Accelerating Collective Communication for Distributed Machine Learning

Liangyu Zhao

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2026

Reading Committee:

Arvind Krishnamurthy, Chair

Ratul Mahajan

Baris Kasikci

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

Accelerating Collective Communication for Distributed Machine Learning

Liangyu Zhao

Chair of the Supervisory Committee:
Arvind Krishnamurthy
Paul G. Allen School of Computer Science & Engineering

Collective communication has emerged as a cornerstone of distributed machine learning, enabling datacenter-scale clusters of accelerators to collaboratively train or serve large language models. However, it has also become a significant performance bottleneck, impeding the efficient utilization and scalability of hardware resources. This dissertation focuses on optimizing collective communication for machine learning hardware and workloads, approaching the challenge from the perspectives of network topology, communication scheduling, and parallelization strategies.

We first present our work on co-optimizing network topology and communication scheduling for direct-connect optical circuit networks. We propose expansion techniques and a linear programming-based schedule generation algorithm to synthesize efficient large-scale topologies and schedules, thereby forming a Pareto frontier of the latency-throughput trade-off. Our approach enables efficient collective communication on low-diameter topologies.

Then, we introduce ForestColl, a schedule generation algorithm capable of producing throughput-optimal schedules for any network topology in polynomial time. ForestColl leverages prior graph-theoretical results to construct spanning trees for collective communications. It is the first work to achieve throughput optimality for collective communications while delivering orders-of-magnitude speedups in schedule generation compared to prior approaches.

Finally, we outline our future work on automating the search for parallelization and optimization strategies in machine learning training. We propose a strategy grounded in the sharding and processing states of tensors within the compiled computation graph. By adopting a unified view of all tensor types, we propose a method that can discover optimal parallelization and optimization strategies through the determination of abstract tensor states.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Collective Communication in Distributed Machine Learning

Originally developed in the context of high-performance computing (HPC), collective communication has long been a cornerstone of parallel computing, enabling efficient coordination and data exchange among distributed processing units. With the rapid advancement of machine learning (ML) [66, 132, 144], particularly the rise of large language models (LLMs) [117, 20, 93, 82], deep neural networks (DNNs) have grown dramatically in both parameter scale and data consumption. To mitigate accelerator memory pressure and speedup computation, collective communication has become a critical component in distributed DNN training and inference, facilitating the exchange of model states—such as parameters, activations, and gradients—across tens of thousands of accelerators or more [120, 131, 99, 75, 161].

While collective communication enables large-scale DNN training and inference, it has also emerged as a major performance bottleneck. Accelerators' compute cores often remain idle while waiting for collective operations to synchronize results from previous steps and fetch data for subsequent computations, leading to substantial underutilization of hardware resources and reduced model FLOPs utilization (MFU) [151, 163, 113, 131, 152, 60, 43, 115]. Depending on the workload characteristics, collective communication can be either throughput-bound or latency-bound. For instance, during model training—where the objective is to process internet-scale datasets—maximizing training throughput is paramount, and communication optimization primarily targets the efficient transmission of large data volumes, i.e., the throughput-bound regime [43, 115, 42]. In contrast, during inference serving, time to first token (TTFT) and the iterative nature of the decoding stage shift optimization focus toward minimizing the latency of small data exchanges, i.e., the latency-bound regime [35, 108]. In both scenarios, reducing communication overhead is crucial for achieving

high hardware utilization and system scalability.

Collective communication can be optimized across three layers: network topology, communication scheduling, and software–hardware implementation. The network topology forms the foundation for collective performance. For example, communication latency is fundamentally lower-bounded by the network diameter—the maximum number of hops between any two nodes. Moreover, topology design must be tailored to the target workload, as different applications demand distinct trade-offs among communication patterns and between throughput and latency. The communication scheduling (or algorithm) determines how to complete a collective pattern efficiently, minimizing latency and avoiding network congestion. For operations such as allreduce, the schedule also needs to coordinate data multicast and aggregation on accelerators. Finally, the software–hardware implementation lowers the communication schedule to execution on hardware, spanning from parallelization strategies to CUDA kernels. Efficient implementations must be tailored to the target hardware, balancing compute, communication, and memory needs according to the hardware characteristics.

1.2 Thesis Statement & Contributions

In this thesis, we demonstrate that optimizing network topology and communication scheduling can substantially improve the efficiency of software–hardware implementation for collective communication. These improvements in communication efficiency, in turn, translate directly into significant performance gains for distributed machine learning workloads.

First, we explore how to leverage optical circuit networking to accelerate collective communication in machine learning workloads. By exploiting the reconfigurability of optical circuits [62, 79, 63, 164], we co-optimize the network topology and communication schedule to achieve both low latency and high throughput. However, the discrete nature of graph and scheduling optimization often leads to NP-hard formulations [128, 81, 21, 22], such as mixed-integer linear programs (MILPs), which severely limit scalability. To overcome this challenge, we propose two key techniques: (1) expansion techniques that scale small, optimized topologies and schedules directly to large systems, and (2) a polynomial-time schedule generation algorithm with provable optimality on many commonly used topologies. We further develop a topology finder to identify the best-suited topology for a given latency-bandwidth trade-off, and a compiler to generate executable schedules for real hardware. Through evaluations on both real optical circuit hardware and large-scale simulations, we demonstrate that our co-optimized topologies and schedules significantly improve the efficiency of collective communication and, consequently, the performance of distributed ML training.

Second, we focus on collective communication over the network topologies of widely used, state-of-the-art machine learning GPU platforms, such as the NVIDIA DGX and AMD Instinct series. These systems comprise multi-GPU boxes interconnected by high-speed intra-box networks and lower-speed inter-box scale-out links. The diversity of topology designs and the inherent bandwidth heterogeneity present substantial challenges for communication scheduling, further complicated by the flexible traffic patterns enabled by network switches. To address these challenges, we develop *ForestColl*, a scalable, polynomial-time schedule generation algorithm inspired by spanning tree packing techniques. ForestColl constructs tree-flow schedules that achieve the theoretical optimal throughput for arbitrary network topologies. We first derive the throughput optimality for any topology—an open question prior to this research—and propose a suite of algorithmic techniques to logically eliminate switches and generate trees with minimum overlap/congestion. Our evaluation demonstrates that the generated schedules deliver significantly higher communication and training throughput compared to vendor-optimized libraries, and that our scheduling algorithm achieves orders-of-magnitude greater scalability and superior schedule performance over previous approaches.

Finally, we outline future work that further optimizes collective communication through improved parallelization strategies. We propose an abstraction in which parallelism and memory optimizations are represented as states (e.g., sharding pattern and processing strategy) of tensors within the compute graph. By determining the tensor states, the compute, communication, and memory costs of a compute graph can be inferred. This formulation enables automatic search for parallelization strategies that best match the characteristics of the target hardware, leading to superior end-to-end performance.

1.3 Published Works

- Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Jason Fantl, Prithwish Basu, Joud Khoury, Arvind Krishnamurthy. 2025. **Efficient Direct-Connect Topologies for Collective Communications.** *In Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '25).*

- Liangyu Zhao, Saeed Maleki, Yuanhong Wang, Zezhou Wang, Ziyue Yang, Hossein Pourreza, Arvind Krishnamurthy. 2026. **ForestColl: Throughput-Optimal Collective Communications on Heterogeneous Network Fabrics.** *In Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '26).*

Chapter 2

Background

2.1 Collective Operations

Collective operations are a set of standardized communication patterns with well-defined input and output states. In this thesis, our work focuses on three key collective operations—*reduce-scatter*, *allgather*, and *allreduce*—as they are the most widely used in distributed ML workloads. Table 2.1 summarizes the input and output states of these operations. In reduce-scatter, each node i computes the reduction of the i-th data shard contributed by all nodes. In allgather, each node i broadcasts its own i-th data shard to all other nodes. In allreduce, each node obtains the complete result of the reduction over data from all nodes.

While the input and output states of a collective operation are predefined, there exist various communication schedules (or algorithms) to complete it. Take allgather as an example: the traditional ring allgather algorithm propagates data shards in a circular fashion. In the first step, each node i sends its i-th data shard to node $(i+1) \bmod N$. In each subsequent step, every node forwards the shard it received in the previous step to node $(i+1) \bmod N$, and after $N-1$ steps, all nodes have received the complete set of data shards. The ring algorithm is often throughput-optimal, as each node simultaneously sends and receives an equal amount of data in every step, achieving perfect bandwidth utilization. However, its latency grows linearly with the number of nodes, requiring $\Theta(N)$ steps. To reduce latency, tree-based algorithms can be employed, lowering the step count to $\Theta(\log N)$. Nevertheless, tree-based approaches often have lower throughput than the ring algorithm, since maintaining balanced bandwidth utilization across the network is more challenging.

Collective operations underpin various forms of parallelism that enable distributed ML systems to scale. The most classic parallelism is data parallelism [75], where each node maintains a full replica of the model parameters and processes a distinct subset of the training data in each iteration. After the forward and backward

Input			Output		
Reduce-Scatter					
Node 0	Node 1	Node 2	Node 0	Node 1	Node 2
$S_0^{(0)}$	$S_0^{(1)}$	$S_0^{(2)}$	$\bigoplus_i S_0^{(i)}$		
$S_1^{(0)}$	$S_1^{(1)}$	$S_1^{(2)}$		$\bigoplus_i S_1^{(i)}$	
$S_2^{(0)}$	$S_2^{(1)}$	$S_2^{(2)}$			$\bigoplus_i S_2^{(i)}$
Allgather					
Node 0	Node 1	Node 2	Node 0	Node 1	Node 2
$S_0^{(0)}$			$S_0^{(0)}$	$S_0^{(0)}$	$S_0^{(0)}$
	$S_1^{(1)}$		$S_1^{(1)}$	$S_1^{(1)}$	$S_1^{(1)}$
		$S_2^{(2)}$	$S_2^{(2)}$	$S_2^{(2)}$	$S_2^{(2)}$
Allreduce					
Node 0	Node 1	Node 2	Node 0	Node 1	Node 2
$S_0^{(0)}$	$S_0^{(1)}$	$S_0^{(2)}$	$\bigoplus_i S_0^{(i)}$	$\bigoplus_i S_0^{(i)}$	$\bigoplus_i S_0^{(i)}$
$S_1^{(0)}$	$S_1^{(1)}$	$S_1^{(2)}$	$\bigoplus_i S_1^{(i)}$	$\bigoplus_i S_1^{(i)}$	$\bigoplus_i S_1^{(i)}$
$S_2^{(0)}$	$S_2^{(1)}$	$S_2^{(2)}$	$\bigoplus_i S_2^{(i)}$	$\bigoplus_i S_2^{(i)}$	$\bigoplus_i S_2^{(i)}$

Table 2.1: Reduce-Scatter, Allgather, and Allreduce Collective Operations.

passes, each node has the gradients of the model parameters with respect to its local data. To obtain a globally consistent model update, an allreduce operation is performed to average and synchronize the gradients across all nodes. As model sizes continue to grow, however, the parameters can no longer fit within the memory of a single GPU. To overcome this limitation, additional parallelism strategies have been developed, such as tensor parallelism [131, 99] and fully sharded data parallelism (FSDP) [161, 120]. In tensor parallelism, the model's weight matrices are partitioned across multiple GPUs; depending on the partitioning strategy, either allgather, reduce-scatter, or allreduce operation is required to exchange the resulting activations. Fully sharded data parallelism further improves scalability by sharding both model parameters and optimizer states across devices—performing allgather of parameters and reduce-scatter of gradients during the training iteration.

2.2 Network Topologies in Distributed Machine Learning Systems

As distributed machine learning systems scale to tens of thousands of accelerators or more, designing network architectures that deliver scalability while maintaining performance and cost efficiency becomes a critical challenge. Unlike traditional datacenter networks, where traffic patterns are random and unpredictable, distributed ML workloads exhibit highly regular traffic patterns arising from fixed parallelism strategies and collective communication operations. These structured patterns create opportunities for specialized

optimization. In this thesis, we explore two approaches: optical circuit networks and multi-GPU box platforms.

2.2.1 Optical Circuit Networking

Compared to traditional electrical networking, optical circuit networking offers an order of magnitude higher bandwidth while significantly reducing capital and energy costs, making it a promising foundation for large-scale distributed machine learning systems. Unlike electrical packet switching, which performs per-packet routing to enable all-to-all connectivity, optical circuits require physical reconfiguration—typically on the order of milliseconds [79]—to change their connectivity, and can therefore only support partially connected topologies, where each node connects to a limited subset of peers. Hybrid designs [164, 141] have been proposed to combine optical circuits with electrical packet switching, but such approaches diminish the advantages of optical networking and reintroduce the same bottlenecks inherent to electrical packet switches [63, 152]. Leveraging the regularity of communication patterns in ML workloads, recent systems [63, 152, 62] adopt a direct-connect approach, where optical circuits directly link endpoints without passing through electrical packet switches. Due to the high reconfiguration latency, the network topology must remain static over a certain duration—typically the entire collective operation. Consequently, achieving high-performance collective communication over optical circuit networks requires joint optimization of the network topology and communication schedule.

2.2.2 Multi-GPU Box Platforms

Traditional GPU vendors such as NVIDIA and AMD adopt a different architectural approach: they integrate 4 or 8 GPUs within a single server (box) using a high-speed scale-up network, and then interconnect many boxes through a relatively lower-speed scale-out network. The intra-box scale-up network typically consists of direct GPU interconnects such as NVLink or AMD Infinity Fabric, or a high-speed electrical switch like NVSwitch or PCIe switch. In contrast, the inter-box scale-out network follows conventional datacenter design, often employing topologies like fat-tree [3] to provide flexible all-to-all connectivity. Unlike optical circuit networks, however, the topology of multi-GPU box platforms is static and highly heterogeneous across the scale-up and scale-out domains: scale-up networks deliver bandwidths on the order of hundreds of GB/s per GPU, whereas scale-out networks typically remain in the tens of GB/s range. Moreover, scale-up topologies exhibit significant diversity, both across GPU vendors and across successive generations of the same vendor. Consequently, the key challenge lies in developing a general solution to communication scheduling that can adapt to these diverse and heterogeneous topologies.

Chapter 3

Efficient Direct-Connect Topologies for Collective Communications

3.1 Introduction

Collective communication operations involve concurrently aggregating and distributing data on a cluster of nodes and are used in both machine learning (ML) and high-performance computing (HPC). With the improved computational capabilities of accelerators, collective operations are a significant overhead in large-scale distributed ML training [127, 45, 145, 86].

An emerging approach to address these challenges has been to employ various forms of optical circuit switching to achieve higher bandwidth at reasonable capital expenditure and energy costs [63, 152, 164, 85, 141, 62, 79]. Hosts communicate using a limited number of optical circuits that can be reconfigured at timescales appropriate for the hardware, thus exposing network topology as a configurable component. We refer to this setting as *direct-connect* with circuits configured and fixed for an appropriate duration.

Existing optical-circuit-based ML systems [63, 164, 85, 141] fit this direct-connect model but do not exploit the flexibility topology reconfiguration offers. Collective operations such as allreduce are still limited to a few well-known algorithms that can fit the degree constraints of the optical fabric (e.g., rings, multi-rings, tori) and accept the consequent performance tradeoffs. For example, ring allreduce, while bandwidth-efficient, has a high graph diameter, causing high total-hop latency. A double binary tree, on the other hand, has a logarithmic diameter but suffers from load imbalances and bandwidth inefficiencies. Conversely, the broader spectrum of well-known collective algorithms that achieve desired latency and bandwidth (e.g., recursive-doubling, Bruck algorithm) [139, 154] use dynamic communication patterns ideal for switch networks but are ill-suited for degree-constrained direct-connect networks.

To fill this gap, we seek to identify new custom-built topologies and communication schedules for

direct-connect networks. We pose the following question: *How to efficiently construct high-performance direct-connect topologies and communication schedules for collectives given the network's performance characteristics and degree constraints?*

This question poses several challenges. First, jointly optimizing *both* the network topology and the corresponding communication schedule is intractable at a large scale. Prior efforts reduce the search cost by optimizing only one or the other (e.g., schedules for a given topology [145, 21, 128] or topology permutations while retaining a ring schedule [152]). The combination of topological structure and communication schedule as degrees of freedom explodes the search space, making this a seemingly intractable problem. Second, the optimization must carefully consider the workload and the network's performance characteristics when distilling a topology and schedule. For example, minimizing the topology's diameter is ideal not only for latency-sensitive allreduce at small data sizes but also for all-to-all throughput; however, this could come at the cost of load imbalance across links in bandwidth-sensitive allreduce at large data sizes. Finally, lowering the synthesized schedules to the underlying hardware and runtimes [95, 55] in an efficient way requires careful scheduling to achieve the desired performance in practice.

Our work addresses these issues by developing an algorithmic toolchain for quickly synthesizing efficient topologies and schedules for collective communications.

1. We devise a range of **expansion techniques** for synthesizing custom large-scale network topologies and schedules. The expansions start with small, optimal topologies and communication schedules and expand them to achieve near-optimal large-scale topologies and schedules.

2. We devise a **polynomial-time schedule generation algorithm** to produce optimal collective communication schedules for large-scale topologies with specific symmetry properties. This exposes many well-known topologies as options for the direct-connect network fabric.

3. We devise a **topology enumeration and search algorithm** to identify the best option for a target cluster and workload by exploring the *Pareto-efficient* options that provide different tradeoffs for bandwidth efficiency, total-hop latency, and also all-to-all throughput.

4. We develop **compilers** to realize our optimized schedules. We offer efficient implementations for both GPUs and CPUs, integrating with ML frameworks (e.g., PyTorch) through the MSCCL [95] and oneCCL [55] runtimes.

We evaluate our approach using two testbeds: a 12-node GPU cluster capable of topology reconfiguration and torus clusters on Frontera [134] supercomputer with up to 54 CPU nodes. Our techniques reduce collective communication times by $>30\%$ for DNN training on the GPU testbed and up to $3.1\times$ in supercomputing settings. Simulations for large-scale DNN training show up to an order of magnitude reduction in total communication time from topology and schedule optimization. Our schedule generation algorithm is orders of magnitude faster than the state-of-the-art (e.g., SCCL [21] and TACCL [128]), capable of producing schedules for topologies with thousands of nodes in a minute.

3.2 Background & Related Work

3.2.1 Network Fabric

Our work identifies topologies and schedules helpful for a broad range of settings, such as *switchless physical circuits*, *patch-panel optical circuits*, and *optical circuit switches*. While these options differ in cost and reconfigurability [152], they are all significantly cheaper than packet-switch solution [62, 79, 152] and can benefit from our work.

Switchless physical circuits require the least amount of fabric hardware. However, the topology must remain reasonably static for long periods, as the reconfiguration is manual. *Patch-panel optical circuits* provide a higher degree of reconfigurability by using a mechanical solution (e.g., robotic arms) to perform physical reconfigurations through a patch panel. The reconfigurations occur on the scale of minutes, but the patch panel itself can scale to a large number of duplex ports and is reasonably cheap (e.g., 1008 ports at $100 per port [137]). Both options can benefit from a carefully curated topology optimized for the workload, but they require it to remain static for a job given the reconfiguration time.

Commercial *optical circuit switches (OCS)* can perform reconfigurations in $\approx$10ms, are more expensive than patch panels, and scale to fewer ports (e.g., Polatis 3D-MEMS switch has 384 ports at $520 per port). Though OCSes support faster reconfigurations, the delays are still too high to support the rewiring of the circuits *during* a typical collective operation.[1] Thus, they cannot take advantage of algorithms designed for full-bisection switches, such as recursive halving/doubling [139, 154], that exploit high logical degree over time to provide both latency and bandwidth optimality. Thus, OCSes can also benefit from the custom-built

[1] Research prototypes [92, 8] support μs to ns reconfigurations using overlay-hop relays and Valiant load balancing (VLB). While this is valuable for generic workloads, collectives have structured communication patterns, and it would be ideal to realize them without incurring the VLB overheads.

and low-degree topologies synthesized by our approach.

All of these optical technologies allow for a shared cluster to be split into multiple subclusters for running separate jobs [62], so each job can be configured with its own topology. Further, unidirectional topologies are technically feasible on optical testbeds. Unidirectionality gives greater freedom in topology design and can enable lower-diameter networks.

Evaluation Target: In this paper, we use a reconfigurable optical patch panel to configure and evaluate different topologies. Given the high reconfiguration costs for the patch panel, we identify an efficient topology that will remain static for the duration of a job. Nevertheless, our techniques could be used to derive topologies for finer reconfiguration timescales if the hardware can efficiently support frequent reconfigurations.

3.2.2 Related Work

Several existing optical-circuit-based ML systems [63, 164, 85, 141, 152] fit the direct-connect model; however, they rely on existing implementations of collectives. Typically, communication libraries for ML training [127, 103, 45] offer either ring collective for high-latency bandwidth-optimal transfers or tree collective, which has logarithmic latency but suffers from load-imbalances across links. Other topologies such as mesh, tori, hypercubes, etc., have also been explored in HPC systems [18, 50, 11, 110, 23, 24, 39], but their bandwidth-latency tradeoff choices are limited as well. Bandwidth and latency optimal collectives for switch networks such as recursive-doubling, Bruck algorithm [154, 139], BlueConnect [28], etc., are unsuitable for direct-connect networks, because their one-to-one communication patterns fail to utilize all available links, and they assume a fully connected network.

Our work uniquely considers joint optimization of *both* the network topology and the corresponding collective communication schedule at a large scale, while prior work either optimizes one or the other. For instance, TopoOpt [152] generates customized shifted-ring topologies to optimize concurrent collective and non-collective communications for hybrid data-parallel [75, 74, 34] and model-parallel [131, 99, 59] DNN training, respectively. The collective communications in TopoOpt still use existing ring collectives. Consequently, when data parallelism, for example, dominates the workload, TopoOpt's performance suffers from similar latency issues present in existing ring collectives. Our effort is complementary as it synthesizes new topologies and schedules for collectives that span the entire cluster, but we do not optimize sub-cluster communications for hybrid parallelism. Extending our work to jointly optimize topologies and schedules for hybrid parallelism is future work.

Recent work like Blink [145], SCCL [21], and TACCL [128] also focus on generating a collective schedule for a given topology. However, they all involve NP-hard optimizations that severely limit their scalability. SCCL is capable of generating optimal schedules, but it fails to generate a schedule in a reasonable time when the topology size is beyond 30 nodes. TACCL improves the scalability of SCCL by using communication sketches and also handles switch networks, but it sacrifices schedule performance and is still limited in scalability. In our approach, we either synthesize the schedule along with the topology or rely on a polynomial-time schedule generation technique that is provably optimal for networks with certain symmetry properties.

Generic large-scale topologies are typically not optimized for collective communications but for general datacenter traffic [133, 143, 64, 16, 67, 159]. Our framework can incorporate any degree-constrained regular topology (e.g., low-diameter expander graphs [122, 54]) and generate candidate schedules.

3.2.3 All-to-All Throughput

While we optimize allreduce, reduce-scatter, and allgather, the performance of all-to-all communication is also crucial for training DNN models like Mixture of Experts (MoE) [70, 40, 119, 73] and Deep Learning Recommendation Model (DLRM) [101, 100, 87]. Unlike other collectives, the scheduling of all-to-all can be easily formulated and efficiently solved as a multi-commodity flow (MCF) problem [12, 153, 65, 47, 143]. However, the graph diameter of the underlying topology is critical for all-to-all throughput [12, 84, 83, 152, 92]. The intuition is that if the nodes are far from each other, then all-to-all flows cost more *bandwidth tax* [91, 152] (i.e., the bandwidth of the flow multiplied by the length of the flow). With a fixed *network capacity* (i.e., the total number of links times link bandwidth), longer flows reduce the available bandwidth for each flow, thus decreasing all-to-all throughput. In this work, we construct topologies and associated schedules that are high-performance in both allreduce-type collectives and all-to-all by **deriving efficient allreduce-type schedules on existing or our synthesized low-diameter topologies**.

3.3 Formal Model of Collective Communications

We provide a formal model of ***reduce-scatter***, ***allgather***, and ***allreduce*** collectives. In each operation, there are N nodes operating on a vector of data of total size M. The data can be divided into N **shards**. In *reduce-scatter*, each node i reduces the i-th shard from all other nodes; in *allgather*, each node i broadcasts the i-th shard to all other nodes; in *allreduce*, each node i ends up with the fully reduced vector of data.

Throughout the paper, we elaborate only on *allgather* schedule construction because the other two collectives are direct transformations. Since *allgather* and *reduce-scatter* are, respectively, simultaneous broadcasts

M	total data size	α	single-hop latency
N	number of nodes	B	total egress bandwidth of a node
S	data shard ($\|S\| = \frac{M}{N}$)	B/d	bandwidth of a single link
C	data chunk ($C \subseteq S$)	$T_L(A)$	total-hop latency of schedule
d	degree of topology	$T_B(A)$	bandwidth runtime of schedule
V_G	vertex/node set of G	$T_L^*(N,d)$	Moore optimality (Def 10)
E_G	edge/link set of G	$T_B^*(N)$	bandwidth optimality $\frac{M}{B} \cdot \frac{N-1}{N}$
$D(G)$	graph diameter of G	$N_x^+(u)$	nodes at distance x from u
Table A.3 for graph symbols		$N_x^-(u)$	nodes at distance x to u

Table 3.1: Summary of Important Notations

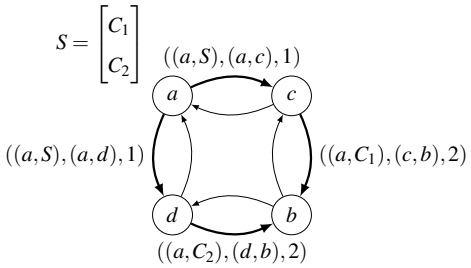


Figure 3.1: The allgather schedule of complete bipartite graph $K_{2,2}$. Shard S is divided into two half chunks C_1 and C_2. From a, at the 1st comm step, a sends the entire shard S to both c and d. At the 2nd comm step, c and d send the two half chunks C_1 and C_2, respectively, to b. Thus, every node receives the full shard from a. By applying similar broadcast from c,b,d in parallel, we have a complete BW-optimal *allgather* schedule with $T_L = 2\alpha, T_B = \frac{M}{B} \cdot \frac{3}{4}$.

and reductions for each node, we can **construct *reduce-scatter* schedules** in bidirectional topologies by simply reversing the communications in *allgather* schedules [23]. In unidirectional topologies, we utilize graph transposition to achieve a similar transformation (Appendix A.2).[2] **To construct an *allreduce* schedule,** we concatenate *reduce-scatter* and *allgather*.

