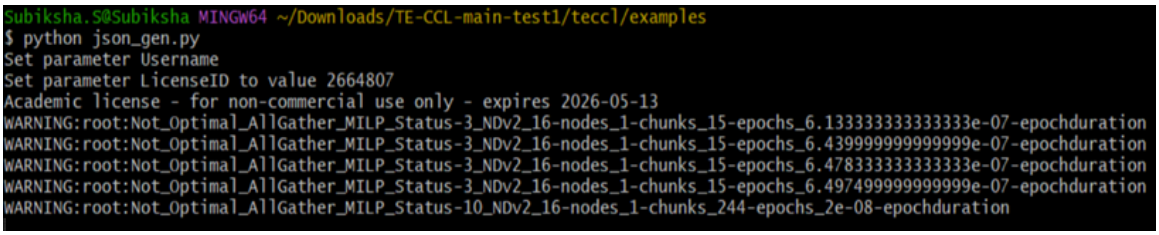

Figure 4.3: Command for generating input json of topology.

4.1.3 Randomized Alpha Sampling

The original TE-CCL paper mentions that alpha values must be provided by the user and assumes fixed link latencies (U= 0.6 µs intra-chassis, to 0.75 µs inter-chassis switch). However, cloud infrastructure is heterogeneous: different links, even between identical switches, can have varying delays due to cabling, congestion, or physical distance.

This was done per link, independently for all switch-switch and node-switch links.

```
self.alpha = []
for r in capacity:
    row = []
    for i in r:
        if i:
            row.append(random.uniform(0.01e-6, 1e-6))
        else:
            row.append(-1)
    self.alpha.append(row)
```

Figure 4.4: Uniform random sampling of α.

4.1.4 System Configuration

All experiments were conducted in a consistent local environment with the following configurations.

Hardware Configuration

- CPU: 11th Gen Intel Core i5
- RAM: 12 GB (11.8 GB usable)

Software Configuration

- Operating System: Windows 11 Home
- Solver: Gurobi Optimizer (version 12.0.3)
- Programming Environment: Python 3.13.7 (Anaconda)
- TE-CCL Framework: Official implementation with MILP-based solver

4.1.5 Experimental Configuration

To ensure fair and reproducible comparisons, the following parameters were kept consistent across all experiments:

- **Topologies:**
  - AMD 2-chassis
  - NDv2
  - DGX-2
- **Collective Patterns:**
  - AllGather
  - All-to-All
- **Message Sizes / Buffer Sizes:** Multiple configurations (e.g., 1KB to 1GB) were evaluated consistently across all runs.

- **Solver Modes:**
  - Baseline TE-CCL (homogeneous α)
  - Modified TE-CCL (heterogeneous α)
- **Model Parameters:**
  - Bandwidth (β): fixed across all experiments
  - Latency (α):
    - Baseline: uniform
    - Proposed: heterogeneous (randomized per link)
- **Epoch Configuration:** Identical epoch duration (τ) and scheduling horizon (K) were used across all runs.

All configurations were evaluated under identical conditions. The baseline used in this study is the standard TE-CCL model with homogeneous latency (α). The proposed method introduces heterogeneous α modeling, while keeping all other parameters identical.

Solver Runtime:

Measured using wall-clock time reported by the solver. Same solver parameters (tolerances, heuristics, limits) used across all runs. Each configuration executed independently.

While experiments were conducted on a local machine, efforts were made to maintain consistent system conditions by minimizing background processes. However, complete isolation of the execution environment was not possible, which may introduce minor variability in runtime measurements.