3.3.1 Communication Topology & Schedule

The network topology is modeled as a directed graph (digraph) $G = (V,E)$, where V denotes the set of nodes ($|V| = N$) and E denotes the set of directed links/edges. The direct-connect network imposes a constraint that all nodes have degree d, which is the number of connection ports on each host and is typically low and independent of N.

[2]The main text of this paper focuses on high-level ideas of various techniques. We provide detailed mathematical analyses in the appendix.

A communication **algorithm** (G,A) uses the communication **schedule** A on topology G. Schedule A can be specified as what chunk C is communicated over which link in which **communication (comm) step** t. We define **chunk** C as a subset of shard S. Both C and S are specified as *index sets* of elements. Typically, S is interval $[0,1]$ representing the whole shard, and C is some subinterval. We denote v's chunk C as (v,C), which is a subset of v's starting shard (v,S). Let $((v,C),(u,w),t)$ denote that v's chunk C is sent by node u to its neighbor w at comm step t. Schedule A then is specified as a list of tuples $((v,C),(u,w),t)$. Figure 3.1 gives an example of an allgather schedule in such a tuple notation. Within a schedule, chunks can be different-sized subsets of S. Appendix A.2 gives formal definitions of reduce-scatter/allgather schedules.

3.3.2 Cost Model

We use the well-known α-β cost model [51]. The cost of sending a message of size H over a link is $\alpha+\beta H$. This cost comprises two components: the constant single-hop latency α and a bandwidth component β, which is the inverse of link bandwidth, i.e., $\beta=\frac{1}{b}$. This simple model has been shown to be appropriate for GPU interconnects [71, 21, 128]. In our analysis, we use node bandwidth B with $B=db$. In this paper, we focus on homogeneous networks, although some techniques also support heterogeneous ones (Appendix A.5.3).

The runtime of a schedule A can be broken down into a total-hop latency component and a bandwidth component. The **total-hop latency** component $T_L(A)=t_{\max}\alpha$, where $t_{\max}$ is the number of comm steps. $T_L(A)$ represents the cost of performing schedule A on an infinitesimal amount of data. The bandwidth component $T_B(A)$, or **bandwidth (BW) runtime**, is the sum of the BW runtime of each comm step, i.e., $\sum_t T_B(A_t)$. The BW runtime of comm step t is the max amount of data transmitted by a link within the comm step, divided by link bandwidth $b=B/d$. For N-node d-regular graphs, the optimal runtime of both *reduce-scatter* and *allgather* is approximately $\alpha\log_d N+\frac{1}{B}\cdot\frac{M(N-1)}{N}$. The 1st term represents the total-hop latency required for communicating across the diameter of a topology, while the 2nd term represents the transmission time for any node to send/recv $N-1$ shards in reduce-scatter/allgather. **One should not confuse total-hop latency with overall latency,** which is the sum of total-hop latency and BW runtime. We omit the computational time of reduction and discuss this in Appendix A.3.4.

We analyze the optimality of total-hop latency and BW runtime separately. An algorithm (G,A) is optimal in one component if no (G',A') with the same N,d can perform better. For BW runtime, an algorithm is **bandwidth (BW) optimal** iff its T_B equals $T_B^*(N):=\frac{M}{B}\cdot\frac{N-1}{N}$. For total-hop latency, given G, the lowest T_L achievable is $\alpha\cdot D(G)$, where $D(G)$ is the graph diameter of G. Thus, the optimal total-hop latency equals

Topology Type	Small-Data Allreduce (Total-Hop Latency T_L)	Large-Data Allreduce (BW Perf T_B)	All-to-All Throughput
Low-Hop	✓	–	✓
Load-Balanced	–	✓	–

Table 3.2: The tradeoffs of low-hop topology vs. load-balanced topology. Reduce-scatter and allgather perform similarly to allreduce.

the smallest diameter of any N-node d-regular graph, which remains an open question in graph theory [96]. Therefore, we define *Moore optimality* based on *Moore bound*, which provides a lower bound for diameter given N, d and thus a well-defined $T_L^*(N,d)$. An algorithm is **Moore optimal** iff $T_L = T_L^*(N,d)$. Moore optimal topologies have the lowest diameters, which is also ideal for all-to-all throughput. Appendix A.3 gives formal definitions of optimalities.

Ring allreduce has a total-hop latency that is linear in N, while the BW runtime is optimal. *Double binary trees* (DBT), on the other hand, offer the advantage of logarithmic total-hop latency but have suboptimal BW performance. Our work offers a range of topologies that are Pareto-efficient in total-hop latency and BW performance.

3.4 Overview of Our Approach

Direct-connect topologies can typically be categorized as either **low-hop** topologies, which have low diameters (e.g., expander graphs) suited for all-to-all throughput and small-data allgather/reduce-scatter/allreduce, or **load-balanced** topologies, which have simplistic structure (e.g., ring, torus) with easy load-balanced schedule for large-data allgather/reduce-scatter/allreduce (see Table 3.2). We seek to jointly identify network topologies and schedules that achieve high performance in both categories to the extent possible. Specifically, this entails the challenging task of constructing load-balanced allgather[3] schedules for low-hop topologies.

At a small scale, one could handpick a topology such as the complete bipartite graph $K_{2,2}$ defined at $N=4, d=2$. $K_{2,2}$ is both low-hop and load-balanced that a Moore- and BW-optimal allgather could be manually constructed (Figure 3.1). But how do we scale the topology and the schedule to larger sizes? Our work approaches this problem with two tools: *expansion techniques* (§3.5) and *BFB schedule generation* (§3.6).

Expansion Techniques: Given a base topology and its schedule, expansion techniques can expand them into a larger topology and associated schedule with minimal loss in performance. We call the resulting

[3]We construct allreduce and reduce-scatter from allgather schedules.

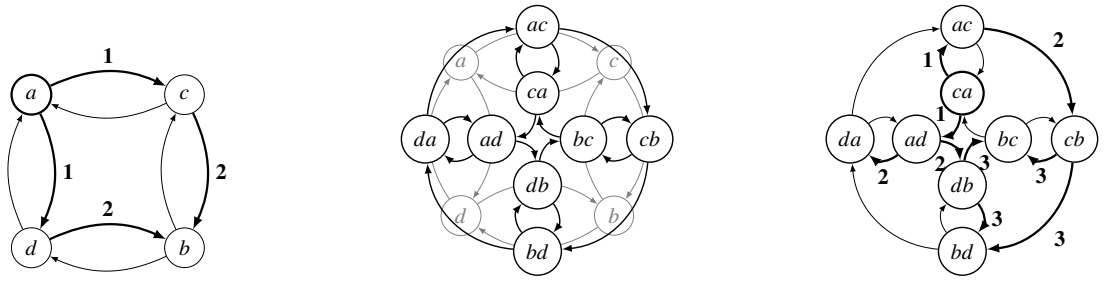
topologies **synthesized topologies**. The base topologies are small in scale, such as $K_{2,2}$ in Figure 3.1, for which straightforward schedules exist or an exhaustive search for the schedule is feasible. The line graph expansion, for example, can then expand $K_{2,2}$ and its schedule in Figure 3.1 to an allgather for $N = 4 \cdot 2^n$, for arbitrarily large n, while retaining a node degree of 2. Multiple expansion techniques can be composed to achieve the desired N and d.

Breadth-First-Broadcast (BFB) Schedule Generation: Besides synthesized topologies, we can use known topologies from graph theory (e.g., twisted torus, expander graphs). We call them **generative topologies** as they can be instantiated at various Ns and ds. Generative topologies are often low-hop, beneficial for total-hop latency and all-to-all throughput. The problem, though, is that efficient load-balanced collective schedules are not known for many of these topologies, and existing schedule generation methods [21, 128] are intractable even at moderate scales. Our work offers BFB, a polynomial-time schedule generation that can yield efficient schedules for large-scale topologies. For allgather, it performs a breadth-first broadcast from each node and uses linear programs to balance the workload on links. Although not always optimal, BFB schedules are provably optimal for many topologies exhibiting certain symmetries. For instance, BFB can generate a schedule with the lowest total-hop latency and BW optimality on any torus, including those with *unequal dimensions*.

With expansion techniques and BFB schedules, our *topology finder* (§3.5.4) assembles a large pool of topologies and schedules, identify *Pareto-efficient* ones from a low-hop vs. load-balanced perspective, and select from them for a given workload. When two options are Pareto-efficient, one must be better than the other in either low-hop (i.e., total-hop latency) or load-balanced (i.e., BW performance) but not in both. We choose low-hop options for workloads requiring all-to-all throughput and small-data allgather/reduce-scatter/allreduce, and load-balanced options for large-data allgather/reduce-scatter/allreduce. Finally, the *compiler* (§3.7) lowers the chosen topology and schedule to the runtime and hardware.

3.5 Expansion Techniques

We present three techniques that can be applied to construct near-optimal large-scale *synthesized topologies* and schedules by expanding small-scale topologies and associated schedules. The three techniques provide different options for increasing the size of the topology and the per-node degree while preserving either total-hop latency or BW optimality of the base graph and schedule (Table 3.3). While we describe the techniques in the context of allgather, corollary 1.2 in §A.2 implies equivalent constructions for reduce-scatter



(a) $K_{2,2}$ $(N=4, d=2)$ (b) $L(K_{2,2})$ $(N=8, d=2)$ (c) Broadcast from ca

Figure 3.2: The complete bipartite topology $K_{2,2}$ with its line graph $L(K_{2,2})$. Figure (a) shows the base topology and broadcast paths from a to c, b, d in $A_{K_{2,2}}$ (see Figure 3.1). The number next to edge shows the comm step using the edge. Figure (b) shows the expanded topology. Observe that every edge in $K_{2,2}$ becomes a vertex in $L(K_{2,2})$, and two vertices are connected if the corresponding edges in $K_{2,2}$ have one's head node being the other's tail node. Figure (c) shows the broadcast paths of node ca, transformed from the broadcast paths of a in figure (a). At the 1st comm step, by step 1 of Def 1, ca broadcasts its shard to all its neighbors: $((ca,S),(ca,ac),1)$, $((ca,S),(ca,ad),1)$. The rest of the broadcast paths are transformed from $A_{K_{2,2}}$ by step 2 of Def 1, e.g. $((a,C_1),(c,b),2) \mapsto \{((ca,C_1),(cb,bc),3),((ca,C_1),(cb,bd),3)\}$. Each of the nodes bc and bd receives C_1, C_2 from its two in-neighbors, just like b does in $A_{K_{2,2}}$.

and allreduce.

3.5.1 Line Graph Expansion

We borrow the line graph transformation from graph theory [49], which transforms an input graph G into a larger graph $L(G)$ as follows: every edge in G becomes a node in $L(G)$, and two nodes in $L(G)$ are adjacent if the corresponding edges are adjacent in G (Definition 12).

Intuition: Line graph expands an N-node degree-d topology into a dN-node topology. The degree d remains the same, which is crucial since the degree is often limited by hardware constraints like the number of available ports. While the number of nodes grows by d-fold, the diameter of the topology only increases by one, which is also optimal for total-hop latency and all-to-all performance. In addition, the paths in the base topology are mapped into the expanded topology, allowing the communication schedule for the base to be expanded as well. Line graph expansion can be applied repeatedly to scale the topology and schedule to arbitrarily large sizes.

Figure 3.2 gives an example of the line graph expansion of the complete bipartite graph $K_{2,2}$. Any (shortest) path $w_0 \to w_1 \to \ldots \to w_n$ in $K_{2,2}$ can be mapped to a (shortest) path $w_{-1}w_0 \to w_0w_1 \to \ldots \to w_{n-1}w_n \to w_nw_{n+1}$ in $L(K_{2,2})$ from $w_{-1}w_0$ to w_nw_{n+1}, for any w_{-1}, w_{n+1} provided that $w_{-1}w_0 \neq w_nw_{n+1}$.

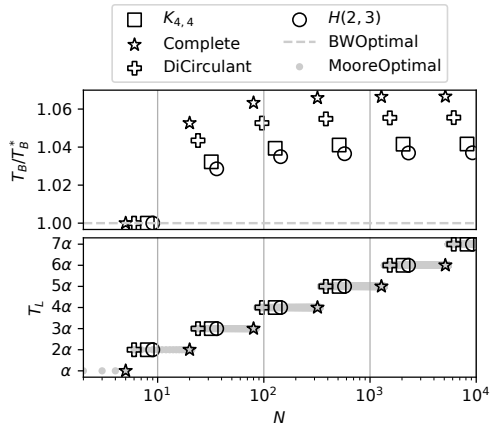


Figure 3.3: Line graph expansion on Moore and BW optimal degree-4 base graphs: complete bipartite graph $K_{4,4}$, complete graph, directed circulant graph, and Hamming graph $H(2,3)$. $T_B^* = \frac{M}{B} \cdot \frac{N-1}{N}$ is the optimal BW runtime.

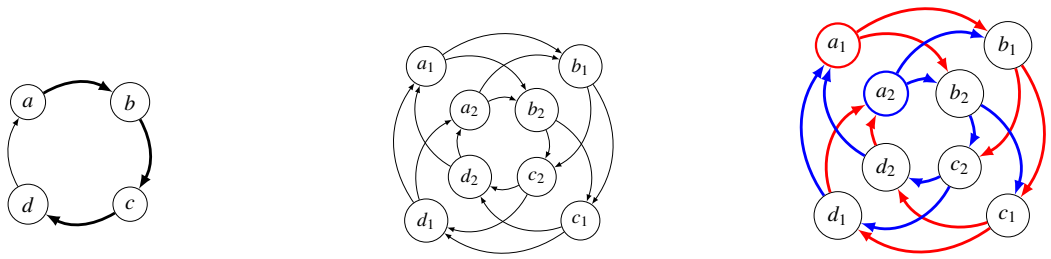
Given an allgather schedule A_G for G, we construct schedule $A_{L(G)}$ for $L(G)$. Pick any node $v'v$ in $L(G)$. It needs to broadcast its shard to every other node in $L(G)$. Pick any other node, say, uu'. For each element x of $v'v$'s shard, we want to send x to uu'. Since v broadcasts x to every other node in A_G, there is a path $v \to w_1 \to \ldots \to w_{n-1} \to u$ in G along which x is sent to u in A_G. Thus, the path $v'v \to vw_1 \to w_1w_2 \to \ldots \to w_{n-1}u \to uu'$ can be utilized to send x from $v'v$ to uu' in $L(G)$.

Definition 1 (Schedule of Line Graph)**.** *Given an allgather schedule A_G for topology G, let $A_{L(G)}$ be the schedule for line graph $L(G)$ containing:*

1. $((v'v, S), (v'v, vu), 1)$ *for each edge* $(v'v, vu) \in E_{L(G)}$ *with* $v'v \neq vu$. **[Insert the 1st comm step in $A_{L(G)}$.]**
2. $((v'v, C), (uw, ww'), t+1)$ *for each* $((v,C),(u,w),t) \in A_G$ *and* $v'v \neq ww'$. **[Adapt A_G to form $A_{L(G)}$.]**

At the 1st comm step, x is broadcasted by $v'v$ to every neighbor, including vw_1. Then, for every $((v,C),(w_i,w_{i+1}),t)$ in A_G with $x \in C$, there is $((v'v,C),(w_iw_{i+1},w_{i+1}w_{i+2}),t+1)$ in $A_{L(G)}$ that takes x from w_iw_{i+1} to $w_{i+1}w_{i+2}$ and, eventually, to uu' ($v = w_0, u = w_n$). Since x and uu' are picked arbitrarily, $v'v$ broadcasts every element of its shard to all nodes in $L(G)$. Figure 3.2c shows an example of schedule construction.

As for the performance of $A_{L(G)}$, we leave the mathematical details in Appendix A.4.1. In practice, one can apply line graph expansion repeatedly to scale the topology and schedule indefinitely. The more optimal the base topology and schedule are, the more optimal the expanded topology and schedule will be. Figure 3.3 shows how the performance evolves as we continuously apply line graph expansion to several Moore and BW optimal base graphs. The total-hop latency always remains Moore optimal. The BW performance deviates



(a) $G(N=4,d=1)$ (b) $G*2$ $(N=8,d=2)$ (c) Broadcasts from a_1, a_2

Figure 3.4: 4-node unidirectional ring and its degree expansion to $d=2$. Figure (a) shows the base topology and broadcast path from a to b,c,d. Figure (b) shows the expanded topology. Figure (c) shows the broadcast paths from a_1 and a_2 to other nodes marked in red and blue, respectively. For any $u \neq a$, the path from a_i to u_j stays in i until the very last step when it jumps to u_j, e.g., $a_1 \to b_1 \to c_2$ and $a_2 \to b_2 \to c_2 \to d_1$. For a_i to a_j, each in-neighbor of a_j sends an equal portion of a_i's shard to a_j in the end; for example, d_1 and d_2 each send half of a_1's shard to a_2 and half of a_2's shard to a_1. The red and blue broadcast paths are disjoint, resulting in BW optimality.

from optimality T_B^* but remains a constant factor away asymptotically. A key observation in Figure 3.3 is that the larger the size of the base graph is, the closer the expanded schedule is to BW optimality. Line graph expansion is notable for its ability to construct indefinitely large-scale topologies without increasing degree d. The expansion also maintains low-hop, making it ideal for synthesizing all-to-all topologies as well.

3.5.2 Degree Expansion

Intuition: While line graph expansion expands the number of nodes, degree expansion additionally expands the topology degree. Taking a base topology G, we make n copies of it and connect two nodes in different copies if they are adjacent in G. This process forms an expanded topology $G*n$, which multiplies both the number of nodes and the degree of G by n. Because the connections in $G*n$ are derived from G, similar to line graph expansion, we can map paths from G to $G*n$ to expand the communication schedule of G as well.

Figure 3.4 gives an example of expanding a 4-node unidirectional ring into an 8-node degree-2 topology (see formal definition of degree expanded topology in Definition 13). Based on the input schedule A_G for G, we construct a schedule A_{G*n} for $G*n$. For any data traveling along $v \to w^{(1)} \to \ldots \to w^{(m)} \to u$ in A_G, A_{G*n} has the data travel along $v_i \to w_i^{(1)} \to \ldots \to w_i^{(m)} \to u_j$ for all i, j. That is, data is transmitted within the i-th copy of G, except at the last step. With this construction, any node u_i has broadcasted the data to all other nodes except its own copies u_js. We add an additional comm step for u_j to collect the data from its in-neighbors (see Figure 3.4c).

Definition 2 (Degree Expanded Schedule)**.** *Given an allgather schedule A_G for G, construct A_{G*n} for $G*n$:*

1. *For all i,j including $i=j$ and for each $((v,C),(u,w),t)\in A_G$, add $((v_j,C),(u_j,w_i),t)$ to A_{G*n};*
2. *Divide shard S into equal-sized chunks $C_1,\ldots,C_{nd}$. Given $u_i,u_j\in V_{G*n}$ with $i\neq j$, add $((u_i,C_\alpha),(v_\alpha,u_j),t_{\max}+1)$ to A_{G*n} for each $(v_1,u_j),\ldots,(v_{nd},u_j)\in E_{G*n}$, where $t_{\max}$ is the max comm step in A_G.*

Unlike line graph expansion, degree expansion preserves BW optimality. This is because the expanded broadcast paths from copies of an original node are totally disjoint from each other (Figure 3.4c). However, degree expansion does not preserve Moore optimality. While line graph expansion does not change degree, degree expansion increases it, reducing the number of comm steps required for Moore optimality.

3.5.3 Cartesian Product Expansion

The Cartesian product of two graphs G_1,G_2 is an expanded graph $G_1\Box G_2$ with size and degree equal to the product of G_1,G_2's sizes and the sum of their degrees, respectively.

Definition 3 (Cartesian Product)**.** *The Cartesian product digraph $G_1\Box G_2$ of digraphs G_1 and G_2 has vertex set $V_{G_1}\times V_{G_2}$ with vertex $\mathbf{u}=(u_1,u_2)$ connected to $\mathbf{v}=(v_1,v_2)$ iff either $(u_1,v_1)\in E_{G_1}$ and $u_2=v_2$; or $u_1=v_1$ and $(u_2,v_2)\in E_{G_2}$.*

This definition generalizes to the Cartesian product of n digraphs: $G_1\Box G_2\Box\ldots\Box G_n$. When $G_1=\ldots=G_n=G$, the product is denoted as Cartesian power $G^{\Box n}$. We use Cartesian power and product in our topology and schedule expansion.

Intuition: The Cartesian product $G_1\Box G_2\Box\ldots\Box G_n$ consists of n dimensions, with connections in dimension i identical to G_i. Taking the schedules of $G_1,G_2,\ldots,G_n$, we can balance the amount of traffic going through each dimension to achieve high BW performance. Cartesian product expansion greatly expands the set of topologies we construct by enabling the combination of existing topologies to form a new product topology with an efficient schedule.

Cartesian Power Expansion: Given a d-regular G and schedule A_G, we can construct a schedule $A_{G^{\Box n}}$ for $G^{\Box n}$, which is nd-regular and has $|V_G|^n$ nodes. This technique helps generate efficient topologies, including some well-known ones like hypercube and Hamming graph. We describe how to construct allgather schedules for Cartesian power graphs by using $\ell\times\ell$ torus (ℓ-ring$^{\Box 2}$) as an example. A typical allgather schedule on an $\ell\times\ell$ torus is to perform the ℓ-ring allgather along rings in one dimension first and then the other dimension, as in hierarchical ring allreduce [142]. Consider two schedules: $A^{(1)}$ performs allgather on vertical rings first and then horizontal ones; $A^{(2)}$ performs allgather in the opposite order. $A^{(1)},A^{(2)}$ use disjoint set of links at

Expansion Techniques	# of Nodes	Deg	Moore	BW	Perf
Line Graph Exp $L^n(G)$	d^nN	d	✓	×	Thm 7.1
Degree Exp $G*n$	nN	nd	×	✓	Thm 11
Cartesian Power $G^{\square n}$	N^n	nd	×	✓	Thm 12
Cartesian Prod $G_1\square\ldots\square G_n$	$\prod_i N_i$	$\sum_i d_i$	×	✓	Thm 13

Table 3.3: Summary of expansion techniques. The table shows the characteristics of the resulting topology and schedule after applying expansion techniques to an N-node degree-d base topology. "✓, ×" show whether the expansion preserves Moore/BW optimality. The last column refers to the theorems that give the exact performance of expanded schedules. Appendix A.4 presents more formal definitions and analyses of expansion techniques.

Topology	T_L	T_B	$2(T_L+T_B)$	$D(G)$	All-to-All
$\Pi_{4,1024}$	5α	$1.332M/B$	323.5us	5	409.1us
$L^3(C(16,\{3,4\}))$	6α	$1.020M/B$	291.0us	6	403.5us
$L^2(\text{Diamond}^{\square 2})$	8α	$1.004M/B$	328.4us	8	446.6us
$L(\text{DBJMod}(2,4)^{\square 2})$	11α	$1.000M/B$	387.8us	9	529.9us
$(\text{UniRing}(1,4)\square\text{UniRing}(1,8))^{\square 2}$	20α	$0.999M/B$	567.6us	20	1174.4us
Theoretical Bound	**5α**	$\mathbf{0.999}M/B$	**267.6us**	**5**	**382.3us**

Table 3.4: Pareto-efficient topologies at $N=1024$, $d=4$. The $2(T_L+T_B)$ column shows the allreduce runtimes for $\alpha=10$us and $M/B=1$MB$/100$Gbps. We multiply T_L+T_B by 2 because allreduce is performed by combining reduce-scatter and allgather. The all-to-all time is computed via multi-commodity flow (Appendix A.1.5) with each node having 1MB of data to send (i.e., sending $1/N$ MB to each node). For comparison, the baselines Shifted Ring and Double Binary Tree (§3.8.2) have allreduce times of 20640us and 1434us, and all-to-all times of 10738us and 21475us, respectively. Table A.3 shows the details of the base topologies.

any comm step. Thus, we divide each data shard in $\ell\times\ell$ torus into two halves and let them be allgathered by $A^{(1)},A^{(2)}$ separately. The combined schedule, where $A^{(1)}$ and $A^{(2)}$ are performed in parallel, is BW-optimal, with a total-hop latency of $2T_L(A)$.

The above torus schedule has appeared in previous literature [123]. It can be generalized to generate schedules for Cartesian power of arbitrary topologies (see Appendix A.4.3).

Cartesian Product Expansion: One can also construct a schedule for the Cartesian product of distinct topologies. For example, an $a\times b\times c$ 3D torus is the Cartesian product of three rings with lengths a,b,c. Constructing this schedule requires BFB schedule generation technique, which we introduce in §3.6. If individual topologies have BW-optimal BFB schedules, as in the case of any torus, then the schedule generated for the Cartesian product is BW-optimal (Table 3.3).