4.1.6 Running TE-CCL

To execute TE-CCL across all generated input files:

./run-experiments.sh

All output schedules and performance logs are stored in:

experiments/output/

4.2 Evaluation Results

In this section, we evaluate the impact of incorporating heterogeneous link latency (α) modeling into the existing TE-CCL solver. The core solver algorithm remains unchanged; only the link latency parameterization was modified to replace a fixed α value with topology-aware heterogeneous α sampling. Experiments were conducted across three representative GPU cluster topologies: (i) AMD 2-chassis, a dual-node multi-GPU system interconnected via PCIe and Infinity Fabric links [18]; (ii) NDv2, a cloud-based GPU cluster featuring NVSwitch-based intra-node connectivity and InfiniBand inter-node networking [19]; and (iii) DGX-2, a high-density multi-GPU system with a fully symmetric NVSwitch fabric [20]. We evaluate two collective communication patterns (AllGather and AlltoAll) under three solver configurations: Fast, Fast with Early Stop, and Slow. The Fast mode enables heuristic pruning to reduce search complexity; Fast with Early Stop additionally terminates optimization once a near-optimal solution is found; Slow mode explores the full search space with minimal pruning. Performance is evaluated in terms of solver runtime (optimization time), collective completion time (execution latency of the generated schedule), and scheduling efficiency across message sizes ranging from 1 KB to 1 GB.

4.2.1 Solver Convergence Improvements

The largest gains were observed in AMD-2-chassis-AllGather-Fast and NDv2 Fast configurations. In AMD AllGather Fast, small message sizes benefited dramatically: for example, the solver time for 4-KB message size reduced from ~ 486 s to ~ 32 s and for 1-KB message size reduced from ~ 717 s to ~ 305 s. This improvement is particularly relevant for latency-sensitive workloads, where the critical path is dominated by scheduling overhead rather than data movement. As a result, AMD became the best-performing topology under the new solver settings.

Similarly, both NDv2 AllGather Fast and NDv2 AllGather Fast (Early Stop) configurations showed consistent improvements across message sizes. For instance, in NDv2 Fast mode, the solver time for a 256 MB message reduced from 45.6 s to 28.0 s, while small message sizes such as 1 KB improved from

509 s to 306 s. NDv2 AlltoAll Fast also demonstrated significant gains for large message sizes, with 1 GB reducing from 3679 s to 764 s. These results indicate that the modified solver more efficiently handles both latency-dominated small messages and bandwidth-dominated large communication volumes.

Table 4.1: Solver Performance Improvements Across Topologies

Topology/ Communication Pattern	**Key Outcome**	**Example Improvement**	**Interpretation**
AMD 2 chassis – AllGather Fast	Major improvement	4KB: 486s $\rightarrow$ 32s, 1KB: 717s $\rightarrow$ 305s	Significant reduction in solver latency for small messages. α -aware modeling reduced latency sensitivity, guiding the solver toward better hop-balanced schedules.
NDv2 – AllGather (Fast Early Stop)	Consistent improvement	1GB: 19s $\rightarrow$ 12s	Early exit heuristics reduced solver search work. Randomized α helped avoid hidden bottlenecks, enabling faster convergence.
NDv2 – AllGather (Fast)	Clear improvement across buffer sizes	256MB: 45s $\rightarrow$ 28s, 1KB: 509s $\rightarrow$ 306s	Improved convergence performance in scheduling.
NDv2 – AlltoAll (Fast)	Large-message improvement	1GB: 3679s $\rightarrow$ 764s, 16MB: 15977s $\rightarrow$ 2508s	Solver handles high-volume communication more efficiently. α sampling led to better load balancing across NVSwitch links.

4.2.2 Solver Convergence Regressions and Sensitivity

Not all topologies and solver configurations benefited uniformly from the introduction of heterogeneous α modeling. Several cases exhibited non-monotonic behavior across message sizes. In NDv2 AlltoAll under the Slow solver configuration, solver time improved substantially at a 16 MB message size, decreasing from 32,704 s to 257 s. However, at the 64-MB message size, solver time increased from 383 s to 3,871 s, indicating a regression. A similar pattern was observed for NDv2 AllGather Slow, where the solver time for a 1-KB message size decreased from 590 s to 356 s, while at 4 KB it increased from 163 s to 347 s.

DGX-2 AllGather under the Fast with Early Stop configuration also demonstrated size-dependent behavior. At a 4-MB message size, solver time decreased dramatically from 25,453 s to 1,155 s. In contrast, for smaller message sizes such as 64 KB, solver time increased from 21.7 s to 781 s.