3.5.4 Topology Finder

The goal of Topology Finder is to produce the best topologies and schedules for a target N and d. If we aim for asymptotic performance ($N \to \infty$ with fixed d), we want **the base topology to be as large as possible and the base schedule to be as optimal as possible** (Theorem 9). However, for a target N and d, only base topologies with certain sizes (e.g., divisors of N) and degrees can be expanded to the target. Thus, we keep a collection of known base topologies and their schedules (Table A.3). These topologies and schedules are highly optimized and cover a wide range of N and d.

Given base topologies, we perform a bottom-up search for the combinations of expansion techniques to reach the target N and d. We iteratively apply expansions to candidates. At intermediate sizes, we prune candidates with inferior performance and keep the best ones for further expansion. Because each expansion multiplies the topology size (Table 3.3), the number of expansions that can be applied before the size gets too large—and hence the number of possible combinations—is limited, making the search feasible.

While we expand the topologies, proved theorems (Table 3.3) allow us to predict the performance of expanded topologies. This is vital for the search because it is intractable to construct schedules for every topology and compare their performance. Using a simple formula for prediction enables us to quickly compare different topologies and prune inferior ones. We keep a Pareto frontier of topologies for each given N and d. A topology is inferior to another only if it is worse in both total-hop latency and BW runtime. Ultimately, the search finds all Pareto-efficient topologies for the target N and d. Depending on the testbed, we may convert unidirectional topologies to bidirectional ones (Appendix A.1.6). Then, we determine the best-performing topology for a given workload.

Table 3.4 shows the result for $N = 1024$ and $d = 4$. From top to bottom, the Pareto frontier exhibits an increasing T_L and a decreasing T_B, with the top and bottom being Moore and BW optimal, respectively. On the all-to-all side, the diameters of the topologies also follow the same trend as T_L because of $T_L \geq \alpha \cdot D(G)$ (Theorem 3). Table 3.4 also shows the allreduce and all-to-all times calculated based on specific α, M, B. Notably, the line graph of circulant graph $L^3(C(16,\{3,4\}))$ has both the lowest allreduce and all-to-all times, within 9% and 6% of the theoretical bounds. Table A.1 in appendix contains more results for $N = 32, 64, \ldots, 1024$.

While low-hop/diameter indicates high all-to-all throughput, other metrics like the average distance between nodes [84, 83] also play a role. Thus, despite having a lower diameter, $\Pi_{4,1024}$ underperforms

$L^3(C(16,\{3,4\}))$. Including other metrics makes the search process more complex and computationally expensive. In practice, T_L and $D(G)$ are feasible and accurate enough for predicting all-to-all throughput.

In DNN training experiments, we use one topology for the entire training due to the high reconfiguration latency of our target patch panel platform. We select the best option based on the distribution of collective sizes Ms, which depends on the communication strategy of the training framework [75]. With faster reconfiguration, one could change topology to optimize for different collective runs during training.

Our implementation runs under a minute for all $d = 2,4,8,16$ and N up to 2000. While this can be sped up, we find it acceptable for now, given that the search is performed once for all Ns and ds, and results can be saved for future use.

3.6 Breadth-First-Broadcast (BFB) Schedule

We now present a scalable algorithm for generating schedules for *generative topologies*, which are directly borrowed from graph theory, as well as for topologies obtained through Cartesian Product expansion—the only expansion technique that does not yield a schedule. State-of-the-art schedule generations (e.g., Blink [145], SCCL [21], and TACCL [128]) can scale only to a modest number of nodes because they involve NP-hard optimization. To ensure polynomial-time generation, we impose a *breadth-first* broadcast order from each node such that (1) data always travels along the shortest paths between source and destination nodes; (2) the schedule is structured as a series of comm steps, where each comm step is responsible for eagerly transmitting data to a set of nodes that is one additional hop away. Our BFB schedule generation technique does not guarantee optimality in an arbitrary topology, given these constraints prohibit the use of longer paths or delayed (non-eager) transmissions along paths, but these constraints enable polynomial-time generation.

Despite these constraints, BFB schedules guarantee the following: (1) The schedules have the lowest total-hop latency as all data is eagerly communicated over the shortest paths. (2) For Cartesian product topologies, BFB schedule generation yields a BW-optimal solution if the underlying product components admit BW-optimal BFB schedules (thus yielding optimal schedules for networks such as torus with arbitrary dimensions). (3) BFB schedules are also provably BW-optimal for many generative topologies with certain symmetries.

3.6.1 BFB Schedule Generation Linear Program

Intuition: Allgather is a simultaneous broadcast from every node in the topology. A BFB allgather schedule, as the name suggests, performs a *breadth-first* broadcast from every node. At each comm step, a node typically

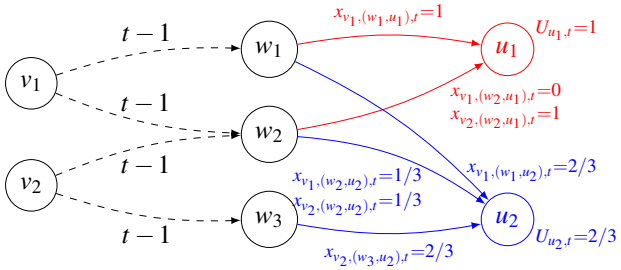


Figure 3.5: Example of BFB allgather schedule at comm step t. Here, $w_1, w_2 \in N^+_{t-1}(v_1)$ and $w_2, w_3 \in N^+_{t-1}(v_2)$. u_1, u_2 are at distance t from both v_1, v_2, so they both need to receive the data shards of v_1, v_2 in comm step t. Note that u_1 cannot get v_2's shard from w_3 because w_3 is not an in-neighbor of u_1. The figure also shows the solutions to LPs (3.1). The red and blue are independent LPs optimizing $U_{u_1,t}$ and $U_{u_2,t}$ respectively.

has the option to receive a chunk from multiple in-neighbors on the previous breadth-first frontier. To optimize BW performance, BFB uses a linear program to balance the traffic across the ingress links.

At each comm step t, a BFB schedule requires that for every node v, all nodes at a distance t from v, i.e., $N^+_t(v)$, receive v's data shard within the comm step. To achieve this, all nodes at distance $t-1$, i.e. $N^+_{t-1}(v)$, need to collectively multicast the data shard to nodes $N^+_t(v)$ in comm step t.

Given any v and $u \in N^+_t(v)$, u may have multiple in-neighbor ws in $N^+_{t-1}(v)$. All of them can provide v's data shard because they have received it in comm step $t-1$. Since the BW runtime of a comm step equals the transmission time of the most congested link, a question is **how to allocate the amount of data u receives from each w to balance the workload on links?** Figure 3.5 shows an example. Here, u_1 needs to get v_1's shard from w_1, w_2 and v_2's shard from w_2. The solution is simple: since u_1 can only get v_2's shard from w_2, we let w_1 send v_1's shard and w_2 send v_2's shard, achieving a perfectly balanced workload. For u_2, it is more complicated. We formulate such a problem as a linear program:

$$\begin{aligned} \text{minimize} \quad & U_{u,t} \\ \text{subject to} \quad & \sum_v x_{v,(w,u),t} \le U_{u,t}, \quad \forall w \in N^-(u) = N^-_1(u) \\ & \sum_w x_{v,(w,u),t} = 1, \quad \forall v \in N^-_t(u) \\ & 0 \le x_{v,(w,u),t} \le 1. \quad \forall w, v \end{aligned} \tag{3.1}$$

$x_{v,(w,u),t}$ is the proportion of v's shard that is sent from w to u and is defined for every v, w such that $w \in N^-(u)$ and $d(v,u) = d(v,w)+1 = t$. $U_{u,t}$ is the max workload among links to u, i.e., $(w_1,u_2),(w_2,u_2),(w_3,u_2)$ in the case of u_2. Minimizing $U_{u,t}$ is equivalent to minimizing $\frac{M/N}{B/d} \cdot U_{u,t}$, the max transmission time among links to u

at comm step t. The 1st and 2nd constraints ensure correct max workload and u receiving all data shards, respectively. Appendix A.5 gives the specific LP for u_2, and the solution is shown in blue in Figure 3.5. The workload is also balanced with each link sending $2/3$ shard and hence BW runtime $\frac{M/N}{B/d}\cdot\frac{2}{3}$.

SCCL [21] and TACCL [128] use NP-hard optimizations with discrete variables used to ensure each chunk is received before being sent. In contrast, we do not need discrete variables. A key observation from Figure 3.5 is that because w_1, w_2, w_3 all receive the entire shard of v_1 at comm step $t-1$, the $x_{v_1,(w_1,u_2),t}=2/3$ and $x_{v_1,(w_2,u_2),t}=1/3$ in the solution can be any portions of the data shard, as long as their union is the entire shard. Assuming $[0,1]$ is the entire shard of v_1, no matter the $2/3$ sent by w_1 to u_2 is $[0,\frac{2}{3}]$ or $[\frac{1}{3},1]$, the $1/3$ sent by w_2 can simply be $[\frac{2}{3},1]$ or $[0,\frac{1}{3}]$ accordingly. Thus, **we only need to decide the amount of data sent on each link, which are continuous variables**, enabling polynomial-time schedule generation. To obtain a complete schedule, one needs to solve an LP (3.1) for each $u\in V_G$ and $t\in\{1,\ldots,D(G)\}$. The BW runtime of the generated schedule is

$$T_B=\frac{M/N}{B/d}\sum_{t=1}^{D(G)}\max_{u\in V_G}U_{u,t}. \tag{3.2}$$

One could create an LP incorporating all $U_{u,t}$s and minimize T_B (3.2) *"globally"*. However, the result is equivalent to individually solving small LPs (3.1) for each u and t. This is because the LPs are independent of each other, e.g., the decisions made to minimize $U_{u_2,t}$ in Figure 3.5 do not affect $U_{u_1,t}$, and vice versa. The advantage of small LPs is that they can be solved in parallel. Due to the breadth-first nature of BFB, data always follows the shortest paths between source and destination. Thus, the number of comm steps of the BFB schedule always equals the graph diameter, i.e., $T_L=\alpha\cdot D(G)$, the lowest possible T_L given G.

Appendix A.5 analyzes the BFB schedule and includes modifications to generate **discrete chunked schedules** (§A.5.2) and schedules for **heterogeneous link bandwidths** (§A.5.3). Corollary 1.1 implies how to generate **reduce-scatter** schedules.

3.6.2 Generative Topologies

Generative topologies, unlike synthesized ones, are large graphs directly borrowed from graph theory. They are too large for manual or NP-hard schedule generation. Thus, we use the BFB linear program to generate schedules. Since a BFB schedule always has the lowest T_L for a topology, if it is also BW-optimal, then it is *the optimal schedule* for that topology. Generative topologies often have symmetries that allow us to prove optimality mathematically. Their low diameters are also ideal for all-to-all throughput.

Torus is a widely used topology in parallel computing systems. Our torus schedule generated by BFB is

theoretically optimal and represents a significant improvement over traditional torus schedules [123]. Given a $d_1 \times d_2 \times \ldots \times d_n$ torus, the traditional schedule, which performs parallel ring collectives on dimensions, only works (or is efficient) when dimensions are equal, i.e., $d_1 = d_2 = \ldots = d_n$, and has $T_L = \sum_i (d_i - 1)\alpha$. BFB torus schedule, however, is BW-optimal with any d_is and $T_L = \sum_i \lfloor d_i/2 \rfloor \alpha$. The BW optimality is due to torus being the *Cartesian product* of rings, each of which has a BW-optimal BFB schedule. BFB torus opens up many more construction possibilities since d_is can be any combination.

Generalized Kautz Graph (§A.6.2) and **Circulant Graph** (§A.6.4) are a pair of versatile graphs in our toolbox. The former can be constructed for any N and d, while the latter can be constructed for any N and even-value d. Furthermore, the BFB schedule of the former is at most one α away from Moore optimality, making it the topology with the lowest T_L, while the latter always has a BW-optimal BFB schedule. Thus, they can fill gaps in N and d that expansion techniques fail to cover (e.g., prime N) or provide good candidates.

Besides the aforementioned ones, the following graphs also have optimal schedules by BFB. **Distance-Regular Graph** (§A.6.3) is a family of large highly-symmetric graphs that are both BW-optimal and low-hop at the same time. The **Twisted Torus** [30] used by TPU v4 [62] is also computationally verified to be BW-optimal for at least $N \leq 10^4$. A **BFB Ring Schedule** with half the T_L of traditional one is shown in §A.6.1.

3.7 Schedule Compilation

We implemented two compilers for lowering communication schedules to both GPU and CPU clusters, given the significance of collective communication for both ML and HPC workloads. We lowered over 1K schedules for various topologies and configurations. The compilers enable us to evaluate the performance of our topologies and schedules on hardware and to validate our mathematical model.

For GPUs, our compiler lowers a mathematically defined schedule to an XML file that can be executed by the MSCCL runtime [95]. MSCCL is an open-source collective communication library that extends NCCL [103] with an interpreter providing the ability to program custom schedules. Communication schedules are defined in XML as instructions (send/receive/reduce/copy) for each GPU threadblock. Our compiler also performs certain optimizations, such as consolidating non-contiguous sends using a scratch buffer and evenly distributing the computational workload across threadblocks.

For CPU-based supercomputers, we use Intel oneCCL [55] + libfabric [77] to execute schedules. We extended oneCCL with an interpreter that executes XMLs. The mathematical schedules are lowered into instructions (send/recv/reduce/copy/sync) for CPUs in an XML file and then executed.

3.8 Evaluation

We present performance evaluation results on a 12-node direct-connect optical GPU testbed and a supercomputing CPU torus cluster with up to 54 nodes. We also present analytical and simulation results at larger scales.

Collective Communication: On the 12-node testbed, our topologies consistently outperform baselines in allreduce, reduce-scatter, and allgather (§3.8.3, Fig 3.6, Fig A.1). Analytical model shows order-of-magnitude improvements in allreduce and all-to-all performance at larger scales (Fig 3.7).

DNN Training: In data-parallel training experiments, our topologies achieve the best performance for both small models and GPT-2 [117] (§3.8.4, Fig 3.8). In simulated large-scale training, our topologies demonstrate order-of-magnitude improvements in all-to-all involved expert-parallel training (Fig 3.9).

Schedule Generation: While generating optimal schedules, BFB is orders of magnitude faster and more scalable than SCCL [21] and TACCL [128] (§3.8.5, Tab 3.6 & Fig 3.10). On supercomputing torus clusters, BFB schedules outperform traditional scheduling [123], SCCL, and TACCL (§3.8.5, Fig 3.11).

Finally, we also conducted experiments on our testbed to compare BFB against widely adopted solutions for switch networks (e.g., NCCL and recursive halving & doubling) (§A.1.1) and to validate the α-β cost model (§A.1.2).

3.8.1 Direct-Connect Optical Testbed

Our testbed consists of 12 servers, each with an NVIDIA A100-PCIe-40GB GPU [102] and an HPE Ethernet Adapter, configured as 4x25Gbps breakout interfaces. The NICs are directly connected via a Telescent optical patch panel [137]. Our testbed can realize topologies by reconfiguring the patch panel. We limit our evaluation to bidirectional topologies due to limitations in our testbed (discussed in Appendix A.1.6).

3.8.2 Experiment Setup

Baselines: We evaluate against the two baselines at $d = 4$: (1) *ShiftedRing*, which improves upon NCCL ring [103], is used by TopoOpt [152] for data-parallel training. The topology is a superposition of two bidirectional rings, each allreducing half of the data. (2) *Double Binary Tree* (DBT), implemented in NCCL [58], uses the topology and schedule from [124].

Methodology: We use the MSCCL runtime [95] to evaluate the topologies and schedules. We sweep through runtime parameters, such as the protocol (Simple or LL), number of channels (1, 2, 4, or 8), degrees of

N	**Topology**	T_L
5	Complete Graph: K_5	2α
6	Degree Expansion of Complete graph: $K_3 * 2$	4α
7	Circulant Graph: $C(7,\{2,3\})$	4α
8	Complete Bipartite Graph: $K_{4,4}$	4α
9	Hamming Graph: $H(2,3)$	4α
10	Degree Expansion of BFB augmented Bidirectional Ring: BiRing$(2,5) * 2$	4α
11	Circulant Graph: $C(11,\{2,3\})$	4α
12	Circulant Graph: $C(12,\{2,3\})$	4α

Table 3.5: OurBestTopo at $d=4$ generated by topology finder (§3.5.4). All topologies listed above are BW-optimal.

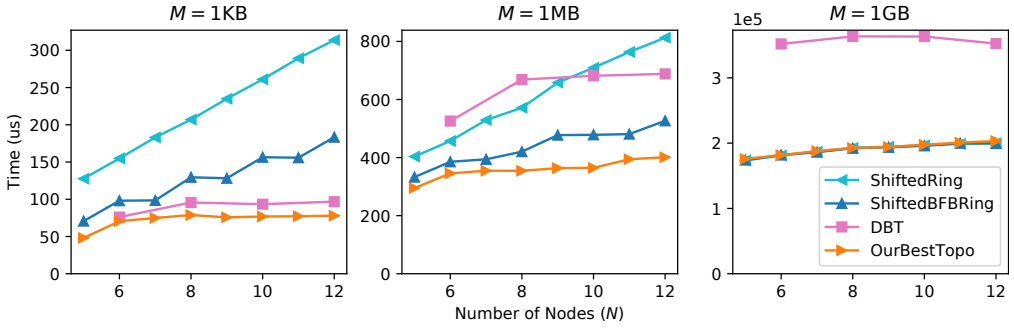


Figure 3.6: Allreduce experiment results on testbed at $M=1\text{KB}, 1\text{MB}, 1\text{GB}$. "OurBestTopo" topologies are listed in Table 3.5. Reduce-scatter and allgather results are in Appendix Figure A.1.

pipelining for the DBT baseline, etc., and choose the best-performing schedule for each data size. For DNN training, we run our schedules in PyTorch through MSCCL as the backend.

3.8.3 Collective Communication Evaluation

Figure 3.6 shows allreduce results for varying topology sizes N and data sizes M. Table 3.5 shows the topologies generated by our topology finder (§3.5.4). We also add *ShiftedBFBRing*, which is ShiftedRing topology but with our BFB generated schedule. We observe that in the small data regime ($M=1\text{KB}$), our topology beats ShiftedRing by a significant margin ($\sim$75% at $N=12$) and also outperforms DBT ($\sim$20% at $N=8,10,12$). Our ShiftedBFBRing beats ShiftedRing ($\sim$40% at $N=12$) despite using the same topology. At small data sizes, the runtime is dominated by total-hop latency T_L, and hence, we can significantly outperform ShiftedRing, which has linear instead of logarithmic T_L growth with respect to N.

In the large data regime ($M=1\text{GB}$), our topology beats DBT by a significant margin ($\sim$50% lower at

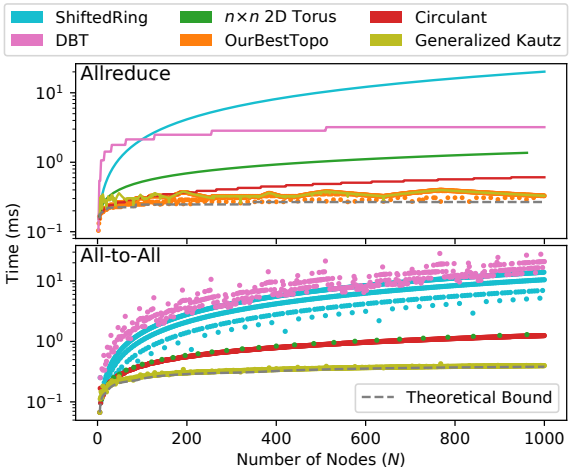
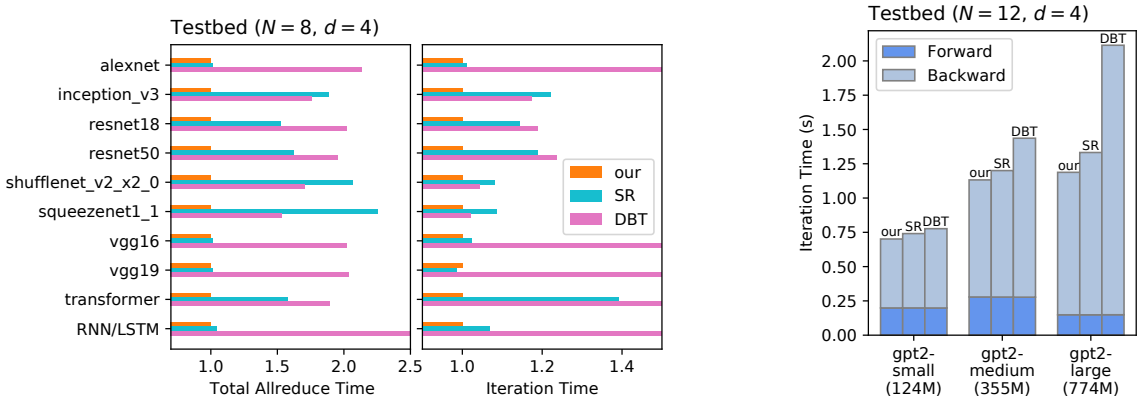


Figure 3.7: Comparing theoretical allreduce and all-to-all runtimes analytically at large N for $d=4$, $\alpha=10$us, and $M/B=$ 1MB/100Gbps. The all-to-all times are computed via multi-commodity flow (Appendix A.1.5) with each node having 1MB of data to send (i.e., sending $1/N$ MB to each node).

$N=8,10,12$) and matches the performance of ShiftedRing. At large data sizes, the runtime is dominated by BW runtime T_B. Since the ShiftedRing is BW-optimal, we can only match its performance. Due to the influence of both total-hop latency and BW runtime at intermediate data sizes ($M=1$MB), our topology outperforms ShiftedRing ($\sim$50% at $N=12$) and DBT ($\sim$45% at $N=8,10,12$) in this regime. Our ShiftedBFBRing also outperforms ShiftedRing ($\sim$35% at $N=12$). Note that although our gains over ShiftedRing diminish as M grows, future increases in hardware bandwidth will enhance gains at large M due to T_L playing a more significant role. Appendix Figure A.1 shows the reduce-scatter and allgather results, which demonstrate trends and conclusions similar to those in Figure 3.6.

Figure 3.7 shows the allreduce and all-to-all runtime comparison for large N based on our analytical model. Topologies generated by our topology finder perform orders of magnitude faster in both allreduce and all-to-all. In allreduce, our best topologies outperform ShiftedRing and DBT by 56$\times$ and 10$\times$, respectively, near $N=1000$, due to the former's linear growth in T_L and the latter's poor BW performance. When compared against 2D torus, our topologies also achieve 4$\times$ better allreduce performance near $N=1000$ (see §A.1.3 for a detailed analysis of our topologies at large N for different $\alpha, M/B$). As for all-to-all, we compare baseline topologies against our lowest-diameter topology, generalized Kautz, and our highest-diameter topology, circulant, from our Pareto-frontier for any N and d. These two represent our best and worst all-to-all topologies, respectively, while also serving as the worst and best BW-efficient allreduce topologies. Nevertheless, circulant



(a) Small Model testbed training results (normalized by our $K_{4,4}$)

(b) GPT2 testbed training results

Figure 3.8: Testbed data-parallel training results with different topologies. We compare our topologies (Table 3.5) against ShiftedRings (SR) and double binary trees (DBT) at 8- and 12-node scale. (a) shows results of training small models on 8 A100 GPUs of our testbed, all using batch size 64. The total allreduce time is the sum of the allreduce times for all layers in the model. (b) shows results of training GPT-2 with 12 A100 GPUs. The per-GPU batch sizes are selected to max out the 40GB GPU memory, with the small, medium, and large models having per-GPU batch sizes of 8, 4, and 1, respectively.

still outperforms all baselines in all-to-all: 9× and 14× better than ShiftdRing and DBT, respectively, on average from $N = 900$ to 1000. It is barely matched by the 2D torus, which is limited to the square number Ns. Our lowest-diameter topology, generalized Kautz, outperforms ShiftedRing and DBT by 28× and 42×, respectively, and is within 5.2% of the theoretical bound from $N = 900$ to 1000.

3.8.4 DNN Training Evaluation

We compare our topologies against ShiftedRings and double binary trees in DNN training. On our small-scale testbed, we demonstrate improvements in data-parallel training across various small models and also GPT-2 [117]. For full-scale LLM training involving up to 1024 nodes and all-to-all communications, we simulate expert-parallel training of a Mixture of Experts (MoE) model to show our improvements at scale.

Testbed Training: We run PyTorch Distributed Data Parallel (DDP) [75] training experiments on our testbed. Figure 3.8 shows the results of training both small DNN models and GPT-2. We compare our topologies (from Table 3.5) against ShiftedRings and DBT. In training small models (Figure 3.8a), our topology improves total allreduce time by 30% and 50% on average against ShiftedRing and DBT, respectively. With optimizations such as compute-communication overlap, our topology still secures a 10% and 25% average improvement in iteration time over the baselines. In GPT-2 training (Figure 3.8b), despite the limited scale of our testbed, our topology enhances iteration time by 7% and 25% on average compared to

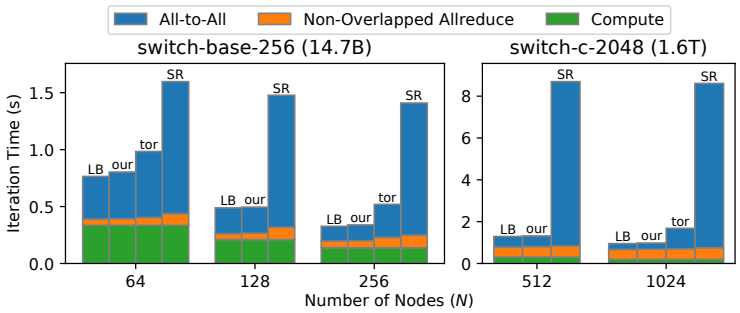


Figure 3.9: Simulated expert-parallel training of Switch Transformers across various topologies of different sizes. The simulation is conducted assuming $\alpha = 10$us, $B = 100$Gbps. All-to-all time is computed via multi-commodity flow (Appendix A.1.5). We detailed our setup in Appendix A.1.4.

ShiftedRing and DBT, respectively.

Large-Scale Simulation: While improvements over ShiftedRing and DBT have been shown in testbed training, full-scale LLM training is performed on much larger clusters. In Figure 3.9, we simulate expert-parallel training of Switch Transformers [40] on a much larger scale with parameter sizes up to 1.6 trillion. We collect execution timestamps from one A100 GPU to derive the compute times for each layer. Communication times are then added to simulate training, ensuring compute-communication overlap/dependency. Appendix §A.1.4 provides further details of the simulation.

Expert-parallel training involves not only data-parallel allreduce for non-expert layers but also all-to-all communications to transfer tokens to and from the routed experts, which are in the critical path of compute [40, 119, 70, 73]. In Figure 3.9, we break down the iteration time into compute time, non-overlapped allreduce time, and all-to-all time for a better understanding of performance. As previously analyzed in Figure 3.7, ShiftedRings (SR) exhibit all-to-all performance that is order-of-magnitude worse than our topologies. At 256-node training of 14.7B MoE model, ShiftedRing has 8× greater total all-to-all time, resulting in 4× longer iteration time compared to our topology. We also include 2D torus (tor) for comparison due to its relatively better all-to-all performance. However, it still has all-to-all and iteration times that are 2× and 1.5× greater, respectively, than our topology. The disparity is even larger at 1024-node training of 1.6T MoE model, where ShiftedRing and 2D torus show all-to-all times that are 27× and 3.3× greater, and iteration times that are 9× and 1.7× longer, respectively. At this scale, ShiftedRing and 2D torus spend 91% and 58% of iteration time on all-to-all communications, while our topology only spends 30%. We omit DBT in Figure 3.9 due to its significantly worse performance (∼2× of ShiftedRing). Due to high performance in

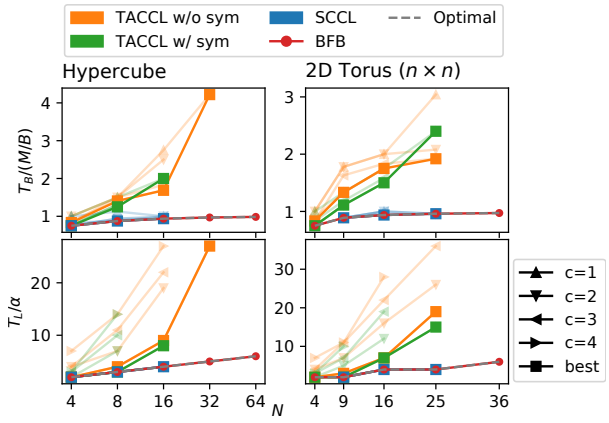


Figure 3.10: Comparing theoretical performances of schedules from Table 3.6. We show both T_L and T_B of the schedules, along with the theoretical optimal. For SCCL and TACCL, the solid lines show the best results from parameter sweeps. The inferior ones are dimmed.

both allreduce and all-to-all, our topologies consistently remain within 5% of the theoretical lower bound (LB) for iteration time.

Since large models involve large allreduce sizes and both torus and ShiftedRing are BW-optimal, the allreduce performance is similar across these topologies. For a broader spectrum of all-to-all efficient low-hop topologies like expander graphs, the lack of efficient allreduce schedules prior to our work has prevented their use in allreduce-involved training.

3.8.5 BFB Schedule Evaluation

We evaluate schedule generation from two aspects: schedule generation runtime and the performance of generated schedules. In §3.8.5, we compare BFB with state-of-the-art schedule generations: SCCL [21] and TACCL [128], in both generation runtime and theoretical schedule performance. In §3.8.5, we compare, on supercomputing torus clusters, the performance of torus schedules generated by BFB, traditional torus scheduling [123], SCCL, and TACCL.

Schedule Generation

In schedule generation, SCCL and TACCL are the closest in spirit to BFB schedule generation. Table 3.6 shows the runtime comparison between SCCL, TACCL, and BFB when generating allgather schedules for hypercube and 2D torus. Both SCCL and TACCL use NP-hard optimization to generate schedules. SCCL, which uses an SMT solver, fails to generate a schedule within 10^4 seconds beyond $N = 30$. TACCL formulates

N	SCCL				TACCL w/o Symmetry				TACCL w/ Symmetry				BFB
	$c=1$	$c=2$	$c=3$	$c=4$	$c=1$	$c=2$	$c=3$	$c=4$	$c=1$	$c=2$	$c=3$	$c=4$	
Hypercube													
4	0.59	0.64	0.68	0.72	0.89	0.50	0.83	0.75	0.62	0.51	0.71	0.60	<0.01
8	0.86	1.22	1.86	2.48	96.9	807	63.2	1800	7.97	645	7.39	1801	<0.01
16	21.4	48.4	130	573	1801	1801	1801	1802	1801	n/a	n/a	n/a	<0.01
32	$>10^4$	$>10^4$	$>10^4$	$>10^4$	1802	n/a	n/a	n/a	n/a	n/a	n/a	n/a	0.03
64	$>10^4$	$>10^4$	$>10^4$	$>10^4$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	0.17
1024	$>10^4$	$>10^4$	$>10^4$	$>10^4$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	52.7
2D Torus ($n \times n$)													
4	0.61	0.63	0.67	0.76	0.68	0.50	0.82	0.72	0.45	0.51	0.76	0.64	<0.01
9	1.00	1.51	2.22	3.44	1801	189	67.8	262	88.6	71.1	67.8	105	<0.01
16	17.5	60	131	603	1801	1801	1801	1802	1801	1801	1801	n/a	<0.01
25	3286	5641	$>10^4$	$>10^4$	1802	1802	1803	n/a	1802	n/a	n/a	n/a	0.01
36	$>10^4$	$>10^4$	$>10^4$	$>10^4$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	0.03
2500	$>10^4$	$>10^4$	$>10^4$	$>10^4$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	61.1

Table 3.6: Comparing allgather schedule generation runtimes (in seconds) of SCCL, TACCL, and BFB. The setup of SCCL is to generate schedules with the least number of comm steps. Both SCCL and TACCL were run with chunks=1,2,3,4 (number of chunks per shard), and TACCL was run w/ and w/o manually set topology symmetry. "n/a" indicates where TACCL reports an error due to failure to generate a solution within its 1800s time limit for MILP solver.

the scheduling problem as a mixed integer linear program (MILP). It sets an 1800s time limit for its MILP solver, after which it will return the best solution found up to that point. However, for larger topologies, TACCL's solver fails to find a solution within the time limit, resulting in an error. In comparison, BFB schedule generation is faster by orders of magnitude due to its polynomial-time generation.

In terms of theoretical schedule performance, Figure 3.10 compares the total-hop latency and BW runtime of generated schedules. Given a topology, SCCL and TACCL need to perform a sweep across parameters such as the number of chunks and symmetry. They have to generate schedules for different parameter sets to identify the high-performance ones, unlike BFB, which requires no parameter. In Figure 3.10, the schedules of SCCL and BFB can both achieve exact optimality, but TACCL's have significantly worse performance, especially at large Ns. SCCL is uniquely capable of generating all Pareto-efficient schedules. However, due to the prohibitive runtime of parameter sweep, SCCL can only do so for very small Ns.

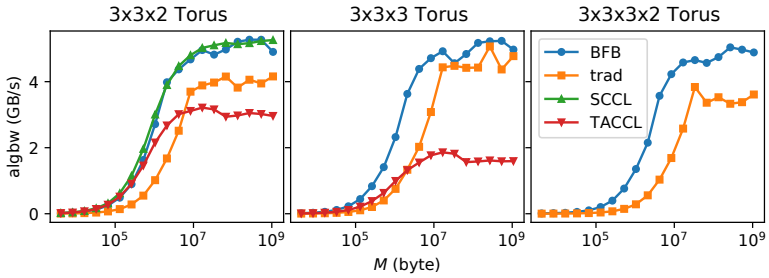


Figure 3.11: Comparing allreduce performances of torus schedules generated by BFB, traditional torus scheduling [123], SCCL, and TACCL on Frontera [134] supercomputer. The y-axis is algorithmic bandwidth (algbw), computed as M divided by end-to-end runtime. SCCL fails to generate a schedule for 3×3×3 and 3×3×3×2 tori, and TACCL fails to generate a schedule for 3×3×3×2 torus within the time limits.

Supercomputing Allreduce Experiments

In the supercomputing setting, we run torus schedules generated by BFB, traditional torus scheduling [123], SCCL, and TACCL on Frontera [134] supercomputer at the Texas Advanced Computing Center (TACC) [138]. The cluster consists of 396 nodes in a 6D torus direct-connect topology. Each node is equipped with an Intel Xeon Platinum 8280 CPU and a Rockport NC1225 network card, capable of delivering 25 Gbps per link, with degree 12. However, the total BW of a single node may be bottlenecked by the 100 Gbps host BW of PCIe Gen3 x16. Finally, the schedules are lowered and run using Intel oneCCL [55] + libfabric [77].

We run schedules on two types of sub-torus within the cluster: equal-dimension (3×3×3) and unequal-dimension (3×3×2 & 3×3×3×2). As shown in Figure 3.11, BFB schedules achieve the highest performance in all settings. As mentioned in §3.6.2, the traditional torus schedule can only achieve high BW performance in tori with equal dimensions. At large M, it matches BFB's performance in 3×3×3 torus but significantly underperforms in 3×3×2 and 3×3×3×2, where BFB has 29% and 42% higher algbw on average for $M \geq 100$MB. At small to intermediate M (< 100MB), BFB outperforms traditional schedules by 3.1× on average in all settings due to its 2× lower in total-hop latency and higher BW performance.

As for SCCL and TACCL, we adhere to the same time limits and parameter sweeps as in §3.8.5 and select the best result at each M from all parameter sets. In 3×3×2 torus, SCCL is able to generate an optimal schedule, matching BFB's performance across all M. However, it fails to generate a schedule within 10^4 seconds for other larger tori. TACCL can only generate schedules in 3×3×2 and 3×3×3, and its schedules underperform BFB's by a large margin. One additional observation is that the algbw of BFB at large M hardly changes from 18-node (3×3×2) to 54-node (3×3×3×2) torus. This can be explained by the fact that BFB

schedules have theoretically achieved allreduce BW optimality ($\frac{2M}{B} \cdot \frac{N-1}{N}$), which remains nearly constant as N increases.

3.9 Concluding Remarks

Collective communications are critical to both ML training and HPC workloads. Current solutions often rely solely on existing topologies and schedules, resulting in high total-hop latency, bandwidth inefficiency, or low all-to-all throughput. We presented a general, highly scalable, and automated algorithmic framework for optimizing topology and schedule generation for collectives by leveraging scalable graph-theoretic approaches. Our evaluation demonstrates significant performance gains across multiple testbeds and large-scale simulations in both standalone collective communications and end-to-end ML training.

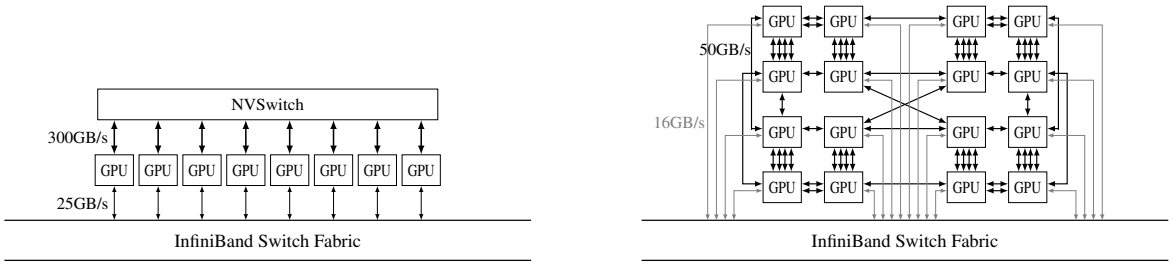
Chapter 4

ForestColl: Throughput-Optimal Collective Communications on Heterogeneous Network Fabrics

4.1 Introduction

Collective communications have become a cornerstone of distributed machine learning training [45, 127, 131]. As large language models (LLMs) scale to hundreds of billions of parameters [82, 109, 140, 97], their training demands immense volumes of collective communication traffic, creating a performance bottleneck [151, 163, 113, 131, 152, 60, 43, 115]. Operational insights from AI hyperscalers (e.g., Meta [43], Alibaba [115], AWS [42]) highlight that LLM training traffic, characterized by *bursty elephant flows* capable of saturating NIC line rate, is predominantly **throughput-bound**. As a result, today's ML hardware providers have focused on enhancing inter-accelerator network speed [105, 106, 107, 4, 5].

A key observation is that *today's ML network topologies are becoming **heterogeneous** within individual networks and highly **diverse** across different hardware platforms.* Because scaling high-speed networks



(a) NVIDIA DGX A100 (b) AMD MI250

Figure 4.1: Network Topologies of NVIDIA DGX A100 and AMD MI250. The PCIe switches and IB NICs are omitted for simplicity.

(a) Ring (b) ForestColl

Figure 4.2: Example of ring's suboptimality. (a) and (b) show two broadcast paths from one of the GPUs in ring allgather and ForestColl, respectively. Note that in (a), ring's path crosses IB switch twice, whereas the path in (b) crosses only once. In allgather, each GPU broadcasts a distinct shard of data to all other GPUs. When all GPUs broadcast simultaneously, ring allgather generates nearly twice the traffic across IB compared to (b), making it suboptimal due to IB's much lower bandwidth compared to NVSwitch.

homogeneously (i.e., uniformly across the fabric) is both technically challenging [48, 7, 52] and prohibitively costly [152, 150], hardware providers adopt *heterogeneous* networks, which typically consist of separate high-speed scale-up networks within multi-GPU boxes and lower-speed scale-out networks between boxes. Figure 4.1 shows the network topologies of NVIDIA DGX A100 [106] and AMD MI250 [4]. In both topologies, the intra-box network is an order of magnitude faster than the inter-box one—300GB/s vs 25GB/s per GPU in DGX A100 and 350GB/s vs 16GB/s per GPU in MI250. Moreover, different hardware platforms feature highly *diverse* network designs. DGX A100 uses NVSwitch for inter-GPU traffic within a box, while MI250 relies on direct connections between GPUs. Although both platforms use InfiniBand for inter-box traffic, the IB switches can also be configured in various topologies, such as fat-tree [3] or rail networks [150, 89].

Given the heterogeneity and diversity, traditional *static* collective algorithms (e.g., ring, recursive halving/-doubling), assuming a simple homogeneous network, are ill-suited for today's ML networks. The mismatch between the assumed homogeneity and the actual heterogeneity leads to network imbalance, congestion, and, ultimately, poor throughput. For instance, ring allreduce [45] assumes a flat network where each node sends data to the next node at equal bandwidth. When applied to multi-box DGX A100, however, ring allreduce is bottlenecked by the slower inter-box bandwidth and underutilizes the faster intra-box bandwidth (see example in Figure 4.2). Further, existing static algorithms (e.g., ring, recursive halving/doubling, Bruck algorithm, BlueConnect) [116, 28] assume that communicating with a single peer can saturate a node's bandwidth. Yet,

today's ML networks often feature multi-ported nodes with connections to multiple GPUs/switches, which these static algorithms cannot fully exploit.

To address the heterogeneity and diversity of ML network topologies, recent works (e.g., SCCL [21], TACCL [128], TE-CCL [81], TACOS [156], BFB [160], Blink [145], MultiTree [53], TTO [69], SyCCL [22]) seek to *dynamically generate a collective communication schedule tailored to a given topology.* However, these schedule generation methods still face limitations. They either rely on NP-hard optimizations (SCCL, TACCL, TE-CCL, SyCCL), use suboptimal greedy algorithms (MultiTree, TACOS), support limited collective operations (Blink), or work only with specific topologies (TTO, BFB). Moreover, network switches pose challenges for all methods. Unlike direct GPU connections, switches support flexible traffic patterns that require specialized modeling in schedule generation. Existing methods either ignore switches (SCCL, TTO, BFB) or rely on suboptimal solutions (Blink, MultiTree, TACCL, TE-CCL, TACOS, SyCCL).

We argue that an ideal schedule generation method should achieve a **triathlon**: *optimality* to produce throughput-efficient schedules, *generality* to support heterogeneous and diverse topologies, and *scalability* to handle large topologies. In this work, we present ForestColl, a triathlete method capable of generating collective communication schedules with theoretically **optimal throughput** for any **heterogeneous topology** in **polynomial time**. Our approach leverages *spanning tree packing* from graph theory for schedule generation. However, applying spanning tree packing directly is hindered by several key technical challenges, including deriving the throughput optimality, limiting the number of trees, and supporting traffic patterns of switches. ForestColl overcomes these challenges through algorithmic innovations, introducing the following key novelties:

1. **Optimality:** To the best of our knowledge, ForestColl is the first work capable of deriving the optimal throughput of any topology (§4.4 & 4.5.2) and generating optimal schedules for *reduce-scatter*, *allgather*, and *allreduce*.

2. **Generality:** ForestColl introduces a novel transformation for switch topologies that supports their flexible traffic patterns while preserving optimal throughput (§4.5.3). ForestColl also leverages in-network multicast/aggregation capabilities when supported by switches (§4.5.6).

3. **Scalability:** Every part of ForestColl runs in polynomial time (Appendix B.6), scalable to large topologies (§4.6.5).

We evaluated ForestColl on AMD MI250 and NVIDIA DGX A100 platforms, as well as on a large-scale

128-GPU DGX H100 cluster. At 1GB data size, ForestColl achieves 16%∼61% higher throughput than state-of-the-art schedule generation methods on AMD and NVIDIA platforms, and 14%∼32% higher throughput than NCCL on the 128-GPU cluster. ForestColl also reduces iteration time in PyTorch FSDP training by 20% on 70B+ LLMs. In schedule generation, ForestColl is orders of magnitude faster than competing methods while maintaining throughput optimality.

4.2 Background & Related Work

Based on the generated schedules, current schedule generation methods can be categorized into step schedules (SCCL, TACCL, TE-CCL, TACOS, BFB, SyCCL) and tree-flow schedules (Blink, MultiTree, TTO). **Step schedules** specify the data exchanged between GPUs at each step, with the entire network progressing through steps in sync. **Tree-flow schedules** let data flow fluidly through a set of spanning trees, either broadcasting from or reducing to the roots. In this section, we examine the three goals of the *triathlon* and explain why existing methods fail to achieve all three simultaneously.

Scalability: Optimizing collective communication is computationally challenging. Unlike point-to-point traffic, where data dependencies can be enforced by flow conservation, collective operations involve one-to-many multicast and many-to-one aggregation, rendering flow conservation inapplicable. Step schedules like SCCL, TACCL, TE-CCL, and SyCCL choose to track data dependencies in discrete chunks and formulate the scheduling problem as NP-hard SMT or MILP. As a result, they struggle with even modestly sized topologies (§4.6.5). In contrast, tree-flow schedules scale better as dependencies are naturally maintained through trees. However, existing methods fail to ensure optimality, as we discuss next.

Optimality: The performance of collective operations depends on two key metrics: *throughput* and *latency*. Latency, the fixed time cost incurred by send/recv hops, is critical for small data transfers. However, as data size grows, throughput becomes the dominant factor, as the bandwidth-bound transmission cost quickly outweighs the fixed latency. Step schedules are convenient for optimizing latency, as the number of steps directly corresponds to the number of hops. However, optimizing throughput with step schedules is challenging, requiring careful minimization of congestion across possibly heterogeneous links *within* each step while maintaining data dependencies *across* steps. Tree-flow schedules are better suited for throughput optimization, reducing the problem to minimizing congestion/overlap between trees. The remaining challenge lies in constructing optimal trees, where existing methods—such as Blink's approximate tree packing and MultiTree's greedy construction—are inherently suboptimal.

Generality: A general schedule generation should support topologies with (i) diverse graph structures, (ii) links with varying bandwidths,[1] and (iii) switches. While most existing works address (i), support for (ii, iii) remains limited. Heterogeneous links present a challenge for step schedules, as they require synchronized step execution across the entire network. Switches add further complexity as they do not produce/consume data, and many cannot multicast/aggregate, necessitating an operating model different from GPUs.

Related Work: Apart from the scalability consideration discussed earlier, none of the existing methods simultaneously achieves both optimality and generality. SCCL can achieve optimality for a given number of data chunks, but the optimal chunking is unknown, and it does not support switch topologies. TACCL, TE-CCL, and SyCCL rely on heuristic tuning (e.g., sketches in TACCL and SyCCL, the reward-based objective in TE-CCL) that does not guarantee optimality. Beyond scalability issues, they are tuned and evaluated on only limited scales and topology types, offering no guarantees for broader settings (§4.6.5). TACOS and MultiTree employ greedy approaches to assign traffic to links, which also do not ensure optimality. BFB and TTO provide optimality but only for specific types of non-switch topologies. Blink's tree packing constructs all trees rooted at a single node, lacking support for allgather and reduce-scatter. The single root becomes a bottleneck in allreduce, as Blink performs allreduce via reduce+broadcast. Its solution for multi-box switch settings is ad hoc and unrelated to its tree packing. Table B.1 in the appendix summarizes the comparison of related work.

Other efforts to accelerate ML training communications, such as network infra optimizations, hybrid parallelism, and comp-comm overlap, are orthogonal and complementary to ForestColl. In particular, ForestColl's communication acceleration reduces the need for compute and memory trade-offs to hide communication costs in hybrid parallelism and comp-comm overlap. Detailed discussion is in Appendix B.2.

4.3 Overview of ForestColl

We now introduce ForestColl. To ensure throughput optimality for throughput-bound LLM training [43, 115, 42, 76], ForestColl adopts tree-flow schedules. Unlike prior tree-flow methods, ForestColl leverages spanning tree packing to construct a "forest" of spanning trees—an equal number of trees rooted at each node—that achieves the triathlon for multi-root collectives. To accomplish this, ForestColl introduces several key techniques, including a method to compute the optimal throughput of any given topology and a topology transformation to support throughput optimality in switch networks.

[1] Support for heterogeneous GPUs is captured by varying link bandwidths.

(a) Direct Tree Construction

(b) Desired "Tree" Construction

Figure 4.3: Example of spanning tree construction on a switch topology. (a) shows a spanning tree constructed directly on the input switch topology, resulting in two issues: (1) the construction assumes switches are capable of in-network multicast/aggregation, which is not always supported; (2) the tree unnecessarily spans the bottom NVSwitch, which does not consume data. (b) shows the desired "tree" construction, as provided by ForestColl, which is a spanning tree among GPU nodes only. Switches are not part of the tree but serve only to provide connections between the GPUs.

Spanning Tree Packing is a well-studied topic in graph theory that focuses on determining the maximum number of spanning trees that can be constructed in a graph given edge capacities [10, 136, 37, 13, 126]. In ForestColl's tree-flow schedules, each tree occupies and utilizes an equal share of bandwidth, making spanning tree packing useful for constructing trees that make optimal use of the available link bandwidths. While efficient and optimal tree-packing algorithms have been proposed in graph theory, they are not directly applicable to our schedule generation, leaving several challenges.

Technical Challenges: Traditional spanning tree packing defines link capacity as the number of trees a link can support. As a result, directly using link bandwidth as capacity in our case would require constructing hundreds or even thousands of trees, as bandwidth values are typically large. This presents a scalability challenge: *How to achieve throughput optimality with a small number of trees?* To overcome this challenge, we choose to first determine the optimal bandwidth each tree occupies and scale link capacities accordingly before applying tree packing. This, however, raises a broader, previously unsolved optimality challenge: *How can the optimal collective communication throughput of a topology be determined?* Finally, as shown in Figure 4.3, the traffic patterns of network switches render the desired schedule no longer properly defined by spanning trees, raising a generality challenge: *How to apply spanning tree packing while supporting the unique traffic patterns of switches?*

Overview: ForestColl effectively solves the challenges. We studied the throughput optimality of a

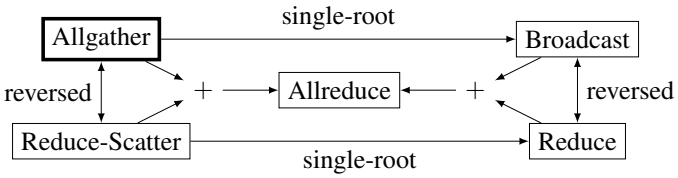


Figure 4.4: Relationships between collective operations. Reduce and reduce-scatter can be constructed by reversing the communications of broadcast and allgather [23]. Reduce and broadcast are single-root versions of reduce-scatter and allgather. Finally, allreduce can be performed via reduce-scatter plus allgather or reduce plus broadcast. While this paper focuses on allgather, the method applies to other operations.

topology and found that it is determined by the *throughput bottleneck cut*—the network cut with the highest ratio of minimal required traffic to available bandwidth (§4.4). In ForestColl, we combine binary search and network flow to compute this optimality, determining the optimal number of trees and the bandwidth each tree occupies (§4.5.2). Once edge capacities are scaled accordingly, the tree packing algorithm can be applied to construct the trees (§4.5.4). For switches, we devised a way to transform a switch topology into a switch-free logical topology before applying tree packing (§4.5.3). Unlike TACCL and TACOS, which remove switches and connect nodes in preset patterns that overlook the performance impact of losing switches' all-to-all connectivity, ForestColl's transformation ensures no compromise in performance. Together, ForestColl achieves the triathlon of ideal schedule generation.