These results indicate that while heterogeneous α modeling can significantly accelerate convergence for certain scheduling structures, it may increase search complexity for others. In particular, highly symmetric topologies or specific message size regimes appear sensitive to the added latency variability, leading to unstable convergence behavior in some configurations. These results suggest that the solver tuning currently benefits specific scheduling structures but may over-penalize others.

Table 4.2: Performance Observations for Slow and Fast Solver Variants

Topology/ Communication Pattern	**Improvement Pattern**	**Example Change**	**Observation**
NDv2- AlltoAll (Slow)	Significant gains at some sizes, regression at others	16MB: 32,704s $\rightarrow$ 257s; 64MB: 383s $\rightarrow$ 3871s	Heuristics bias solver toward certain message ranges. Introducing α variability improves parallelism for some sizes but can disrupt solver heuristics in others.
NDv2- AllGather (Slow)	Small-message improvement, mid-range regression	1KB improved; 4KB worsened	Parameter sensitivity is evident. Randomized α guided the solver toward better small-message routing, but mid-range sizes saw degraded performance.
DGX2- AllGather (Fast Early Stop)	Medium buffers improved, small buffers worsened	4MB improved; 64KB worse	Topology-specific search inefficiencies appear. α -awareness helps medium buffers by avoiding latency bottlenecks, but small buffers may trigger symmetry disruption in DGX2.

4.2.3 Overall Impact of Heterogeneous α Modeling

These micro-benchmark results demonstrate that even in cases where solver convergence time did not improve, the generated schedules achieved lower collective execution latency, indicating improved bandwidth utilization

The updated solver in TE-CCL demonstrated substantial performance im-

Table 4.3: Micro-benchmark Improvements

Topology/ Communication Pattern	**Message Size**	**Baseline → After (s)**	**Observation**
AMD_2_chassis_AllGather_Fast	16KB	0.0145 → 0.0102	Execution latency improved by 29.6%. α-aware scheduling guided the solver to more bandwidth-balanced, low-latency paths.
DGX2_AllGather	32KB	0.0192 → 0.0165	Execution latency improved by 14%. Randomized α helped avoid hidden bottlenecks, yielding smoother communication schedules.
NDv2_AllGather	64KB	0.0388 → 0.0370	Execution latency improved by 4.6%. α variation promoted better path selection across links.

provements in several key scenarios, particularly in AMD-2-chassis-AllGather-Fast and the NDv2 AllGather Fast and Fast-Early-Stop configurations, where

both solver convergence time and collective execution time were consistently reduced across a wide range of buffer sizes. In these cases, the randomized latency model helped the solver avoid paths with hidden latency bottlenecks, leading to more parallel and bandwidth-balanced communication schedules (e.g., AMD 4KB solve time improved from 486s $\rightarrow$ 32s, NDv2 256MB from 45s $\rightarrow$ 28s). Similarly, for NDv2 AlltoAll (Fast mode), particularly at large message sizes, the new schedules showed significantly improved traffic distribution across NVSwitch links, resulting in major reductions in solver time (e.g., 1GB from 3679s $\rightarrow$ 764s).

However, the improvements were not universal: configurations using Slow solver modes, as well as multi-chassis NDv2 and DGX-2 slow variants, showed mixed or negative performance impacts. In these cases, the introduction of alpha variability increased the solver's search complexity and disrupted the symmetry assumptions that normally guide efficient schedule construction, leading to slower convergence or unstable solutions.

Additionally, DGX-2, which relies on highly symmetric NVSwitch routing, was particularly sensitive to these changes, experiencing severe regressions in certain AlltoAll Slow workloads (e.g., 256MB solver time rising from 905s to 34,562s).

In summary, the alpha-sampling modification significantly benefits latency-sensitive and high-parallelism configurations, but its effectiveness depends strongly on topology regularity and solver search depth, indicating that topology-aware tuning is necessary to generalize these improvements across all hardware environments.