Limitations: ForestColl does not solve all problems in collective communication. First, efficient communication requires not only effective scheduling but also optimized implementations for different hardware. ForestColl derives optimal schedules to guide such implementations. Second, ForestColl prioritizes throughput over latency. Latency is better optimized at the implementation level (e.g., through low-latency protocols and CUDA kernels), and low-latency scheduling has been extensively studied in step schedules. Third, ForestColl is designed as an offline schedule generator that runs once per topology, rather than real-time online scheduling in subseconds. The generation time is trivial when amortized over cluster setup and model training. Finally, ForestColl does not exploit topology symmetry to speedup scheduling. While it can generate symmetric schedules, preserving both symmetry and optimality does not yield performance benefits. Since the current algorithm has no scalability issues, leveraging topology symmetry is left for future work.

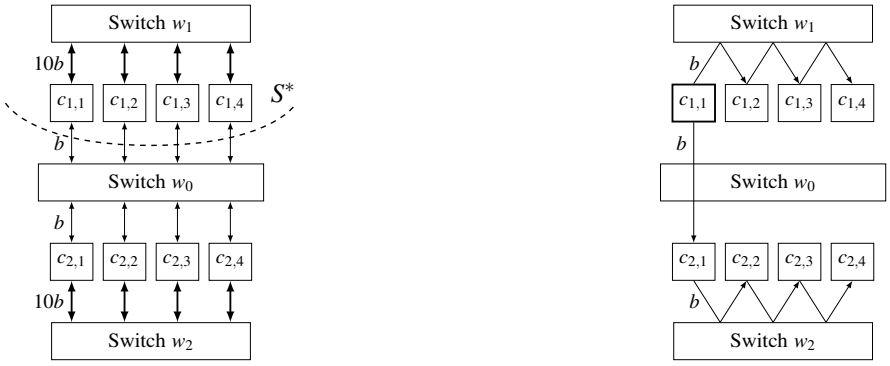


Figure 4.5: Example of Spanning Out-Tree. (a) shows a 2-box 8-compute-node switch topology along with the throughput bottleneck cut. The intra-box connections (thick lines) have 10x the bandwidth of inter-box ones (thin lines). (b) shows one example of ForestColl's spanning out-trees rooted at $c_{1,1}$.

4.4 Throughput Optimality for Collectives

Collective operations can be classified into *aggregation only* (e.g., reduce, reduce-scatter), *broadcast only* (e.g., broadcast, allgather), and *aggregation plus broadcast* (e.g., allreduce). Aggregation requires *in-trees*, with edge directions flowing from leaves to the root, while broadcast requires *out-trees*, where edges flow from the root to leaves. In terms of roots, collective operations can also be categorized as *single-root* (e.g., reduce, broadcast) or *multi-root* (e.g., reduce-scatter, allgather, allreduce). Figure 4.4 shows the relationships between operations. While we explain ForestColl in the context of allgather, it can be easily applied to other operations (§4.5.7).

Knowing the throughput optimality of a given network is crucial for optimizing schedules. Previous work often defines optimality as $\frac{M(N-1)}{N} \cdot \beta$ [21, 160, 23], where $\frac{M(N-1)}{N}$ is the amount of data each node must receive in allgather, and β is the time cost per unit of data. However, this definition only holds when the bottleneck is each individual node's bandwidth. In ML hardware, the bottleneck is often the scale-out network, e.g., the IB bandwidth of a multi-GPU box. In this section, we introduce the concept of *throughput bottleneck cut*, which determines the throughput optimality of allgather.

We model a network topology as a directed graph G, where edge capacities signify link bandwidths, and the vertex set V consists of *compute nodes* V_c (e.g., GPUs) and *switch nodes* V_s. Figure 4.5(a) shows an example. In allgather, each compute node needs to broadcast an equal shard of data to all other compute nodes. We denote the total amount of data M, the number of compute nodes $|V_c| = N$, and thus shard size $\frac{M}{N}$.

In Figure 4.5(a), consider the network cut S^*, which contains all nodes in the top box: compute nodes $c_{1,1}, c_{1,2}, c_{1,3}, c_{1,4}$ and switch node w_1. To finish an allgather, each compute node within S^* must send at least one copy of its shard across the cut to the bottom box; otherwise, $c_{2,1}, c_{2,2}, c_{2,3}, c_{2,4}$ will fail to receive some shards. Therefore, at least $4 \cdot \frac{M}{N}$ amount of data has to exit cut S^*. Note that the total bandwidth exiting S^* is $4b$, counting all four links connecting $c_{1,*}$ to the inter-box switch w_0. Thus, a lower bound for the allgather communication time in this topology is $4 \cdot \frac{M}{N} / (4b) = \frac{M}{8b}$.

The lower bound can be generalized to any topology G. Given an arbitrary network cut $S \subset V$ in G, if there is any compute node not in S (i.e., $S \not\supseteq V_c$), then at least $\frac{M}{N}|S \cap V_c|$ amount of data has to exit S. Let $B^+(S)$ denote the exiting bandwidth of S, i.e., the sum of bandwidths of links going from S to $V - S$, then $\frac{M}{N} \cdot \frac{|S \cap V_c|}{B^+(S)}$ is a lower bound for allgather communication time T_{comm} in topology G. Consider all such cuts in G, then T_{comm} satisfies

$$T_{\text{comm}} \geq \frac{M}{N} \max_{S \subset V, S \not\supseteq V_c} \frac{|S \cap V_c|}{B^+(S)}. \qquad (\star)$$

We call any cut S that maximizes $\frac{|S \cap V_c|}{B^+(S)}$, the ratio of compute nodes within the cut to the exiting bandwidth, as the *throughput bottleneck cut*[2]. In this work, we present ForestColl, which can achieve the RHS of ($\star$). Since ($\star$) is a lower bound of allgather time, ForestColl achieves throughput optimality.

4.5 Algorithm Design

ForestColl's algorithm solves the following problem:

ForestColl Problem Definition

Input: A topology[3] modeled as a directed graph G with integer link bandwidths and a vertex set V consisting of compute nodes V_c and switch nodes V_s.

Output: A set of spanning out-trees $\{T_{u,i}\}_{u \in V_c, i \in [k]}$ over compute nodes, where each tree occupies an equal amount of bandwidth and collectively, they achieve optimality ($\star$).

The set of spanning out-trees $\{T_{u,i}\}_{u \in V_c, i \in [k]}$ consists of k trees rooted at each compute node u, with k determined algorithmically. Correspondingly, a $1/k$ shard of data is broadcast along each out-tree simultaneously. Note that the out-trees are spanning trees of compute nodes only, as explained in Figure 4.3. Figure 4.5(b) shows an example of the out-tree with allocated bandwidth b.

[2] Different from traditional min cut, which only minimizes $B^+(S)$.

[3] Each node must have equal total ingress and egress bandwidth. *Does not exclude oversubscription,* as network tiers can still have varying bandwidths.

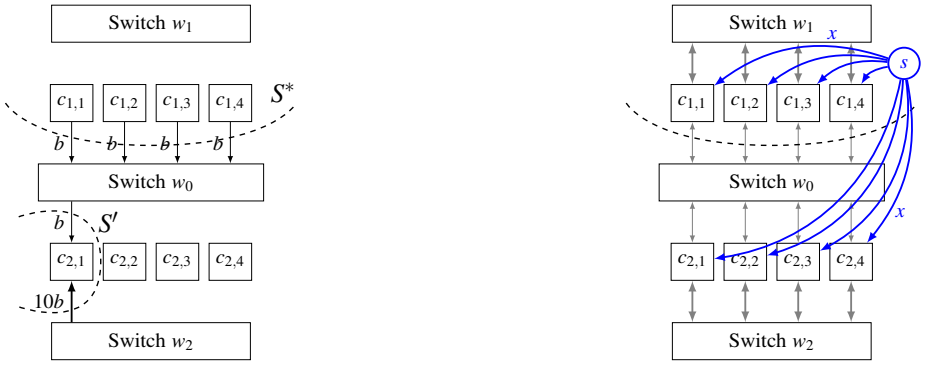


Figure 4.6: The auxiliary network for optimality binary search. (a) shows two cuts: S^*, S', along with their exiting bandwidths. Note that S' is $V - c_{2,1}$ instead of $\{c_{2,1}\}$. (b) shows the auxiliary network that there exists a set of spanning out-trees broadcasting x amount of flow from each compute node if and only if the maxflow from s to every compute node is Nx.

This section introduces the high-level intuitions and steps of ForestColl's algorithms. We provide detailed mathematical analysis in Appendix B.5 and proofs in Appendix B.8. Appendix B.1 includes a summary of notations used in this paper.

4.5.1 Algorithm Overview

ForestColl starts with a binary search to compute the optimality ($\star$) established by throughput bottleneck cut. Iterating through all cuts to find the bottleneck cut is intractable due to the exponential number of possible cuts. Instead, we design an auxiliary network on which we can compute maxflow to determine if a given value is $\geq$ or $<$ than optimality, thus enabling a binary search. Knowing the optimality is crucial for deciding the number of trees per compute node (i.e., k) and the bandwidth per tree to achieve optimality.

In a switch-free topology, after knowing the bandwidth per tree and the number of trees, we directly apply *spanning tree packing* [10, 136, 37, 13, 126] to construct the optimal set of out-trees. In a switch topology, however, we retrofit the *edge splitting technique* [10, 41, 56] to eliminate switch nodes before constructing spanning trees. We replace each switch node with direct logical links between its neighboring nodes, creating a switch-free logical topology where spanning tree packing can be applied. A post-processing step can further enable in-network switch multicast/aggregation.

4.5.2 Optimality Binary Search

We present ForestColl's binary search to compute the throughput optimality ($\star$). Let the total bandwidth of the out-trees rooted at each compute node be x. As each node simultaneously broadcasts a shard of data, the communication time $T_{\text{comm}} = \frac{M}{N} \cdot \frac{1}{x}$. Therefore, to minimize T_{comm}, we need to maximize x. The goal of the binary search is to *find the maximum x such that there exists a set of spanning out-trees broadcasting x amount of flow from each compute node.* We denote the maximum such x as x^*. Before describing the binary search for computing x^*, we first show that $\frac{1}{x^*}$ is precisely the ratio of compute nodes to exiting bandwidth at the throughput bottleneck cut, i.e., $\frac{1}{x^*} = \max_{S \subset V, S \not\supseteq V_c} \frac{|S \cap V_c|}{B^+(S)}$ in optimality ($\star$).

Since each compute node broadcasts x amount of flow to every other compute node, the exiting flow of any cut S is at least $|S \cap V_c|x$ if there is a compute node outside of S. Figure 4.6(a) shows two such cuts: S^* and S'. S^* includes four compute nodes, resulting in an exiting flow of $4x$. S' includes *all* compute and switch nodes except $c_{2,1}$, with an exiting flow of $7x$ to $c_{2,1}$. Suppose $x = b$ (the inter-box link bandwidth). For cut S', the exiting bandwidth $B^+(S')$ is $11b$, more than sufficient for the exiting flow $7b$. However, for cut S^*, the exiting bandwidth $B^+(S^*)$ is exactly $4b$, equal to the required amount of exiting flow. Thus, we are bottlenecked by S^*: if $x > b$, cut S^* cannot sustain the cumulative exiting flow from $c_{1,*}$ to $c_{2,*}$ anymore. Consequently, $x^* = b$ bounds the maximum flow each compute node can simultaneously broadcast. In an arbitrary topology, as we increase x, we will always be bottlenecked by a cut like S^*. This cut is exactly the throughput bottleneck cut in optimality ($\star$), where $\frac{1}{x^*} = \frac{|S^* \cap V_c|}{B^+(S^*)} = \max_{S \subset V, S \not\supseteq V_c} \frac{|S \cap V_c|}{B^+(S)}$. Therefore, x^* is the maximum x that does not overwhelm any cut in the topology.

Detect Overwhelmed Cut: To conduct a binary search for x^*, given a value of x, we determine if $x \leq x^*$ or $> x^*$ by detecting if x overwhelms any cut in the topology. This presents a challenging problem because (i) both the amount of exiting flow and the bandwidth of the cut need to be considered, e.g., S''s bandwidth is not saturated when $x = b$ despite having a larger exiting flow than S^*; (ii) testing every possible cut is intractable due to the exponential number of cuts.

Auxiliary Network: To address the above issues, we construct an auxiliary network as in Figure 4.6(b). We add a source node s and connect s to every compute node with capacity x. Suppose we want to check if S^* is overwhelmed. We pick an arbitrary compute node outside of S^*, say $c_{2,2}$, and calculate the maxflow from s to $c_{2,2}$. If no cut is overwhelmed, then $c_{2,2}$ should get all the flow that s can emit, which equals $8x$. However, if we set $x > b$, while the $4x$ amount of flow from s to $c_{2,*}$ (compute nodes outside of S^*) can directly bypass

Algorithm 1: Optimality Binary Search

Input: A directed graph $G = (V_s \cup V_c, E)$
Output: $\frac{1}{x^*} = \max_{S \subset V, S \not\supseteq V_c} \frac{|S \cap V_c|}{B^+(S)}$
begin
  $l \leftarrow \frac{N-1}{\min_{v \in V_c} B^-(v)}$ // *a lower bound of* $\frac{1}{x^*}$
  $r \leftarrow N-1$ // *an upper bound of* $\frac{1}{x^*}$
  while $r - l \geq 1/\min_{v \in V_c} B^-(v)^2$ **do**
    $\frac{1}{x} \leftarrow (l+r)/2$
    Add node s to G.
    foreach compute node $c \in V_c$ **do**
      Add an edge from s to c with capacity x.
    if the maxflow from s to each $c \in V_c$ is Nx **then**
      $r \leftarrow \frac{1}{x}$ // *case* $\frac{1}{x} \geq \frac{1}{x^*}$
    else
      $l \leftarrow \frac{1}{x}$ // *case* $\frac{1}{x} < \frac{1}{x^*}$
  Find the unique fractional number $\frac{p}{q} \in [l, r]$ such that denominator $q \leq \min_{v \in V_c} B^-(v)$.
  return $\frac{p}{q}$ as $\frac{1}{x^*}$

$B^+(S^*)$, the $4x$ amount of flow from s to $c_{1,*}$ (compute nodes within S^*) must pass through $B^+(S^*)$ to reach $c_{2,2}$, capped at $4b$. Thus, if $x > b$, then the maxflow from s to $c_{2,2}$ is capped at $4x+4b$, which is less than $8x$, signaling a cut being overwhelmed. *The maxflow to $c_{2,2}$ checks all cuts that do not contain $c_{2,2}$.* To check all the exponential number of cuts in the network, we only need to compute a maxflow from s to every compute node c. If the maxflow from s to any c is $< Nx$, then some cut between s and c is overwhelmed, indicating $x > x^*$; otherwise, $x \leq x^*$ for the binary search.

Binary Search: Algorithm 1 shows ForestColl's search for $\frac{1}{x^*}$. We iteratively narrow l and r by adjusting the edge capacities from s and recomputing the maxflows to determine if the midpoint $(l+r)/2$ is $\geq$ or $<$ than $\frac{1}{x^*}$. Thus, we can shrink the range $[l, r]$ small enough for us to determine $\frac{1}{x^*}$ exactly by finding the unique fractional number $\frac{p}{q}$ within $[l, r]$ with denominator $q \leq \min_{v \in V_c} B^-(v)$ (details in Appendix B.5.1).

Determine k: As previously mentioned, knowing the optimality x^* helps us decide the number of trees rooted at each compute node (i.e., k) and the bandwidth allocated per tree. In the spanning tree packing and edge splitting algorithms that ForestColl will apply later, each unit of edge capacity is interpreted as the allocation of one tree instead of one unit of bandwidth. Suppose y is the bandwidth of each tree. Then, we need to adjust the edge capacities by dividing the bandwidth of each edge b_e by y, so that the new capacity b_e/y is the number of trees edge e can sustain. This leads to two requirements for y: (i) $k = x^*/y$ must be an integer, and (ii) b_e/y must be an integer for all edge bandwidth b_e. In Algorithm 1, we have computed $\frac{1}{x^*} = \frac{p}{q}$.

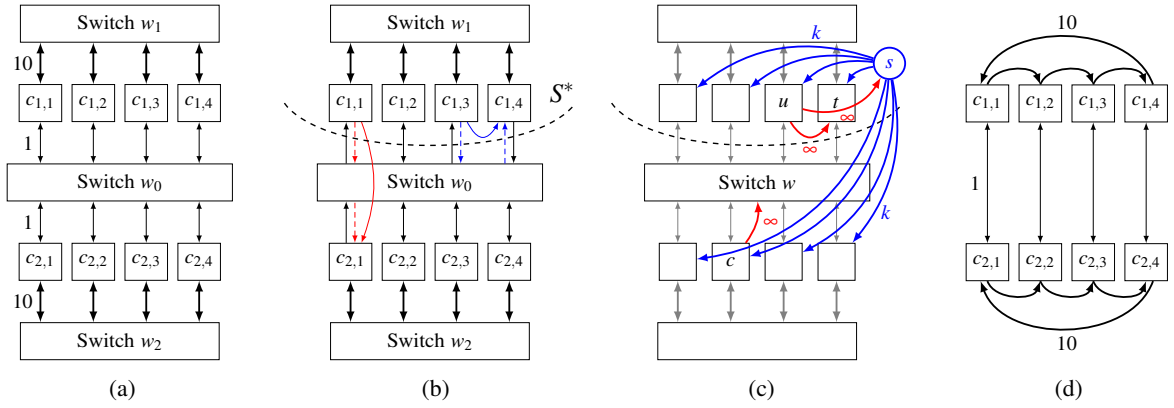


Figure 4.7: Figures explaining the switch node removal process. (a) is the starting topology after optimality binary search scales edge capacities from $\{b, 10b\}$ to $\{1, 10\}$. (b) contains examples of replacing the ingress and egress capacities of a switch node with a direct capacity bypassing the switch. (c) shows an example of the auxiliary network ForestColl uses to compute the γ in Algorithm 2. The ∞ edges are a maxflow trick to ensure $\{u,t,s\}$ and $\{w,c\}$ are on opposite sides of the min cut, as we only want to consider cuts that cut through both $(u,w),(w,t)$. (d) is the final resulting switch-free logical topology.

Thus, by setting $y = \gcd(q, \{b_e\}_{e\in E})/p$, we ensure that both requirements are satisfied, and k, the number of trees rooted at each compute node, is simply x^*/y. For example, the optimality of Figure 4.5(a) is $\frac{1}{x^*} = \frac{4}{4b} = \frac{1}{b}$ bottlenecked by S^*. We have $y = \gcd\{b, b, 10b\} = b$, so the bandwidths of edges are scaled from $\{b, 10b\}$ to $\{1, 10\}$, and $k = 1$. Figure 4.7(a) shows the resulting topology.

The optimality binary search is necessary, as subsequent steps rely on prior knowledge of the optimal k to ensure optimality. A detailed mathematical analysis of the binary search and the derivation of k is included in Appendix B.5.1.

4.5.3 Switch Node Removal

We introduce ForestColl's process to iteratively replace all switch nodes with direct logical links between their neighboring nodes. This allows us to subsequently apply the spanning tree packing algorithm on the resulting switch-free logical topology. The process ensures two key outcomes: (i) *Equivalence:* The spanning trees generated in the logical topology can be mapped back to the original without violating capacity constraints; (ii) *Optimality:* The logical topology retains the same optimal throughput ($\star$). Thus, mapping the optimal trees generated on the logical topology back to the original yields the optimal trees for the switch topology. Although TACCL [128] and TACOS [156] also proposed transforming a switch topology into a switch-free logical one, they replace each switch with fixed, preset connection patterns that guarantee only (i) but not (ii),

leading to performance loss.

Edge Splitting: Originally a graph theory technique for proving connectivity properties [10, 41, 56], we adapt edge splitting to handle switch topologies in the context of collective communications. Starting with the scaled topology as in Figure 4.7(a), for each switch node w, we pair one capacity of an egress link (w,t) with one capacity of an ingress link (u,w), and replace them with one capacity of a direct link (u,t) that bypasses the switch node w. Figure 4.7(b) shows two such examples. In both the red and blue examples, we replace the dashed ingress and egress capacities of switch node w_0 with a direct unit of capacity bypassing w_0. By continuously doing so, we can eliminate all capacities to/from the switch node w_0, which is guaranteed by the assumption of equal ingress and egress bandwidth. Once isolated, w_0 can be safely removed from the topology. Note that u,t do not have to be compute nodes; they can also be other switch nodes that have not yet been removed. By applying this process to each switch node, we derive a switch-free topology, as shown in Figure 4.7(d). The resulting logical topology guarantees *equivalence* to the original, since *we only allocate the capacities of switches to logical connections between compute nodes.*

Choose Ingress Link: Given an egress capacity, there are often multiple ingress links that we can pair and replace. However, arbitrarily choosing an ingress link may lead to performance sacrifice. In the two examples of Figure 4.7(b), the exiting capacity of S^* remains unchanged in the red example but decreases from 4 to 3 in the blue example. This corresponds to decreasing the exiting bandwidth $B^+(S^*)$ from $4b$ to $3b$ in the original topology. Since S^* is a throughput bottleneck cut, any decrease in its bandwidth $B^+(S^*)$ further bottlenecks the overall performance. *Therefore, when replacing capacities, we must ensure that we do not create a bottleneck cut worse than the existing ones.* For an already bottlenecked cut, any decrease in exiting bandwidth is unacceptable. For a non-bottleneck cut, we can only reduce its exiting bandwidth to the point where its ratio of compute nodes to exiting bandwidth just becomes a bottleneck.

From the two examples in Figure 4.7(b), we observe that replacing capacities decreases the exiting capacities of cuts that cut through both the ingress and egress links. Consider replacing a certain capacity of $(u,w),(w,t)$ with (u,t). If we compute among all cuts that cut through both $(u,w),(w,t)$, the minimum decrease γ in exiting capacity that would turn any cut into a bottleneck, then by replacing at most this amount of capacity, we are safe from creating a worse bottleneck cut. Figure 4.7(c) shows the auxiliary network we use to compute γ. Similar to the optimality binary search, we compute maxflow with respect to each compute node and take the minimum. We leave the details to compute γ in Theorem 28 in Appendix B.5.2.

Algorithm 2 shows the pseudocode of the switch node removal process. For each switch node w and each

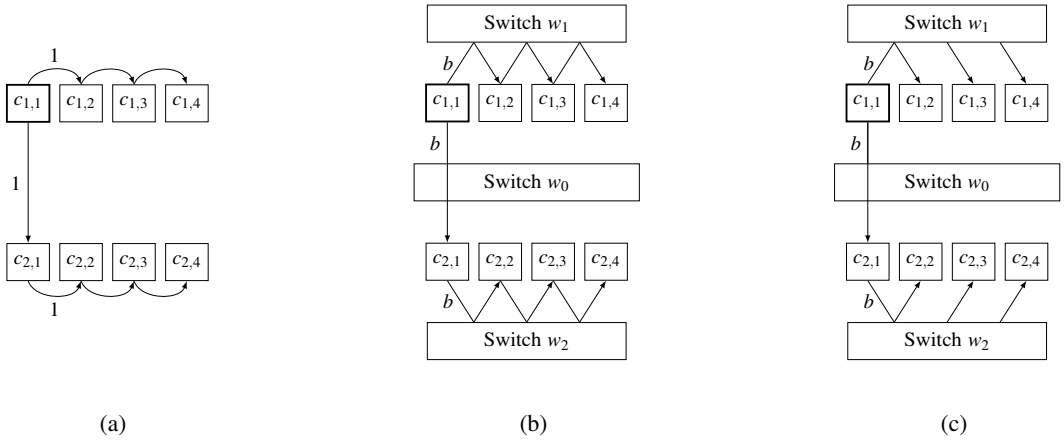
Algorithm 2: Switch Node Removal

Input: A directed graph $G = (V_s \cup V_c, E)$ and k.
Output: A directed graph $H = (V_c, E')$.
begin
  foreach switch node $w \in V_s$ **do**
    foreach egress edge $f = (w,t) \in E$ **do**
      foreach ingress edge $e = (u,w) \in E$ **do**
        Compute γ, the maximum capacity we can safely replace f, e by (u,t), as in Theorem 28.
        if $\gamma = 0$ **then continue**
        Decrease f's and e's capacity by γ. Remove e if its capacity reaches 0.
        Increase the capacity of (u,t) by γ. Add the edge if $(u,t) \notin E$.
        if f's capacity reaches 0 **then break**
      // *Edge f should have 0 capacity at this point.*
      Remove edge f from G.
    // *Node w should be isolated at this point.*
    Remove node w from G.
  return the resulting G as H

egress edge $f = (w,t)$, we pair it with each ingress edge $e = (u,w)$ and calculate γ, the maximum capacity we can safely replace e, f by (u,t). The process iteratively removes switch edges and nodes. Once all switch nodes are removed, we obtain a switch-free logical topology H that is equivalent to the original G and has the same optimal throughput ($\star$). Appendix B.5.2 provides more details of the algorithm.