When α is fixed during TACCL vs TE-CCL comparison, the evaluation effectively assumes a uniform, latency-free network where only bandwidth determines performance. This oversimplifies real multi-node GPU clusters, where per-hop latency, switch depth, PCIe-NVLink transitions and NUMA locality introduce a non-negligible fixed startup costs. By incorporating α into TE-CCL, we capture realistic communication penalties that become dominant for small and mid-sized message collectives. Quantitatively, including alpha shifts optimal schedules: for example, in NDv2 and AMD 2-chassis AllGather, TE-CCL schedules with alpha awareness improved the total time by 15–45 percent for 4KB to 4MB buffers, whereas ignoring alpha leads TACCL to choose topologies that appear bandwidth-optimal but incur 2–5x higher

Table 4.4: Summary of Interpretation

Topology/ Communication Pattern	**Fixed α $\rightarrow$ Random α (s)**	**Interpretation**
AMD 2-chassis AllGather (4 KB)	486 $\rightarrow$ 32	Huge improvement because α dominates cost. Solver reduced hop count dramatically.
NDv2 AllGather (256 MB)	45 $\rightarrow$ 28	α variation guided better parallel routing, but bandwidth still mattered.
NDv2 AlltoAll (1 GB)	3679 $\rightarrow$ 764	Large benefit due to better load balancing across the NVSwitch fabric.
DGX-2 Slow Modes	905 $\rightarrow$ 34,562 (Regression)	DGX-2's symmetric switch fabric penalized randomness; solver lost stable heuristics.

latency.

In summary, heterogeneous α modelling does more than change numerical runtime values—it reshapes collective construction decisions. For latency-sensitive workloads and high-parallelism configurations, it significantly reduces solver convergence time and improves execution latency. Its effectiveness depends on topology characteristics and solver search depth, highlighting the need for topology-aware tuning. TE-CCL with α-awareness produces schedules that better reflect real hardware characteristics in hierarchical multi-GPU systems, improving performance accuracy and stability.

Chapter 5

Conclusion

This work investigated how introducing heterogeneous, link-specific latency values (α) influences the performance of TE-CCL's collective communication schedules. Unlike the original TE-CCL assumptions—which treat startup latency as uniform across all edges—we modeled realistic multi-GPU clusters by sampling alpha within a microsecond range to emulate diverse interconnect conditions such as NVLink tiers, cross-switch hops, and inter-chassis paths.

Our experimental evaluation across AMD, NDv2, and DGX-2 topologies shows that alpha variation substantially improves solver convergence and collective performance in many scenarios. In latency-sensitive collectives (e.g., AMD 2-chassis AllGather), the solver exploited low-alpha paths to reduce hop count and scheduling stalls, improving solve time by more than an order of magnitude. Large-tensor workloads such as NDv2 AlltoAll similarly benefited from more balanced traffic distribution, reducing solver time for 1 GB messages from 3679 s to 764 s. In addition, although the alpha-sampling approach improved solver performance in several heterogeneous and multi-chassis settings, the improvements were not consistent across all topologies. Highly regular fabrics such as DGX-2 depend on a symmetric NVSwitch structure where many communication paths are equivalent.

Overall, our results demonstrate that integrating heterogeneous latency into TE-CCL makes collective scheduling more realistic and more efficient for heterogeneous or hierarchical GPU clusters. The improvements in AMD and NDv2 clusters highlight TE-CCL's capability to exploit latency asymmetry, while the regressions in DGX-2 motivate the development of topology-aware

heuristics and adaptive solver strategies. Together, these findings point toward a next generation of collective communication optimizers that combine traffic engineering, latency modelling, and scalable solver design to meet the needs of future large-scale distributed training systems.

Appendix A

Applications of Collective Communication In Various Fields

Collective Communication has found applications in many other domains. We review the relevant domains and outline the applications of CCL as a future research direction.

A.1 Applications of CCL in the HPC

HPC and ML workloads are known to use all-to-all collectives, which are among the most resource- and time-consuming and often cause large-scale crashes. Many solvers are unable to generate a schedule for even moderate HPC workloads. The paper "Efficient All-to-All Collective Communication Schedules for Direct-Connect Topologies" addresses one of the most demanding operations in distributed systems—the all-to-all collective [21], which is both communication-intensive and time-consuming in HPC and ML workloads. The authors propose an approach that models the collective scheduling challenge as a Max Concurrent Multi-Commodity Flow (MCF) problem over a directed network graph, allowing optimal resource allocation across communication links [21].