4.5.4 Spanning Tree Construction

Given the switch-free logical topology like Figure 4.7(d), ForestColl applies the spanning out-tree packing algorithm [13, 126]. Our earlier efforts have ensured that in the logical topology, there exist k spanning out-trees rooted at each node, with respect to the scaled link capacities. With all switch nodes removed, the out-trees simply span all nodes in the topology. Because k can potentially be large, constructing spanning trees one by one may be intractable and not within polynomial time. It turns out that these k out-trees are often not distinct. For example, we may have a batch of $\frac{k}{2}$ identical out-trees and another batch of $\frac{k}{2}$ identical out-trees rooted at the same node. In the algorithm, we construct the out-trees in batches, or rather, trees with capacities. For each node v, the algorithm starts by initializing a k-capacity out-tree containing only a root node $\{v\}$. Then, it iteratively adds edges to each tree, expanding the tree until it spans all nodes in the graph. When adding an edge to an out-tree, the algorithm calculates the maximum capacity μ of the edge that can be added to the out-tree while maintaining the feasibility of constructing the remaining trees. If μ is less than the tree's capacity m, the algorithm splits the tree into two: one with capacity $m-\mu$ and another with capacity μ, adding the edge to the latter. Appendix B.5.3 describes the details of the algorithm.



(a) (b) (c)

Figure 4.8: The constructed spanning out-tree. (a) shows one example of the spanning out-trees generated by applying the spanning tree packing algorithm to Figure 4.7(d). (b) shows the corresponding tree after mapping (a) back to the original topology. (c) shows the post-processed tree utilizing the in-network multicast/aggregation capabilities of switches w_1, w_2.

Fixed-k	1	2	3	4	5	...	83*
Algbw (GB/s)	320	341	343	341	348	...	354

Table 4.1: Fixed-k algorithmic bandwidth for the 2-box AMD MI250 topology. Although the optimal throughput is achieved at $k = 83$, small values of k can already achieve performance close to optimal.

Figure 4.8(a) shows one example of the spanning out-trees constructed by applying the algorithm to Figure 4.7(d). Thanks to the equivalence guarantee of the logical topology, we can map the out-tree back to the original topology, resulting in the tree shown in Figure 4.8(b).

4.5.5 Fixed-k Schedule Generation

In §4.5.2, the optimality binary search automatically determines k (the number of trees rooted at each compute node) and y (the bandwidth utilized by each tree) to achieve theoretically throughput-optimal allgather. However, the k required by optimality can sometimes be a large number. Although the time complexity of ForestColl does not depend on k, a large k may complicate the implementation of the schedule. To address this, ForestColl provides an option to generate the highest-throughput schedule given any fixed k. The method uses a binary search, similar to §4.5.2, to determine the optimality for the fixed k, followed by the usual switch node removal and spanning tree construction to create the out-trees. Appendix B.5.4 provides further details on the algorithm.

Fixed-k schedule generation can significantly simplify the schedule when the optimal k is large. A small k—much smaller than what is required for exact optimality—can still achieve performance very close to the

optimal. Table 4.1 shows an example. In practice, if the optimality binary search gives a too large k, we opt to scan k values within a much smaller range (< 10) and pick the best k for schedule construction.

4.5.6 In-Network Multicast & Aggregation

On the constructed trees, ForestColl applies a post-processing step to utilize the in-network multicast/aggregation of some switches. Counterintuitively, *in-network multicast/aggregation does not affect allgather/reduce-scatter optimality*, as their optimality is determined by the throughput bottleneck cut in §4.4, which is unaffected by the switches' capabilities. The intuition is that, in allgather, while in-network multicast can save GPUs from repeatedly sending the same data, each GPU still must receive $N-1$ distinct data shards, making ingress bandwidth the true bottleneck. Nonetheless, in-network multicast/aggregation is effective for offloading work from GPUs to switches and for reducing overall network traffic.

For each constructed tree, we traverse it from the root, removing traffic that becomes redundant due to in-network multicast of switches. Figure 4.8 gives an example. In Figure 4.8(b), starting from the root, when we reach a node ($c_{2,1}$) sending data to a switch (w_2) capable of in-network multicast, we check if other nodes in the tree also send data to the same switch ($c_{2,2}, c_{2,3} \rightarrow w_2$). As the data being sent is the same throughout the tree, such traffic can be deleted, resulting in the tree in Figure 4.8(c). The same approach also applies to reduce-scatter using in-network aggregation, with everything in the reversed direction. ForestColl is, thus, fully compatible with switches w/ or w/o in-network multicast/aggregation, maximizing performance in all cases.

4.5.7 Other Collective Operations

While introduced in the context of allgather, ForestColl can be easily adapted for other collectives. For reduce-scatter, we reverse the allgather out-trees to create in-trees for aggregation. For allreduce, the in-trees and out-trees can be combined to first aggregate to the roots and then broadcast. However, simply combining reduce-scatter and allgather trees does not guarantee allreduce optimality, since (i) allreduce allows each root to reduce/broadcast variable amounts of data, and (ii) congestion between in-trees and out-trees can be further optimized. To compute allreduce optimality, a linear program is detailed in Appendix B.7. Nevertheless, in practice, directly combining reduce-scatter and allgather trees has been sufficient to achieve optimality in all topologies we have evaluated, and we hypothesize that this holds for any topology with equal bandwidth per compute node. For non-uniform allgather/reduce-scatter, where compute nodes broadcast/reduce varying amounts of data, the link capacities from source node s to compute nodes in the auxiliary networks can be

adjusted to accommodate such variations.

4.6 Evaluation

We present a comprehensive evaluation of ForestColl. §4.6.1 describes our implementations of ForestColl's schedules. §4.6.2 compares the performance of various schedules on both AMD and NVIDIA hardware. §4.6.3 evaluates ForestColl against NCCL [103] on a large-scale GPU cluster. §4.6.4 presents results from LLM training with PyTorch FSDP. Finally, §4.6.5 compares methods for large-scale schedule generation.

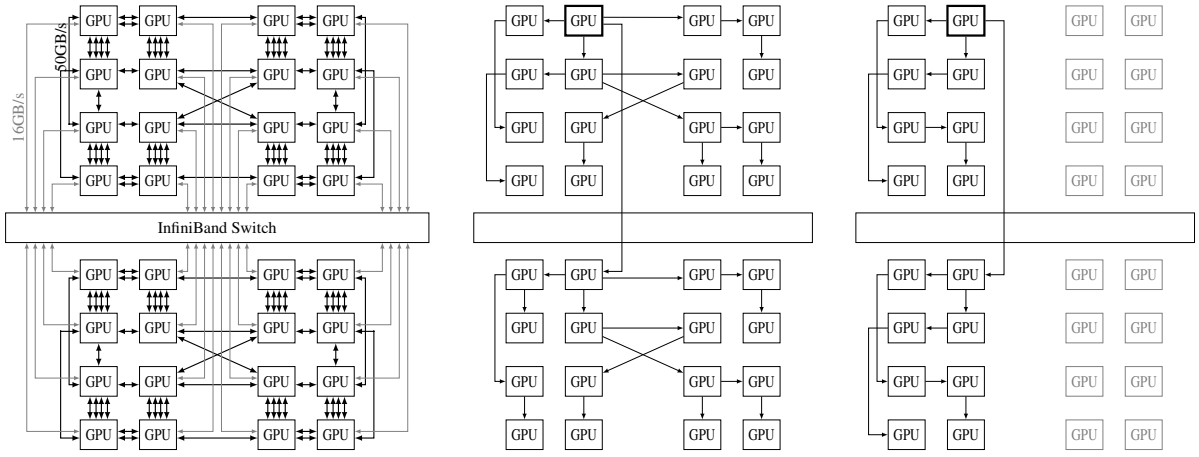
4.6.1 Schedule Implementation

We adopted two implementations: (i) expressing the schedules in XMLs to be executed by the MSCCL runtime, and (ii) using the MSCCL++ library to implement the trees in customized CUDA kernels. MSCCL [95] is built on top of NCCL, sharing the same communication primitives. It is widely used by schedule generation methods, integrates seamlessly with PyTorch, but suffers from scalability limitations. We use it to compare schedule performance, eliminating any differences due to schedule implementation. MSCCL++ [129] is a CUDA library that provides send/recv channels over NVLink and IB networks, supporting zero-copy communication and NVLink SHARP. For large-scale experiments, we use it to build our own customized CUDA kernels that scale effectively and deliver the best performance with ForestColl's schedules.

4.6.2 Schedule Performance Comparison

Setup: We evaluated ForestColl's schedules against vendor libraries and other schedule generation methods on 2-box AMD MI250 and 2-box NVIDIA DGX A100 systems. To eliminate performance differences due to implementation, we uniformly use MSCCL to execute schedules from both ForestColl and the baselines. Thus, any observed performance difference can be attributed solely to the quality of the schedules.

Baselines: We evaluated ForestColl's schedules against TACCL, Blink, and NCCL/RCCL. Due to a runtime error in TACCL's code, we were only able to generate and compare its allgather schedules. For Blink, which lacks publicly available code, we implemented an optimal single-root spanning tree packing based on its paper. Since Blink does not support switch topology, we applied Blink's tree packing to ForestColl's switch-free logical topology to create the "Blink+Switch" baseline. Furthermore, Blink's tree packing is limited to single-root reduce+broadcast for allreduce, and it suggests performing allgather as allreduce without reduction, so we only evaluated Blink's allreduce. Both TACCL and Blink use MSCCL in our experiments. While TACCL's code generates MSCCL schedule XMLs, we used ForestColl's compiler to generate Blink's



(a) MI250 Topology (b) ForestColl 16+16 (c) ForestColl 8+8

Figure 4.9: 2-box AMD MI250 topology and examples of ForestColl's spanning out-trees in 16+16 and 8+8 settings. PCIe switches and IB NICs are omitted for simplicity. (b) and (c) showcase ForestColl's trees rooted at the bold GPU. The complete schedules have at least one tree rooted at each GPU.

XMLs, as both are tree-flow schedules. Finally, on AMD hardware, we compared against RCCL [121] (ROCm Collective Communication Library), AMD's library optimized for its GPUs, instead of NCCL. Because TE-CCL's code lacks executable schedules and SyCCL's was released only shortly before submission, we evaluate them on theoretical performance (§4.6.5).

AMD MI250 Experiments

Testbed Setup: The AMD MI250's complex topology presents significant challenges for schedule generation. Figure 4.9(a) shows the 2-box topology, characterized by a hybrid of direct intra-box connections and an inter-box switch network. Each box contains 16 GPUs, each directly connected to three or four other GPUs through 7× AMD Infinity Fabric links at 50GB/s per link. Each box also has 8× IB NICs, offering 256GB/s total inter-box bandwidth and connected to GPUs via PCIe switches. Note that although ForestColl can model PCIe switches and IB NICs as switch nodes in schedule generation, for simplicity, we omit these components and assume each GPU has 16GB/s bandwidth to the IB switch.

Experiment Setup: We tested allgather, reduce-scatter, and allreduce performance of ForestColl and the baselines in two settings: one involving all 32 GPUs (16+16) and another with 8 GPUs per box (8+8). In the 8+8 setting, we only enable GPUs $0 \sim 7$ in each box, which corresponds to the left half of Figure 4.9(a). The 8+8 setting can result from hybrid training parallelism or bin-packing jobs in a cloud environment.

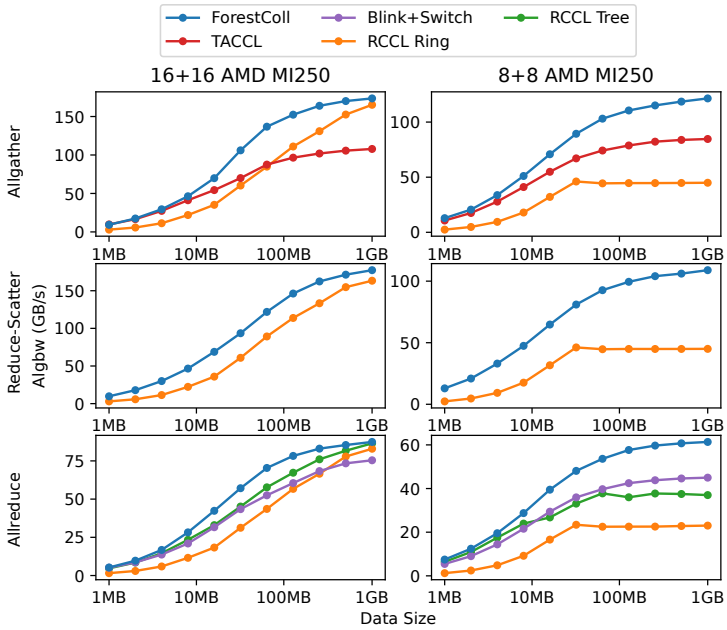


Figure 4.10: Comparing collective communication performance of TACCL, Blink+Switch, RCCL, and ForestColl in 16+16 and 8+8 settings on 2-box AMD MI250. The columns and rows correspond to different settings and collectives, respectively. "Blink+Switch" represents Blink augmented with our switch removal technique, enabling it to support switches.

Figure 4.9(b) and (c) show two examples of the trees ForestColl generated for the 16+16 and 8+8, respectively.

16+16 Results: The left column of Figure 4.10 shows our results in 16+16 setting. We compare performance by algorithmic bandwidth (algbw), calculated as data size divided by runtime. ForestColl consistently outperforms baselines. In allgather comparison with TACCL, ForestColl shows a 61% higher algbw at 1GB data size and an average[4] 36% higher algbw from 1MB to 1GB. Against Blink+Switch in allreduce, ForestColl is 16% faster at 1GB and 23% faster on average. In Figure 4.10, allgather is generally twice as fast as allreduce, contradicting Blink's suggestion to perform allgather as allreduce. RCCL performs comparably to ForestColl at 1GB. However, in allgather and reduce-scatter, RCCL relies solely on the RCCL ring, which has high hop latency. Thus, ForestColl is much faster at smaller data sizes, outperforming RCCL by 91% and 87% on average in allgather and reduce-scatter, respectively. For allreduce, where RCCL tree is available, ForestColl still outperforms RCCL by 15% on average.

8+8 Results: In 8+8 setting, ForestColl also outperforms the baselines. The comparison between ForestColl and TACCL remains similar, with ForestColl being 43% faster at 1GB and 32% faster on average

[4]The arithmetic mean percentage improvement across data sizes.

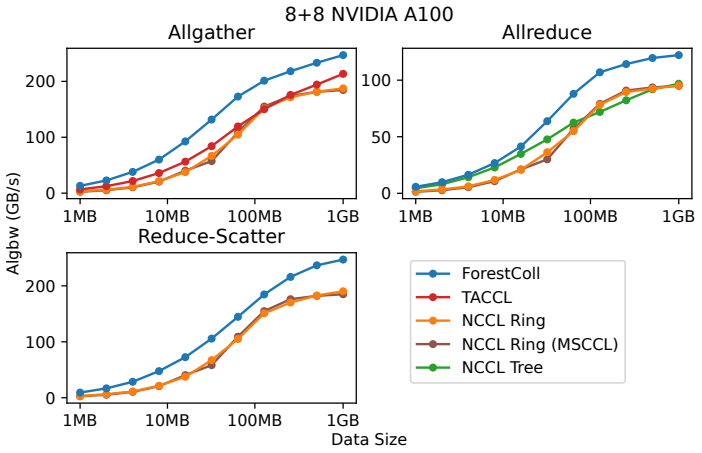


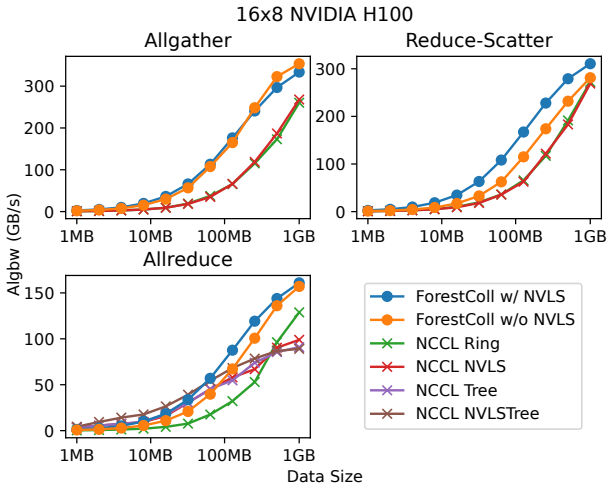
Figure 4.11: Comparing collective communication performance of TACCL, NCCL, and ForestColl on 2-box NVIDIA DGX A100. The "NCCL Ring (MSCCL)" is NCCL ring implemented in MSCCL XMLs to confirm no inherent performance difference between NCCL and MSCCL.

from 1MB to 1GB. Against Blink+Switch, ForestColl is 36% faster both at 1GB and on average. RCCL's performance drops significantly in the 8+8 setting, with ForestColl being on average 2.98x, 2.86x, and 1.40x as fast in allgather, reduce-scatter, and allreduce, respectively. At 1GB data size, ForestColl has 2.7x, 2.42x, and 1.66x algbws compared to RCCL's best-performing algorithm. RCCL's performance drops because it is hand-tuned for fixed topologies with full 16 GPUs per box. In contrast, ForestColl, TACCL, and Blink+Switch can dynamically generate schedules for the new 8+8 topology and have stable performance.

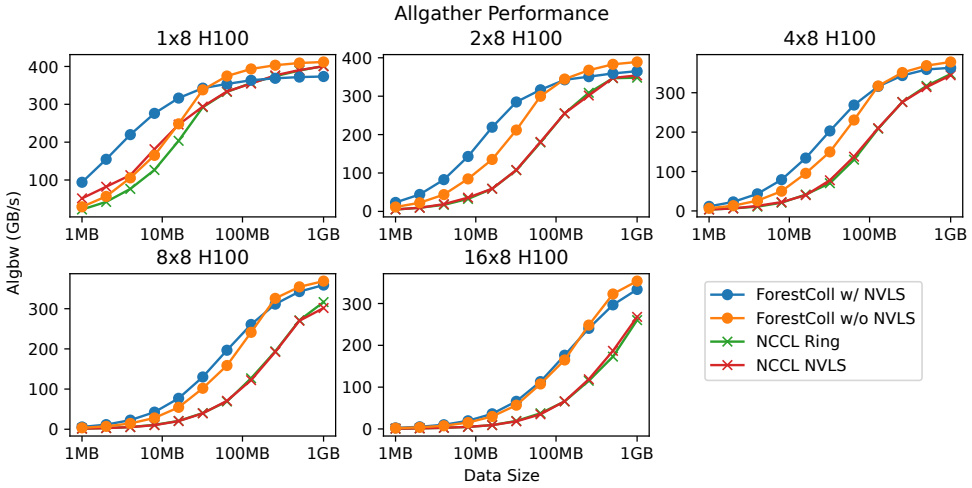
NVIDIA DGX A100 Experiments

Testbed Setup: On a 2-box NVIDIA DGX A100 testbed, we evaluated ForestColl's schedules against TACCL and NCCL. Each box has 8× NVIDIA A100 GPUs, interconnected by an NVSwitch with 300GB/s intra-box bandwidth per GPU. Additionally, every two GPUs are connected to two IB NICs via a PCIe switch. Each NIC offers 25GB/s inter-box bandwidth.

Results: Figure 4.11 presents our experiment results. ForestColl leads in all three collectives by a considerable margin over the closest baseline. While TACCL's performance improves in a switch-only topology, ForestColl still outperforms it by 16% at 1GB and by an average of 53% for sizes from 1MB to 1GB. The improvement of ForestColl over NCCL is even more pronounced. At 1GB, ForestColl achieves 32%, 30%, and 26% higher algbws for allgather, reduce-scatter, and allreduce, respectively. Averaged across 1MB∼1GB, ForestColl is 130%, 85%, and 27% faster than NCCL.



(a) Allgather, reduce-scatter, allreduce algbws at 16x8 H100



(b) Allgather algbws at $\{1, 2, 4, 8, 16\}$x8 H100

Figure 4.12: Comparing NCCL and ForestColl collective communication performance on NVIDIA DGX H100.

In addition to comparing NCCL and ForestColl, we implemented the NCCL ring as MSCCL XML and tested its performance. In Figure 4.11, the NCCL ring in MSCCL shows identical performance to the default NCCL ring in all collectives, showing that ForestColl's improvements stem solely from scheduling optimization, not runtime tuning or inherent performance difference between NCCL and MSCCL.

4.6.3 Large-Scale GPU Cluster Experiments

Setup: We evaluated ForestColl against NCCL on a testbed of 16× DGX H100 boxes (128× H100 GPUs). Each DGX box is equipped with an NVSwitch, providing 450GB/s of intra-box bandwidth per GPU, and

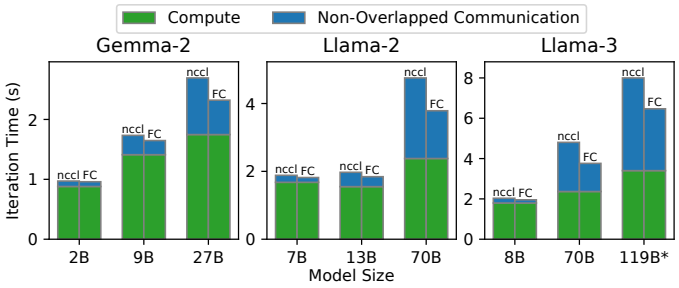


Figure 4.13: Comparing NCCL and ForestColl in Fully Sharded Data Parallel (FSDP) training. The training is on 2x DGX A100 with 16 GPUs using PyTorch FSDP [161]. The compute times are measured by training with communications skipped. Given the limited scale of our testbed, context lengths are set to 2048 for Gemma and 1024 for Llama models, with batch sizes set to the maximum allowed by GPU memory (80GB per GPU). Models are from Hugging Face [155] and use FlashAttention [32, 31] with BFloat16.

8× IB NICs, each offering 50GB/s of inter-box bandwidth. Due to scalability limitations of MSCCL—specifically, each SM can send/recv data from only one peer—we implemented customized CUDA kernels using MSCCL++ based on ForestColl's generated trees.

16x8 Results: Figure 4.12(a) shows allgather, reduce-scatter and allreduce performance on 16× DGX H100 boxes (128× GPUs). ForestColl achieves substantially higher throughput than NCCL in all three collectives, benefiting from both more efficient scheduling and optimized implementation. For allgather and reduce-scatter, ForestColl delivers 32% and 14% higher throughput at 1GB data size. In allreduce, while NCCL's tree algorithms perform better at smaller, latency-sensitive data sizes, ForestColl still dominates at large data sizes, achieving 25% higher throughput at 1GB. In production, existing runtime systems can seamlessly switch between low-latency schedules and ForestColl depending on the input data size, allowing the two to complement each other.

1-16x8 Results: Figure 4.12(b) presents allgather performance from 1 to 16× DGX H100 boxes. At the 1x8 scale, where communication is intra-box only, ForestColl and NCCL have similar throughput, with ForestColl NVLS performing better at smaller data sizes due to the lower latency of zero-copy implementation. At larger scales, where inter-box bandwidth becomes the bottleneck, ForestColl's schedules have less cross-box traffic and outperform NCCL by larger margins. The results show that collective communication implementations guided by ForestColl's scheduling can achieve higher throughput than state-of-the-art communication libraries.

4.6.4 FSDP Training Experiments

To show that the communication speedup provided by ForestColl accelerates LLM training, we run Fully Sharded Data Parallel (FSDP) training [161, 120] with open-source LLMs: Gemma-2 [44] from Google and Llama-2 [93] & 3 [82] from Meta. FSDP is widely used for training large models that far exceed the memory capacity of a single GPU [93, 82, 2]. It shards model parameters across GPUs and allgathers them as needed. In LLM training, FSDP typically allgathers the weights at each layer, performs the computation, and discards the weights to free up memory for the next layer in the forward and backward passes. A reduce-scatter is also needed in the backward pass to aggregate the gradients of each layer.

With MSCCL's seamless integration with PyTorch, we use the same setup as in §4.6.2. Figure 4.13 shows our training results, comparing iteration times (forward+backward) using NCCL vs ForestColl. The iteration times are broken down into compute (comp) time[5] and communication (comm) time not overlapped by comp. For smaller models, such as Gemma-2-2b, Llama-2-7b, and Llama-3-8b, the improvements with ForestColl are minimal, showing reductions in iteration time of less than 5%. With comp accounting for over 88% of the iteration time, these small models are comp-bound, with speedup in comm having little effect on overall performance. However, as model size increases, the trend shifts toward becoming comm-bound. For Gemma-2-27b, Llama-2-70b, and Llama-3-119b[6], comp accounts for only 65%, 50%, and 43% of the iteration times. As a result, compared to NCCL, ForestColl reduces iteration times by 14% for Gemma-2-27b and 20% for both Llama-2-70b and Llama-3-119b.

Large models are more comm-bound for two reasons. First, large models cannot be trained with large batch sizes due to higher GPU memory usage. In our experiments, while a small model like Llama-3-8b can be trained with a batch size of 8, Llama-3-70b is limited to a batch size of 1 to avoid GPU out-of-memory, even with memory-efficient techniques like FlashAttention [32, 31] and BF16 parameters. Second, large models have poor comp-comm overlap due to contention between comp and comm kernels for GPU resources. For example, the comp kernel in FlashAttention uses a number of Streaming Multiprocessors (SM) proportional to the number of attention heads, while comm kernel requires more SMs to saturate bandwidth for large data transfers. For large models, the combined demands of comp and comm kernels exceed a GPU's total SMs,

[5]The comp time is measured by skipping comm operations in an iteration.

[6]Due to the limited scale of our testbed, we reduce num_hidden_layers in Llama-3-405B to 36, creating the 119B model for our experiments.