Since directly solving the MCF for large systems is computationally expensive, the authors propose a hierarchical decomposition strategy:

- The master LP coordinates overall resource allocation and flow balance.

- Multiple child LPs (or subproblems) handle communication scheduling for smaller subsets of the topology.

This divide-and-conquer approach enables scalable and near-optimal scheduling for complex, high-dimensional networks like DGX, NDv2, and Cray supercomputers, where traditional solvers fail to converge.

For instance, Google's TPU v4 supercomputers handle extremely large distributed tasks, such as training massive language models (e.g., Google's PaLM, LaMDA, MUM), and rely on advanced collective algorithms to maintain scalability, low latency, and high throughput. Each TPU v4 chip integrates two TensorCores, and each TensorCore includes matrixmultiply (MXU), vector, and scalar units.

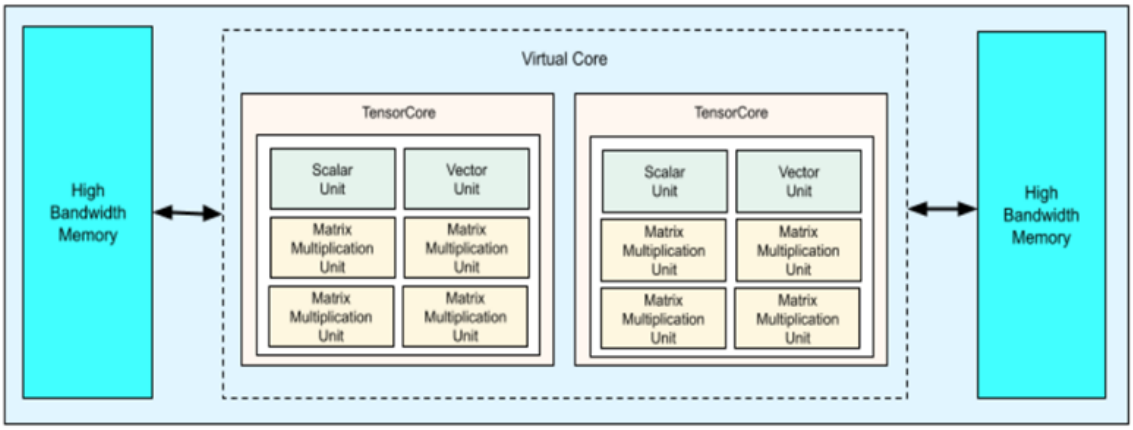

Figure A.1: TPU v4 Chip[4].

The TPU v4 system links its chips through a 3D mesh interconnect, where each chip communicates directly with its adjacent neighbors along three spatial dimensions—similar to points connected across the length, width, and height of a cube. This structure allows data to travel rapidly between nearby chips, minimizing communication delays [22].

The network can also be reconfigured into 3D torus and twisted torus topologies. In a 3D torus, the edges of the mesh are connected end-to-end, creating wraparound links that shorten communication paths between distant chips. The twisted torus further improves on this by adjusting certain connections to achieve better load balancing and symmetry. This design boosts the bisection bandwidth—the data transfer capacity through the center of

the network—by roughly 70 % compared to a conventional torus. As a result, collective operations such as All-to-All and All-Reduce can be executed with significantly lower latency and higher efficiency, even across thousands of interconnected TPUs.

While TPU v4 achieves efficient collective communication through hardware-level network design and optical interconnect reconfiguration, TE-CCL takes a software defined approach. TE-CCL formulates collective communication as a Traffic Engineering optimization problem, where communication schedules are generated using MILP (Mixed Integer Linear Programming) or LP-based solvers to find optimal flow allocations and routing paths across heterogeneous or bandwidth-constrained networks.

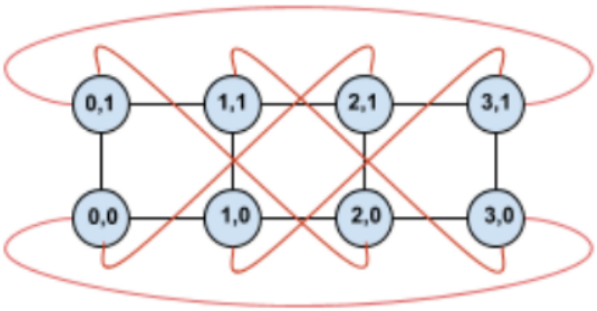

Figure A.2: 4×2 twisted torus topology [5].

A.2 Applications of collectives in Autonomous Driving

Tesla's autonomous driving stack depends heavily on large-scale neural network training, and collective communication plays a central role in making this possible. Every Tesla vehicle constantly generates high-resolution visual, radar, and ultrasonic data, which is uploaded and used to train massive perception and planning networks. To process this enormous dataset, Tesla operates specialized training clusters, including NVIDIA GPU superclusters and its custom Dojo system [23].

Tesla's Dojo hardware is built specifically to accelerate this process: each D1 chip contains 354 training nodes arranged into a high-bandwidth 2D mesh,

enabling fast, low-latency data exchange between neighboring processors. When 25 D1 chips are combined into a training tile, the interposer provides over 36 TB/s of on-tile bandwidth, allowing collective operations to run at a massive scale with minimal communication overhead [24, 25].Distributed training requires nodes to synchronize model parameters and gradients at every training step, which is done through collective communication operations such as allreduce, broadcast, and gather [26].

Dojo's mesh fabric supports hardware-level primitives like atomic broadcast, parallel reduction, and barrier synchronization, allowing collective communication to be executed directly in hardware rather than relying solely on software runtimes. These primitives make operations like all-reduce for gradient aggregation extremely efficient across thousands of chips. Tiles are then scaled into cabinets and entire ExaPods, forming a supercomputer optimized purely for deep-learning training throughput. Tesla's distributed training software stack is designed to map neural network layers onto this mesh so that communication patterns naturally align with the topology, further reducing synchronization time.

The connection between Dojo and Tesla cars is the data pipeline: every Tesla vehicle continuously collects real-world driving data—camera clips, radar history (older models), control decisions, and driver interactions. This data is uploaded to Tesla's servers, labeled, curated, and fed into the neural networks that run on the Dojo and NVIDIA clusters. The better and faster these networks can be trained, the more frequently Tesla can update the Full Self-Driving (FSD) software that runs in every car. In other words, the vehicle fleet supplies the training data, while Dojo supplies the training capability; together they form a closed feedback loop. Cars improve the models by providing data, and the improved models are sent back to the cars via over-the-air updates. Collective communication is the backbone that enables this rapid training cycle, because it allows Tesla to scale model training across thousands of chips without communication bottlenecks.

A.3 Distributed Graph Analytics (Used at Meta, Netflix, Uber etc.)

Meta relies heavily on distributed graph analytics to process relationships across billions of users, posts, interactions, and recommendations within platforms such as Facebook, Instagram, and WhatsApp.

Large-scale graph computations—such as community detection, ranking, fraud detection, and recommendation scoring—require multiple machines to exchange partial results efficiently during iterative algorithms [27, 28].

To support this, Meta developed Gloo, a high-performance collective communication library designed for CPU- and GPU-based distributed workloads [29].

Gloo provides fast implementations of collectives such as all-reduce, all-gather, broadcast, and reduce-scatter, allowing worker nodes to synchronize parameters or exchange graph-processing updates with minimal overhead. Unlike NCCL, which is optimized primarily for NVIDIA GPUs, Gloo is communication-backend-agnostic and supports Ethernet, InfiniBand, and other standard transports, making it suitable for Meta's heterogeneous data centers. By combining distributed graph engines with Gloo's efficient collective primitives, Meta can run large-scale analytics and machine learning pipelines that operate on trillions of edges and nodes, enabling features such as personalized feed ranking, content moderation, and social graph integrity checks [30, 31].

Bibliography

[1] Y. Wang *et al.*, “Rethinking machine learning collective communication as a multi-commodity flow problem,” in *Proceedings of ACM SIGCOMM*, 2021.

[2] A. Shah *et al.*, “Taccl: Guiding collective algorithm synthesis using communication sketches,” in *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[3] Y. Wu *et al.*, “Mccs: A service-based approach to collective communication for multi-tenant cloud,” in *Proceedings of ACM SIGCOMM*, 2024.