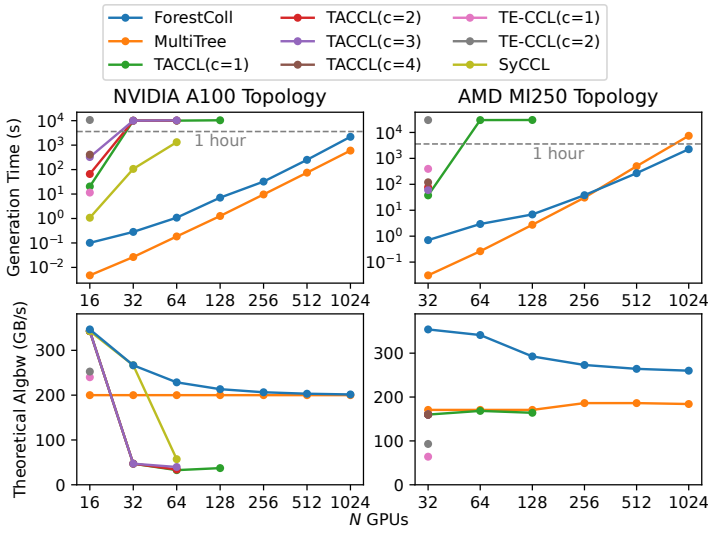


Figure 4.14: Large-scale schedule generation comparison between MultiTree, TACCL, TE-CCL, SyCCL, and ForestColl on NVIDIA A100 and AMD MI250 topologies. The top row compares the time spent on generation, and the bottom row compares the theoretical algorithmic bandwidth of the generated schedules. TACCL and TE-CCL are run with varying numbers of chunks. The time limit is set to 10^4s for A100 and 3×10^4s for MI250.

forcing sequential execution.

4.6.5 Large-Scale Schedule Generation Comparison

In large-scale schedule generation, we compare ForestColl against MultiTree, TACCL, TE-CCL, and SyCCL. While Blink, TACOS, and BFB also conduct schedule generation, Blink and BFB lack support for switch topologies, and the released TACOS implementation does not support switches (as of submission). In Figure 4.14, we compare MultiTree, TACCL, TE-CCL, and ForestColl in generating allgather schedules for NVIDIA A100 and AMD MI250 topologies, with SyCCL included for A100 only due to failures in schedule generation for MI250. Appendix B.3 details our implementation and parallelization of ForestColl's scheduling algorithm.

Setup: TACCL, TE-CCL, and SyCCL use mixed integer linear programming (MILP) to generate schedules. Since solving MILP to optimality is NP-hard and often extremely time-consuming, these methods support setting a time limit to stop early and return the best solution found so far. However, for large topologies, the solver may not find any solution within the time limit. We set a 10^4s time limit for A100 topologies and 3×10^4s (8.3 hours) for the more complicated MI250. MultiTree briefly mentions handling heterogeneity by creating multiedges with unit bandwidth but does not specify how to determine the unit bandwidth. If too

small, all trees could be scheduled to use the same congested link. Here, we set the unit bandwidth to the bandwidth of the slowest link.

Generation Runtime: In Figure 4.14, ForestColl is orders of magnitude faster than TACCL, TE-CCL, and SyCCL in schedule generation. On A100, while TACCL hits the time limit at 4 boxes (32 GPUs), ForestColl generates a schedule in under a second. For larger topologies, TACCL fails to generate any solution within the time limits for >128 GPUs, and TE-CCL cannot generate a schedule beyond two A100 or MI250 boxes. We also tried TE-CCL's A^* technique, but it also failed to scale further. SyCCL runs faster by exploiting topology symmetry, but still fails to scale beyond 64 GPUs. Its paper reports scaling to 512 GPUs (37min) with heuristic tuning, yet ForestColl (4min) is still order of magnitude faster at the same scale. Compared to MultiTree, ForestColl is slower on the A100 topology but faster on the more complex MI250. Despite MultiTree's use of a much simpler greedy algorithm, ForestColl still achieves a similar scalability curve. Notably, ForestColl is the *only* method able to generate both 1024-GPU schedules within 1 hour (A100: 37min, MI250: 38min). Although not generated in seconds, these runtimes are far more practical than MILP approaches and acceptable given that schedules are only precomputed once per topology.

Theoretical Schedule Throughput: ForestColl is always theoretically optimal, outperforming all other methods in Figure 4.14. On A100, TACCL and SyCCL initially match ForestColl but their throughput drops significantly at larger scales due to early stop of the MILP solver at the time limit. MultiTree starts with considerably lower throughput than ForestColl but asymptotically matches it as topology size scales, likely due to the simplicity of A100 topologies. On the more complex MI250, ForestColl outperforms MultiTree by 50%+. Meanwhile, TE-CCL lags behind all other methods.

TACCL, TE-CCL, and SyCCL rely on extensive heuristic tuning with tens of parameters in their configs/sketches. Despite our tuning efforts, we still observed instability in scalability and schedule throughput. In contrast, ForestColl needs only the input topology as a capacitated graph to ensure both scalability and optimality. ForestColl makes finding optimal schedules for large-scale topologies mathematically provable and achievable within tractable runtime bounds.

4.7 Concluding Remarks

Collective communication has become a performance bottleneck in distributed ML. The heterogeneity and diversity of the network topologies pose significant challenges to designing efficient communication algorithms. In this paper, we proposed ForestColl, which *efficiently* generates *throughput-optimal* schedules

for *any type* of network topology. Experiments on popular ML hardware platforms have demonstrated our significant performance improvements over both the platforms' own optimized communication libraries and other state-of-the-art schedule generation techniques.

Chapter 5

Future Work: Automatic Machine Learning Parallelization via Tensor State Search

The previous two chapters focused on optimizing collective communication from the perspectives of network topology and communication scheduling. While accelerating collective communication is essential for distributed machine learning, a more comprehensive end-to-end optimization arises from the co-optimization of computation and communication. In this chapter, we describe our ongoing effort to automate the parallelization of machine learning training, determining the optimal parallelism configuration for a given model and hardware specification.

5.1 Introduction

As machine learning models scale to trillions of parameters, their immense size—coupled with the vast amounts of required training data—necessitates extensive parallelization, often involving tens or even hundreds of thousands of GPUs. An efficient parallelization strategy is crucial for fully utilizing hardware resources, thereby reducing the capital costs and time required to train these models—a mission-critical factor for many AI companies today.

When model sizes are small, data parallelism is commonly employed to parallelize training. In this approach, model parameters are replicated across GPUs, with each GPU processing a distinct shard of the training data to compute gradients, which are then allreduced during the backward pass. As model sizes grow, tensor parallelism and pipeline parallelism become increasingly popular. The former partitions individual model weight matrices across multiple GPUs and uses collective communications to exchange activation results, while the latter divides the model at the layer granularity and processes it in a pipelined manner. In

the era of large language models, novel parallelism strategies have emerged and rapidly become industry standards, due to the special designs of transformer architectures. Context parallelism divides the input sequence into multiple chunks and parallelizes attention computations by exchanging KV caches across GPUs. Expert parallelism assigns the weights of experts in mixture-of-experts (MoE) models to different GPUs, using all-to-all communication to dispatch tokens and aggregate expert outputs.

Modern LLM training requires a combination of these parallelisms to achieve high performance. Automatically determining the optimal multi-dimensional parallelism configuration remains a significant challenge. Prior works such as FlexFlow [59], Alpa [163], and nnScaler [78] have explored automated parallelization for DNN training. However, earlier approaches are limited in several key respects when applied to modern LLM training. First, works like FlexFlow and nnScaler rely on per-device parallelization strategies, which become impractical at the scale of modern LLM training, where tens or hundreds of thousands of GPUs are involved. Second, many earlier works were developed before the rise of massive transformer-based LLMs. They primarily focus on data and model parallelisms and on small models like CNN, rendering them inadequate for the scale, architecture, and communication patterns of contemporary LLMs. Third, memory constraints are paramount in parallelization, yet most prior works' abstractions do not incorporate widely adopted memory-saving techniques such as pipelined processing and activation recomputation. As a result, their parallelization strategies often remain suboptimal in real-world settings.

Our insight into this problem is that all types of parallelisms and optimizations can be represented via abstract states of tensors. Here, we adopt a unified view of tensors, regardless of whether they are parameters, activations, or gradients. The abstract states of tensors encompass their sharding patterns and processing strategies within the computation graph. Once the states of the tensors in a computation graph are determined, the parallelization and optimization strategies can be inferred, along with the associated memory, compute, and communication costs. Thus, we can automatically search for the optimal parallelization and optimization strategy by exploring the space of tensor states.

5.2 Tensor State

In machine learning training, various types of tensors—such as parameters, activations, and gradients—are produced and consumed during computation. Unlike traditional parallelism strategies that distinguish among these tensor types, we adopt a unified view of all tensors. Specifically, we define a tensor's state based on two key aspects: *sharding pattern* and *processing strategy*.

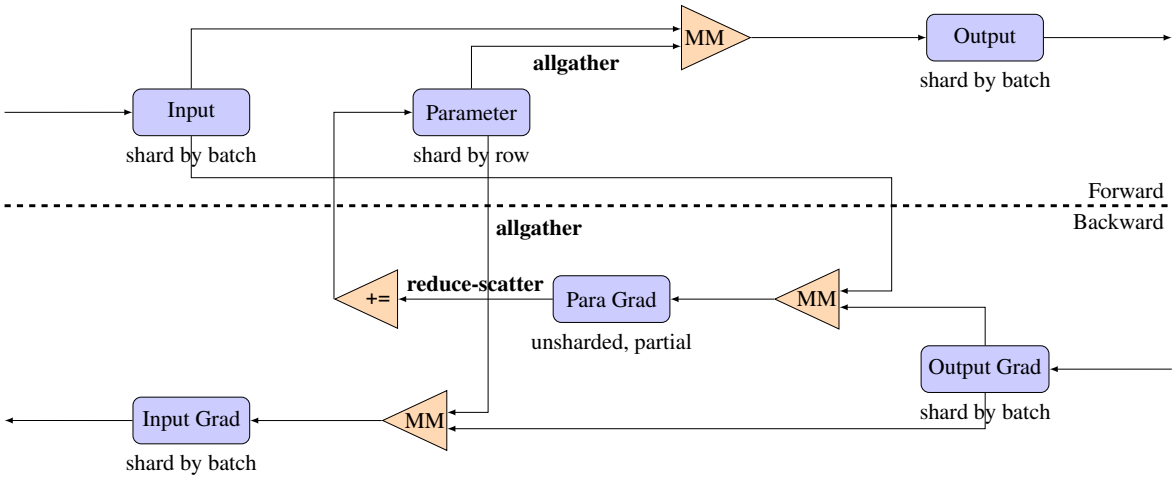


Figure 5.1: FSDP sharding pattern for a matrix multiplication. The compute graph includes both the forward and backward passes. Given the sharding pattern of each tensor, the communication required to execute the graph can be automatically inferred.

5.2.1 Sharding Pattern

Inspired by PyTorch DTensor and JAX Array, the sharding pattern specifies how a given tensor is distributed across a device mesh—a collection of GPUs abstracted as a multi-dimensional topology. For instance, in a typical network combining InfiniBand and NVSwitch, the inter-node InfiniBand network might represent one dimension, while the intra-node NVSwitch connections form another. Ideally, bandwidth resources remain independent across each dimension. For any tensor dimension, it can either be replicated across devices or sharded along one or more dimensions of the device mesh. Additionally, the entire tensor can be *partial* over device mesh dimensions, meaning that obtaining a complete result requires a reduction operation, such as allreduce or reduce-scatter.

All parallelism strategies can be expressed through sharding patterns. For instance, data parallelism shards activation tensors along the batch dimension, while fully sharded data parallelism additionally shards parameter tensors along the row dimension (the outermost dimension). Tensor parallelism and expert parallelism shard parameter tensors across different dimensions. For any operator, such as GEMM or attention, once the sharding patterns of the input and output tensors are determined, the required computations and communications to transition from the input state to the output state can be inferred. Figure 5.1 shows an example of FSDP sharding pattern for a matrix multiplication.

```
graph():
   %primals_1 : [num_users=1] = placeholder[target=primals_1]
   %primals_2 : [num_users=1] = placeholder[target=primals_2]
   %primals_3 : [num_users=1] = placeholder[target=primals_3]
   %tangents_1 : [num_users=1] = placeholder[target=tangents_1]
   %view : [num_users=1] = call_function[target=torch.ops.aten.view.default](args = (%primals_3, [262144, 1024]), kwargs = {})
   %alias_default_1 : [num_users=2] = call_function[target=torch.ops.aten.alias.default](args = (%view,), kwargs = {})
   %mm : [num_users=1] = call_function[target=torch.ops.aten.mm.default](args = (%alias_default_1, %primals_1), kwargs = {})
   %view_1 : [num_users=1] = call_function[target=torch.ops.aten.view.default](args = (%mm, [64, 4096, 2048]), kwargs = {})
   %add : [num_users=1] = call_function[target=torch.ops.aten.add.Tensor](args = (%view_1, %primals_2), kwargs = {})
   %alias_default_2 : [num_users=1] = call_function[target=torch.ops.aten.alias.default](args = (%add,), kwargs = {})
   %alias_default : [num_users=2] = call_function[target=torch.ops.aten.alias.default](args = (%tangents_1,), kwargs = {})
   %sum_1 : [num_users=1] = call_function[target=torch.ops.aten.sum.dim_IntList](args = (%alias_default, [0, 1], True), kwargs = {})
   %view_2 : [num_users=1] = call_function[target=torch.ops.aten.view.default](args = (%sum_1, [2048]), kwargs = {})
   %alias_default_4 : [num_users=1] = call_function[target=torch.ops.aten.alias.default](args = (%view_2,), kwargs = {})
   %view_3 : [num_users=1] = call_function[target=torch.ops.aten.view.default](args = (%alias_default, [262144, 2048]), kwargs = {})
   %permute : [num_users=1] = call_function[target=torch.ops.aten.permute.default](args = (%alias_default_1, [1, 0]), kwargs = {})
   %mm_1 : [num_users=1] = call_function[target=torch.ops.aten.mm.default](args = (%permute, %view_3), kwargs = {})
   %alias_default_3 : [num_users=1] = call_function[target=torch.ops.aten.alias.default](args = (%mm_1,), kwargs = {})
   return [alias_default_2, alias_default_3, alias_default_4, None]
```

(a) torch.fx graph

```
Nodes:
   primals_1: PLACEHOLDER [ab]
   primals_2: PLACEHOLDER [b]
   primals_3: PLACEHOLDER [cda]
   mm: EINSUM "cda,ab->cdb"(primals_3, primals_1) -> [cdb]
   add: ELEMENTWISE(mm[cdb], primals_2[b]) -> [cdb]
   tangents_1: PLACEHOLDER [cdb]
   sum_1: REDUCTION(tangents_1[cdb]) over{c,d} -> [fgb]
   mm_1: EINSUM "acd,cdb->ab"(primals_3, tangents_1) -> [ab]
   output: OUTPUT(add[cdb], mm_1[ab], sum_1[b])
```

(b) Post-processed compute graph

Figure 5.2: Post-processing of torch compiled compute graph

5.2.2 Processing Strategy

While sharding patterns effectively parallelize the computation graph, LLM training often requires additional optimizations, such as pipelined processing and activation recomputation. We argue that these optimizations can also be represented as abstract states of tensors. For instance, in pipelined processing, a tensor does not need to be materialized in memory; instead, the operators that produce and consume it are executed in a streamlined manner to reduce the memory footprint. For activation recomputation, tensors can be marked as persistent or transient. Transient tensors are recomputed during the backward pass, with the recomputation graph derived by backtracing the original computation graph until all encountered tensors are persistent. Thus, tensor states can comprehensively represent these optimization techniques as well.

5.3 Compute Graph

To determine the tensor states, we first need to identify the tensors involved in the training job. Modern machine learning frameworks, such as PyTorch and JAX, support or are based on computation graphs, which encompass all tensors and operators. In PyTorch, for instance, tools like Torch Dynamo [6] can be used to compile the model and extract the computation graph, including both the forward and backward passes.

However, existing computation graph representations are not immediately suitable for our purposes. In PyTorch, the torch.fx graph includes all operations necessary to execute the model. For instance, for every matrix multiplication, the graph incorporates `view` nodes to reshape input tensors into 2D before the GEMM, along with an additional `view` node to restore the output tensor's original dimensions. These nodes do not alter the tensor data and are thus irrelevant to parallelization.

To effectively search the space of tensor states, we must first preprocess the compiled computation graph. This requires an abstraction that captures the essential information for parallelization and optimization while remaining simple enough to facilitate the search. For parallelization, the key information is the dimensions of the tensors. Accordingly, we have implemented a computation graph abstraction that is agnostic to the order of tensor dimensions. For example, a matrix A and its transpose A^T are treated as the same tensor in the graph. This design choice stems from the observation that modern machine learning kernel libraries, such as CUTLASS, incorporate optimizations for different input tensor strides. By delegating the handling of tensors' memory layouts to these lower-level kernel libraries, we alleviate the burden on the parallelization search process. Figure 5.2 shows an example of the post-processed compute graph.

5.4 Future Work: Tensor State Search

The computation graph processing establishes the foundation for the tensor state search. However, two major challenges must be addressed. First, the search space is potentially enormous in complexity. A single transformer block may contain tens of tensors across the forward and backward passes, with each tensor featuring multiple dimensions that can be sharded in various combinations across device mesh dimensions. The possible combinations of states for all tensors can lead to an explosive growth in the search space, posing a significant challenge for efficient exploration. Second, the theoretical cost model used in the search must accurately reflect end-to-end training performance in practice. Parallelization and processing strategies are only high-level abstractions. The actual training performance is also significantly influenced by lower-level implementations, such as kernel libraries and hardware features.

While we are still developing the search algorithms, several preliminary ideas have emerged. In the PyTorch AutoParallel library [114], the authors employ mixed-integer linear programming (MILP) to identify optimal tensor shardings. They precompute a cost map for every possible combination of input and output tensor sharding patterns and incorporate it into the MILP formulation—a technique also used similarly in Alpa. Our ongoing efforts aim to extend this approach to encompass processing strategies, such as

pipelined execution and activation recomputation. Alternative ideas include leveraging genetic algorithms or reinforcement learning techniques to explore the space of tensor states. Ultimately, the search algorithm must efficiently explore the search space while remaining adaptable to the underlying theoretical cost model.

For the theoretical cost model, we determined that it cannot be overly fine-grained if it is to remain meaningful for end-to-end performance. Since our approach cannot account for lower-level execution optimizations, the cost model must operate at an appropriate abstraction level where accuracy can be ensured. Accordingly, we adopt a model that separately computes the total compute cost and total communication cost for a given set of tensor states, then takes the maximum of the two. These costs are derived from the TFLOPs and communication volume of the parallelized computation graph, combined with the target hardware's capabilities. This ensures that the hardware is neither overly compute-bound nor communication-bound, providing a theoretical limit on the performance achievable through lower-level optimizations.

Chapter 6

Conclusion

This thesis develops techniques to optimize collective communication for distributed machine learning workloads from network topology, communication scheduling, and software–hardware implementation.

We first propose methods to co-optimize network topology and communication scheduling for direct-connect networks, with a particular focus on optical circuit networks. To avoid the NP-hardness of discrete optimization, we develop expansion techniques and breadth-first broadcast (BFB) schedule generation algorithms that enable efficient construction of large-scale topologies and their corresponding communication schedules. Our topology finder systematically explores the trade-off between low-hop and load-balanced topologies to determine the most suitable topology for a given workload. We provide rigorous mathematical analysis under the α-β communication cost model and demonstrate that our methods outperform widely used direct-connect topologies in both testbed benchmarks and large-scale LLM training simulations.

To optimize collective communication on state-of-the-art GPU platforms for machine learning, we introduce ForestColl, a framework that generates optimal collective communication schedules for the diverse and heterogeneous network topologies of GPU vendors' hybrid scale-up and scale-out networks. ForestColl addresses a key theoretical question—given a topology, what is the optimal collective communication throughput?—and further provides algorithms that construct schedules attaining this theoretical bound. Unlike prior approaches that rely on NP-hard formulations such as MILP or SMT solving, ForestColl leverages earlier graph theory results to achieve polynomial-time scheduling. In evaluation, ForestColl consistently produces high-throughput schedules across complex hardware topologies, and its schedule generation is orders of magnitude faster than existing methods yet remains throughput-optimal.

Finally, we introduce our novel approach to automatically searching for optimal parallelization and optimization strategies in machine learning training. Our proposal involves parallelizing model training by

determining the sharding and processing states of tensors within the compiled computation graph. We unify various types of parallelisms by treating all tensors equally, irrespective of whether they represent parameters, activations, or gradients. We propose a computation graph abstraction that focuses on tensor dimensions while remaining agnostic to memory layouts. Additionally, we discuss ideas for searching the space of tensor states and an appropriate cost model.

Bibliography

[1] Anirudha Agrawal, Shaizeen Aga, Suchita Pati, and Mahzabeen Islam. Optimizing ml concurrent computation and communication with gpu dma engines, 2024.

[2] Ai2. Olmo: Accelerating the science of language models, 2024.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 63–74, New York, NY, USA, 2008. Association for Computing Machinery.

[4] AMD CDNA^{TM} 2 Architecture. https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf.

[5] AMD CDNA^{TM} 3 Architecture. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf.

[6] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery.

[7] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari,

Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, Boston, MA, April 2023. USENIX Association.

[8] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. 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*, SIGCOMM '20, page 782–797, New York, NY, USA, 2020. Association for Computing Machinery.

[9] S. Bang, A. Dubickas, J.H. Koolen, and V. Moulton. There are only finitely many distance-regular graphs of fixed valency greater than two. *Advances in Mathematics*, 269:1–55, 2015.

[10] JØrgen Bang-Jensen, András Frank, and Bill Jackson. Preserving and increasing local edge-connectivity in mixed graphs. *SIAM Journal on Discrete Mathematics*, 8(2):155–178, 1995.

[11] Mike Barnett, Lance Shuler, Robert van De Geijn, Satya Gupta, David G Payne, and Jerrell Watts. Interprocessor collective communication library (intercom). In *Proceedings of IEEE Scalable High Performance Computing Conference*, pages 357–364. IEEE, 1994.

[12] Prithwish Basu, Liangyu Zhao, Jason Fantl, Siddharth Pal, Arvind Krishnamurthy, and Joud Khoury. Efficient all-to-all collective communication schedules for direct-connect topologies. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '24, page 28–41, New York, NY, USA, 2024. Association for Computing Machinery.

[13] Kristóf Bérczi and András Frank. Packing arborescences (combinatorial optimization and discrete algorithms). *RIMS Kokyuroku Bessatsu*, B23:1–31, 2010.

[14] J-C. Bermond and P. Fraigniaud. Broadcasting and NP-completeness. In *Graph Theory Notes of New York*, volume XXII, pages 8–14, 1992.

[15] J.-C. Bermond, N. Homobono, and C. Peyrat. Connectivity of kautz networks. *Discrete Math.*, 114(1-3):51–62, apr 1993.

[16] Maciej Besta and Torsten Hoefler. Slim Fly: A cost effective low-diameter network topology. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, page 348–359. IEEE Press, 2014.

[17] F. Boesch and Jhing-Fa Wang. Reliable circulant networks with minimum transmission delay. *IEEE Transactions on Circuits and Systems*, 32(12):1286–1291, 1985.

[18] Shahid H Bokhari and Harry Berryman. Complete exchange on a circuit switched mesh. In *Proceedings Scalable High Performance Computing Conference SHPCC-92.*, pages 300–306. IEEE, 1992.

[19] Broadcom P2200G - 2x200GbE PCIe NIC. https://www.broadcom.com/products/ethernet-connectivity/network-adapters/200gb-nic-ocp/p2200g.

[20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[21] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 62–75, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Jiamin Cao, Shangfeng Shi, Jiaqi Gao, Weisen Liu, Yifan Yang, Yichi Xu, Zhilong Zheng, Yu Guan, Kun Qian, Ying Liu, Mingwei Xu, Tianshu Wang, Ning Wang, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. Syccl: Exploiting symmetry for efficient collective communication scheduling. In *Proceedings of the ACM SIGCOMM 2025 Conference*, SIGCOMM '25, page 645–662, New York, NY, USA, 2025. Association for Computing Machinery.

[23] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: Theory, practice, and experience: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(13):1749–1783, sep 2007.

[24] Ernie Chan, Robert Van De Geijn, William Gropp, and Rajeev Thakur. Collective communication on architectures that support simultaneous communication over multiple links. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 2–11, 2006.

[25] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Chengji Yao, Ziheng Jiang, Haibin Lin, Xin Jin, and Xin Liu. Flux: Fast software-based communication overlap on gpus through kernel fusion, 2024.

[26] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 178–191, New York, NY, USA, 2024. Association for Computing Machinery.

[27] Zixuan Chen, Xuandong Liu, Minglin Li, Yinfan Hu, Hao Mei, Huifeng Xing, Hao Wang, Wanxin Shi, Sen Liu, and Yang Xu. Rina: Enhancing ring-allreduce with in-network aggregation in distributed model training. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*, pages 1–12, 2024.

[28] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 241–251, 2019.

[29] Sanghun Cho, Hyojun Son, and John Kim. Logical/physical topology-aware collective communication in deep learning training. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 56–68, 2023.

[30] José M Cámara, Miquel Moretó, Enrique Vallejo, Ramon Beivide, Jose Miguel-Alonso, Carmen Martínez, and Javier Navaridas. Twisted torus topologies for enhanced interconnection networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1765–1778, 2010.

[31] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024.

[32] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[33] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. Flare: flexible in-network allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[34] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.

[35] DeepSeek-AI. Deepseek-v3 technical report, 2025.