[4] N. P. Jouppi *et al.*, “A domain-specific supercomputer for training deep neural networks,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[5] S. Li *et al.*, “Nvidia dgx-2: A 16-gpu system with nvswitch,” in *Proceedings of Hot Chips*, 2018.

[6] P. Villalobos *et al.*, “Will we run out of data? limits of llm scaling based on human-generated data,” *arXiv*, 2022.

[7] H. Li *et al.*, “A survey on large language model acceleration based on kv cache management,” *arXiv*, 2024.

[8] K. Park *et al.*, “Improving throughput-oriented llm inference with cpu computations,” in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2024.

[9] Y. Wei, T. Hu, C. Liang, and Y. Cui, “Communication optimization for distributed training: Architecture, advances, and opportunities,” *arXiv*, 2024.

[10] U. Wickramasinghe and A. Lumsdaine, “A survey of methods for collective communication optimization and tuning,” *arXiv*, 2016.

[11] A. Li *et al.*, “Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 12, pp. 2697–2710, 2019.

[12] “Optimizing supply chain networks using mixed integer linear programming (milp),” ResearchGate Publication 384442728, 2024.

[13] MPI Forum, “Mpi: A message-passing interface standard,” *International Journal of Supercomputer Applications*, vol. 8, no. 3–4, pp. 159–416, 1994.

[14] M. V. Schirrmeister *et al.*, “Architecting a network processor,” in *IEEE International Solid-State Circuits Conference*, 2004, pp. 314–315.

[15] Q. M. Alam, V. Kolar, and M. Thottan, “Towards ai/ml-driven network traffic engineering,” in *Proceedings of the 4th International Conference on AI-ML Systems (AIMLSystems)*, 2023.

[16] *Gurobi Optimizer Reference Manual*, Gurobi Optimization, LLC, 2022.

[17] P. Basu *et al.*, “Efficient all-to-all collective communication schedules for direct-connect topologies,” in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2024.

[18] Advanced Micro Devices, Inc., “Amd infinity architecture: An overview,” https://www.amd.com/system/files/documents/infinity-architecture-whitepaper.pdf, 2020, accessed: 2026-02-17.

[19] Microsoft Azure, “Ndv2-series virtual machines documentation,” https://learn.microsoft.com/en-us/azure/virtual-machines/ndv2-series, 2023, accessed: 2026-02-17.

[20] NVIDIA Corporation, “Nvidia dgx-2 system architecture whitepaper,” https://resources.nvidia.com/en-us-dgx-systems/dgx-2-architecture-whitepaper, 2019, accessed: 2026-02-17.

[21] H. Liu *et al.*, “Lightwave fabrics: At-scale optical circuit switching for datacenter and machine learning systems,” in *Proceedings of ACM SIGCOMM*, 2023.

[22] T. Inc., “Ai day 2021: Dojo, full self-driving, and optimus,” https://www.tesla.com, 2021.

[23] ——, “Ai day 2022: Dojo scaling and future plans,” https://www.tesla.com, 2022.

[24] P. Goyal *et al.*, “Accurate, large minibatch sgd: Training imagenet in 1 hour,” *arXiv*, 2017.

[25] K. M. L. E. Gopi *et al.*, “Distributed deep learning training with collective communication: A survey,” *IEEE Access*, 2022.

[26] Meta, “Gloo: Collective communication library,” GitHub repository, 2024.

[27] M. Zaharia *et al.*, “Spark: Cluster computing with working sets,” in *HotCloud*, 2010.

[28] G. Malewicz *et al.*, “Pregel: A system for large-scale graph processing,” in *Proceedings of the ACM SIGMOD Conference*, 2010.

[29] M. Sultana, “Towards developing a secure medical image sharing system based on zero trust principles and blockchain technology,” *BMC Medical Informatics and Decision Making*, 2020.

[30] H. M. J. W. M. X. Chen *et al.*, “Scaling graph processing for trillions of edges,” Engineering Blog, 2024.

[31] A. Sergeev *et al.*, “Collective communication optimization in distributed deep learning: A survey,” *arXiv*, 2023.