[36] DistanceRegular.org. https://www.math.mun.ca/distanceregular/.

[37] Jack Edmonds. Edge-disjoint branchings. *Combinatorial algorithms*, pages 91–96, 1973.

[38] A.-H. Esfahanian, L.M. Ni, and B.E. Sagan. The twisted n-cube with application to multiprocessing. *IEEE Transactions on Computers*, 40(1):88–93, 1991.

[39] Peyman Faizian, Md Atiqul Mollah, Xin Yuan, Zaid Alzaid, Scott Pakin, and Michael Lang. Random regular graph and generalized de bruijn graph with k-shortest path routing. *IEEE Transactions on Parallel and Distributed Systems*, 29(1):144–155, 2017.

[40] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[41] A. Frank. On connectivity properties of eulerian digraphs. In Lars Dovling Andersen, Ivan Tafteberg Jakobsen, Carsten Thomassen, Bjarne Toft, and Preben Dahl Vestergaard, editors, *Graph Theory in Memory of G.A. Dirac*, volume 41 of *Annals of Discrete Mathematics*, pages 179–194. Elsevier, 1988.

[42] Xinwei Fu, Zhen Zhang, Haozheng Fan, Guangtai Huang, Mohammad El-Shabani, Randy Huang, Rahul Solanki, Fei Wu, Ron Diamant, and Yida Wang. Distributed training of large language models on aws trainium. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 961–976, New York, NY, USA, 2024. Association for Computing Machinery.

[43] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 57–70, New York, NY, USA, 2024. Association for Computing Machinery.

[44] Gemma Team, Google DeepMind. Gemma 2: Improving open language models at a practical size, 2024.

[45] Andrew Gibiansky. Bringing hpc techniques to deep learning. *Baidu Research, Tech. Rep.*, 2017. https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/.

[46] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, oct 1988.

[47] Chen Griner, Johannes Zerwas, Andreas Blenk, Manya Ghobadi, Stefan Schmid, and Chen Avin. Cerberus: The power of choices in datacenter topology design - a throughput perspective. *Proc. ACM Meas. Anal. Comput. Syst.*, 5(3), dec 2021.

[48] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[49] Frank Harary and Robert Z. Norman. Some properties of line digraphs. *Rendiconti del Circolo matematico di Palermo*, 9(2):161–168, 1960.

[50] Ching-Tien Ho and S Lennart Johnsson. Distributed routing algorithms for broadcasting and personalized communication in hypercubes. In *ICPP*, pages 640–648, 1986.

[51] Roger W Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel computing*, 20(3):389–398, 1994.

[52] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, page 92–98, New York, NY, USA, 2016. Association for Computing Machinery.

[53] Jiayi Huang, Pritam Majumder, Sungkeun Kim, Abdullah Muzahid, Ki Hwan Yum, and Eun Jung Kim. Communication algorithm-architecture co-design for distributed deep learning. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 181–194. IEEE Press, 2021.

[54] Imase and Itoh. A design for directed graphs with minimum diameter. *IEEE Transactions on Computers*, C-32(8):782–784, 1983.

[55] Intel. oneAPI Collective Communications Library (oneCCL). https://github.com/oneapi-src/oneCCL.

[56] Bill Jackson. Some remarks on arc-connectivity, vertex splitting, and orientation in graphs and digraphs. *Journal of Graph Theory*, 12(3):429–436, 1988.

[57] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 402–416, New York, NY, USA, 2022. Association for Computing Machinery.

[58] S. Jeaugey. Massively scale your deep learning training with nccl 2.4. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/, 2019.

[59] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13, 2019.

[60] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, Santa Clara, CA, April 2024. USENIX Association.

[61] S.L. Johnsson and C.-T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, 1989.

[62] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 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*, 2023.

[63] 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*, SIGCOMM '21, page 657–675, New York, NY, USA, 2021. Association for Computing Machinery.

[64] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, page 77–88, USA, 2008. IEEE Computer Society.

[65] Joohwan Kim, Arash Ghayoori, and R. Srikant. All-to-all communication in random regular directed graphs. *IEEE Transactions on Network Science and Engineering*, 1(1):43–52, 2014.

[66] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

[67] Kartik Lakhotia, Maciej Besta, Laura Monroe, Kelly Isham, Patrick Iff, Torsten Hoefler, and Fabrizio Petrini. PolarFly: A cost-effective and flexible low-diameter topology. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.

[68] 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. USENIX Association, April 2021.

[69] Sabuj Laskar, Pranati Majhi, Sungkeun Kim, Farabi Mahmud, Abdullah Muzahid, and Eun Jung Kim. Enhancing collective communication in mcm accelerators for deep learning training. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–16, 2024.

[70] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations*, 2021.

[71] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.

[72] Dongsheng Li, Xicheng Lu, and Jinshu Su. Graph-theoretic analysis of kautz topology and dht schemes. In Hai Jin, Guang R. Gao, Zhiwei Xu, and Hao Chen, editors, *Network and Parallel Computing*, pages 308–315, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[73] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. Accelerating distributed MoE training and inference with lina. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 945–959, Boston, MA, July 2023. USENIX Association.

[74] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on operating systems design and implementation (OSDI 14)*, pages 583–598, 2014.

[75] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch distributed: experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, August 2020.

[76] 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.

[77] libfabric Open Fabrics Interfaces (OFI). https://github.com/ofiwg/libfabric.

[78] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, Weijiang Xu, Mao Yang, Lintao Zhang, and Lidong Zhou. nnScaler: Constraint-Guided parallelization plan generation for deep learning training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 347–363, Santa Clara, CA, July 2024. USENIX Association.

[79] Hong Liu, Ryohei Urata, Kevin Yasumura, Xiang Zhou, Roy Bannon, Jill Berger, Pedram Dashti, Norm Jouppi, Cedric Lam, Sheng Li, Erji Mao, Daniel Nelson, George Papen, Mukarram Tariq, and Amin Vahdat. Lightwave fabrics: At-scale optical circuit switching for datacenter and machine learning systems. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 499–515, New York, NY, USA, 2023. Association for Computing Machinery.

[80] Shuo Liu, Qiaoling Wang, Junyi Zhang, Wenfei Wu, Qinliang Lin, Yao Liu, Meng Xu, Marco Canini, Ray C. C. Cheung, and Jianfei He. In-network aggregation with transport transparency for distributed training. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 376–391, New York, NY, USA, 2023. Association for Computing Machinery.

[81] Xuting Liu, Behnaz Arzani, Siva Kesava Reddy Kakarla, Liangyu Zhao, Vincent Liu, Miguel Castro, Srikanth Kandula, and Luke Marshall. Rethinking machine learning collective communication as

a multi-commodity flow problem. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 16–37, New York, NY, USA, 2024. Association for Computing Machinery.

[82] Llama Team, AI @ Meta. The llama 3 herd of models, 2024.

[83] D. Loguinov, J. Casas, and Xiaoming Wang. Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. *IEEE/ACM Transactions on Networking*, 13(5):1107–1120, 2005.

[84] Dmitri Loguinov, Anuj Kumar, Vivek Rai, and Sai Ganesh. Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, page 395–406, New York, NY, USA, 2003. Association for Computing Machinery.

[85] Yunfeng Lu, Huaxi Gu, Xiaoshan Yu, and Peng Li. X-NEST: A scalable, flexible, and high-performance network architecture for distributed machine learning. *Journal of Lightwave Technology*, 39(13):4247–4254, 2021.

[86] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 82–97, 2020.

[87] Liang Luo, Buyun Zhang, Michael Tsang, Yinbin Ma, Ching-Hsiang Chu, Yuxin Chen, Shen Li, Yuchen Hao, Yanli Zhao, Guna Lakshminarayanan, Ellie Wen, Jongsoo Park, Dheevatsa Mudigere, and Maxim Naumov. Disaggregated multi-tower: Topology-aware modeling technique for efficient large scale recommendation. In P. Gibbons, G. Pekhimenko, and C. De Sa, editors, *Proceedings of Machine Learning and Systems*, volume 6, pages 266–278, 2024.

[88] Kshiteej Mahajan, Ching-Hsiang Chu, Srinivas Sridharan, and Aditya Akella. Better together: Jointly optimizing ML collective scheduling and execution planning using SYNDICATE. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 809–824, Boston, MA, April 2023. USENIX Association.

[89] Karthik Mandakolathur and Sylvain Jeaugey. Doubling all2all Performance with NVIDIA Collective Communication Library 2.12. https://developer.nvidia.com/blog/doubling-all2all-performance-with-nvidia-collective-communication-library-2-12/.

[90] Paul Theo Meijer. *Connectivities and diameters of circulant graphs*. PhD thesis, Theses (Dept. of Mathematics and Statistics)/Simon Fraser University, 1991.

[91] 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.

[92] 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*, 2017.

[93] Meta. Llama 2: Open foundation and fine-tuned chat models, 2023.

[94] Dimitrios Michail, Joris Kinable, Barak Naveh, and John V. Sichi. Jgrapht—a java library for graph data structures and algorithms. *ACM Trans. Math. Softw.*, 46(2), May 2020.

[95] Microsoft Collective Communication Library (MSCCL). https://github.com/Azure/msccl.

[96] Mirka Miller and Jozef Siran. Moore graphs and beyond: A survey of the degree/diameter problem. *Electronic Journal of Combinatorics*, 1000, 2013.

[97] Mixtral 8x22B. https://mistral.ai/news/mixtral-8x22b/.

[98] E. A. MONAKHOVA. A survey on undirected circulant graphs. *Discrete Mathematics, Algorithms and Applications*, 04(01):1250002, 2012.

[99] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[100] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, Krishnakumar Nair, Isabel Gao, Bor-Yiing Su, Jiyan Yang, and Mikhail Smelyanskiy. Deep learning training in facebook data centers: Design of scale-up and scale-out systems, 2020.

[101] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems, 2019.

[102] NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/.

[103] NVIDIA Collective Communication Library (NCCL). https://github.com/NVIDIA/nccl.

[104] NVIDIA ConnectX-6 Dx Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectX-6-dx-datasheet.pdf.

[105] NVIDIA DGX-1 With Tesla V100 System Architecture. https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf.

[106] NVIDIA DGX A100 System Architecture. https://resources.nvidia.com/en-us-dgx-systems/dgxa100-system.

[107] NVIDIA H100 Tensor Core GPU Architecture Overview. https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper.

[108] NVIDIA. Optimizing for Low-Latency Communication in Inference Workloads with JAX and XLA. https://developer.nvidia.com/blog/optimizing-for-low-latency-communication-in-inference-workloads-with-jax-and-xla/.

[109] Nvidia. Nemotron-4 340b technical report, 2024.

[110] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[111] Suchita Pati, Shaizeen Aga, Mahzabeen Islam, Nuwan Jayasena, and Matthew D. Sinclair. T3: Transparent tracking & triggering for fine-grained overlap of compute & collectives. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 1146–1164, New York, NY, USA, 2024. Association for Computing Machinery.

[112] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 16–29, New York, NY, USA, 2019. Association for Computing Machinery.

[113] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. 5:606–624, 2023.

[114] PyTorch AutoParallel. https://github.com/meta-pytorch/autoparallel.

[115] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 691–706, New York, NY, USA, 2024. Association for Computing Machinery.

[116] Rolf Rabenseifner. Optimization of collective reduction operations. In Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, pages 1–9, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[117] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[118] 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, Santa Clara, CA, April 2024. USENIX Association.

[119] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 18332–18346. PMLR, 17–23 Jul 2022.

[120] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[121] ROCm Collective Communication Library (RCCL). https://github.com/ROCm/rccl.

[122] Jose Rolim, Pavel Tvrdik, Jan Trdlička, and Imrich Vrto. Bisecting de bruijn and kautz graphs. *Discrete Applied Mathematics*, 85(1):87–97, 1998.

[123] Paul Sack and William Gropp. Collective algorithms for multiported torus networks. *ACM Trans. Parallel Comput.*, 1(2), feb 2015.

[124] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Comput.*, 35(12):581–594, dec 2009.

[125] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[126] A. Schrijver. Combinatorial optimization : polyhedra and efficiency, 2003.

[127] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow, 2018.

[128] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. TACCL: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 593–612, Boston, MA, April 2023. USENIX Association.

[129] Aashaka Shah, Abhinav Jangda, Binyang Li, Caio Rocha, Changho Hwang, Jithin Jose, Madan Musuvathi, Olli Saarikivi, Peng Cheng, Qinghua Zhou, Roshan Dathathri, Saeed Maleki, and Ziyue Yang. Msccl++: Rethinking gpu communication abstractions for cutting-edge ai applications, 2025.

[130] Noam Shazeer, *Azalia Mirhoseini, *Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations*, 2017.

[131] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[132] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[133] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 225–238, San Jose, CA, April 2012. USENIX Association.

[134] Dan Stanzione, John West, R. Todd Evans, Tommy Minyard, Omar Ghattas, and Dhabaleswar K. Panda. Frontera: The evolution of leadership computing at the national science foundation. In *Practice and Experience in Advanced Research Computing*, PEARC '20, page 106–111, New York, NY, USA, 2020. Association for Computing Machinery.

[135] Young-Joo Suh and K.G. Shin. All-to-all personalized communication in multidimensional torus and mesh networks. *IEEE Transactions on Parallel and Distributed Systems*, 12(1):38–59, 2001.

[136] Robert Endre Tarjan. A good algorithm for edge-disjoint branching. *Information Processing Letters*, 3(2):51–53, 1974.

[137] Telescent G4 Network Topology Manager. https://www.telescent.com/products.

[138] Texas Advanced Computing Center (TACC). https://www.tacc.utexas.edu/.

[139] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[140] The Mosaic Research Team. DBRX 132B. https://www.databricks.com/blog/introducing-dbrx-new-state-art-open-llm.

[141] Thao-Nguyen TRUONG and Ryousei TAKANO. Hybrid electrical/optical switch architectures for training distributed deep learning in large-scale. *IEICE Transactions on Information and Systems*, E104.D(8):1332–1339, 2021.

[142] Yuichiro Ueno and Rio Yokota. Exhaustive study of hierarchical allreduce patterns for large messages between gpus. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 430–439, 2019.

[143] Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. Xpander: Towards optimal-performance datacenters. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 205–219, New York, NY, USA, 2016. Association for Computing Machinery.

[144] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[145] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 172–186, 2020.

[146] Guanhua Wang, Chengming Zhang, Zheyu Shen, Ang Li, and Olatunji Ruwase. Domino: Eliminating communication in llm training via generic tensor slicing and overlapping, 2024.

[147] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. Towards Domain-Specific network transport for distributed DNN training. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1421–1443, Santa Clara, CA, April 2024. USENIX Association.

[148] Ruiqi Wang, Dezun Dong, Fei Lei, Junchao Ma, Ke Wu, and Kai Lu. Roar: A router microarchitecture for in-network allreduce. In *Proceedings of the 37th ACM International Conference on Supercomputing*, ICS '23, page 423–436, New York, NY, USA, 2023. Association for Computing Machinery.

[149] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 93–106, New York, NY, USA, 2022. Association for Computing Machinery.

[150] W. Wang, M. Ghobadi, K. Shakeri, Y. Zhang, and N. 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, Los Alamitos, CA, USA, aug 2024. IEEE Computer Society.

[151] Weiyang Wang, Manya Ghobadi, Kayvon Shakeri, Ying Zhang, and Naader Hasani. How to build low-cost networks for large language models (without sacrificing performance)?, 2023.

[152] 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, Boston, MA, April 2023. USENIX Association.

[153] Kevin C Webb, Alex C Snoeren, and Kenneth Yocum. Topology switching for data center networks. In *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 11)*, 2011.

[154] Udayanga Wickramasinghe and Andrew Lumsdaine. A survey of methods for collective communication optimization and tuning. *arXiv preprint arXiv:1611.06334*, 2016.

[155] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[156] William Won, Midhilesh Elavazhagan, Sudarshan Srinivasan, Swati Gupta, and Tushar Krishna. TACOS: Topology-Aware Collective Algorithm Synthesizer for Distributed Machine Learning. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 856–870, Los Alamitos, CA, USA, November 2024. IEEE Computer Society.

[157] Yongji Wu, Yechen Xu, Jingrong Chen, Zhaodong Wang, Ying Zhang, Matthew Lentz, and Danyang Zhuo. Mccs: A service-based approach to collective communication for multi-tenant cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 679–690, New York, NY, USA, 2024. Association for Computing Machinery.

[158] Y. Yang and J. Wang. Optimal all-to-all personalized exchange in self-routable multistage networks. *IEEE Transactions on Parallel and Distributed Systems*, 11(3):261–274, 2000.

[159] Stephen Young, Sinan Aksoy, Jesun Firoz, Roberto Gioiosa, Tobias Hagge, Mark Kempton, Juan Escobedo, and Mark Raugas. SpectralFly: Ramanujan graphs as flexible and efficient interconnection networks. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1040–1050, 2022.

[160] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Jason Fantl, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. Efficient Direct-Connect topologies for collective communications. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 705–737, Philadelphia, PA, April 2025. USENIX Association.

[161] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. PyTorch fsdp: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, aug 2023.

[162] Yihao Zhao, Xuanzhe Liu, and Xin Jin. How useful is communication scheduling for distributed training? In *2024 International Scientific and Technical Conference Modern Computer Network Technologies (MoNeTeC)*, pages 1–13, 2024.

[163] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, Carlsbad, CA, July 2022. USENIX Association.

[164] Ziyi Zhu, Min Yee Teh, Zhenguo Wu, Madeleine Strom Glick, Shijia Yan, Maarten Hattink, and Keren Bergman. Distributed deep learning training using silicon photonic switched architectures. *APL Photonics*, 7(3):030901, 2022.

[165] Barret Zoph, Irwan Bello, Sameer Kumar, Nan Du, Yanping Huang, Jeff Dean, Noam Shazeer, and William Fedus. St-moe: Designing stable and transferable sparse expert models. *arXiv preprint arXiv:2202.08906*, 2022.

Appendix A

Efficient Direct-Connect Topologies for Collective Communications

In this appendix, we give additional evaluation results, along with formal mathematical definitions and analyses of the various techniques and concepts discussed in the main text:

- §A.1 provides supplementary materials to evaluation section.
- §A.2 gives formal definitions of reduce-scatter/allgather schedule and how one can be transformed into another.
- §A.3 gives formal definitions of total-hop latency and bandwidth optimality, along with discussions on optimal allreduce schedule and computational cost of reduction.
- §A.4 provides formal definitions of expansion techniques and optimality analysis of their expanded schedules.
- §A.5 provides optimality analysis of BFB schedule generation and discusses variant formulations that support generating schedules for a fixed number of chunks and for heterogeneous network topology.
- §A.6 discusses various generative topologies and the performance of their generated BFB schedules.
- §A.7 provides proofs of all theorems in this paper.
- §A.8 contains supplementary tables and figures. In particular, Table A.3 gives a summary of topologies in this paper.

A.1 Evaluation Appendix

- §A.1.1 presents experiment results that compare BFB schedule generation with communication solutions for switch networks: NCCL [103] and recursive halving & doubling.
- §A.1.2 shows experiment results to validate α-β cost model.

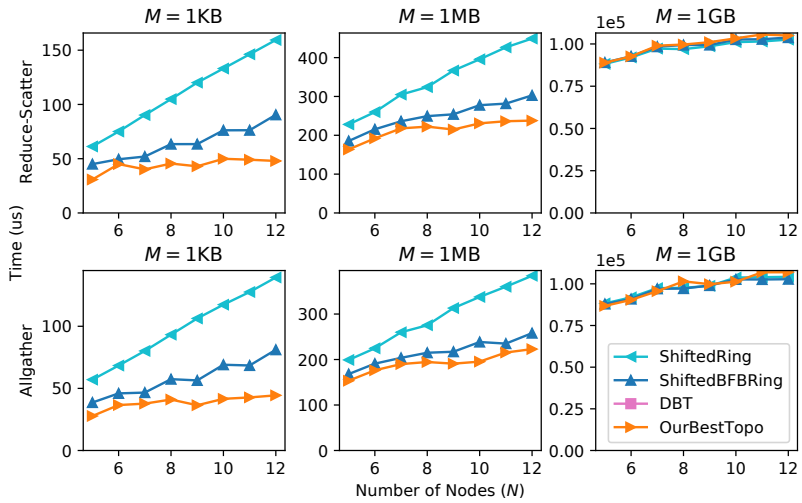


Figure A.1: Comparing reduce-scatter and allgather runtimes of topologies. This figure shows the corresponding reduce-scatter and allgather results of Figure 3.6.

- §A.1.3 gives an analysis of Pareto-efficient topologies/schedules under different hardware and workload specifications.
- §A.1.4 details setup of simulated DNN training and the topologies generated by our topology finder.
- §A.1.5 provides the multi-commodity flow (MCF) formulation used to compute all-to-all throughput.
- §A.1.6 shows how to convert unidirectional topologies/schedules into bidirectional ones.

A.1.1 Comparison Against Switch Solutions

NCCL [103] and recursive halving & doubling (RH&D) are widely adopted collective communication solutions on switch networks. We assess the schedule performance of BFB against these solutions over two direct-connect 8-node topologies: hypercube and twisted hypercube [38]. Hypercube is widely used in HPC settings, and its connections perfectly match the communication pattern of RH&D. Twisted hypercube is a variant of hypercube with a lower diameter.

Figure A.2 compares the baselines against our BFB schedule when run over either hypercube or twisted hypercube with $N=8$, $d=3$ on the testbed. At small M, all schedules and topologies perform roughly the same, except BFB can take advantage of the lower diameter of twisted hypercube and achieve $\sim$20% lower runtime. At large M, because BFB achieves BW optimality on both topologies, it performs even better with 60% lower runtime. RH&D and NCCL perform poorly as M grows because they cannot utilize all $d=3$ links

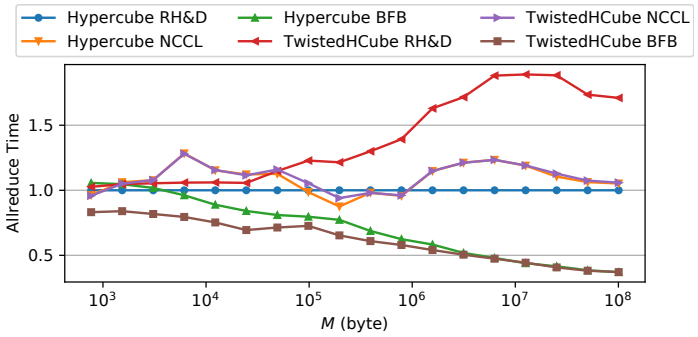


Figure A.2: Comparing switch allreduce solutions (recursive halving & doubling (RH&D), NCCL) against BFB schedule on hypercube and twisted hypercube on $N=8, d=3$ testbed. The runtimes are normalized by the runtime of recursive halving & doubling on hypercube.

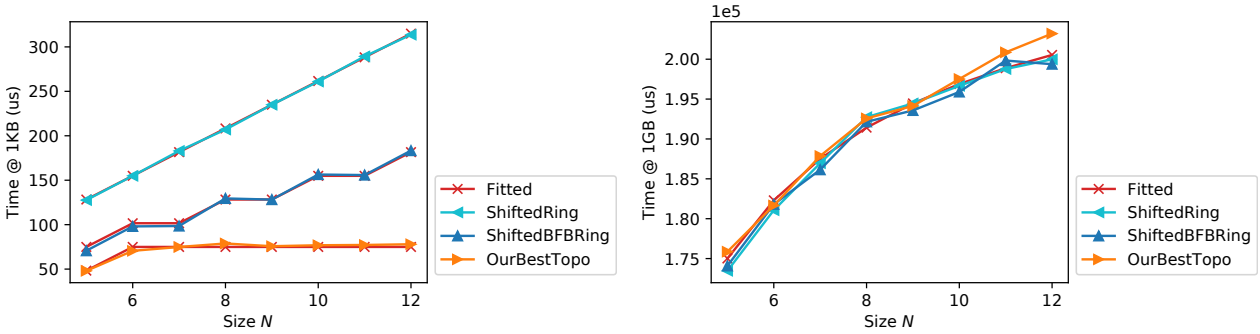
simultaneously. At every comm step of RH&D, a node only communicates with one of the three neighbors, utilizing at most $1/3$ of the total bandwidth (similarly with NCCL). Also, because the schedule is not matched to the twisted hypercube, some nodes communicate with nodes multiple hops away, occupying more links and causing congestion.

A.1.2 Cost Model Validation

Despite the wide acceptance of α-β cost model by previous literature [51, 23, 21, 128, 124], we also conducted a linear regression analysis to validate the cost model on our testbed. In particular, we want to verify that (1) total-hop latency follows $T_L = \alpha \cdot x + \varepsilon$ and (2) BW runtime follows $T_B = \frac{M}{B} \cdot y$, where x and y are the number of comm steps and bandwidth factor respectively ($y = 2 \cdot \frac{N-1}{N}$ if BW-optimal). ε is the constant latency[1] including time costs such as GPU kernel launching. Here, we use linear regression to derive the values of α, ε, and $1/B$, and compute the relative errors between the observed runtimes and expected runtimes. We fit the allreduce runtimes at 1KB to the total-hop latency, since BW runtime is negligible at such a small M. Similarly, we fit the runtimes at 1GB to the BW runtime, since total-hop latency is negligible at such a large M.

Figure A.3 shows our results of linear regression analysis to verify our cost model. For total-hop latency, we obtain estimates $\alpha \approx 13.33$us and $\varepsilon \approx 21.60$us with low errors (average and maximum relative errors of 1.71% and 6.21% respectively). As one can see from Figure A.3a, ShiftedRing and ShiftedBFBRing have a straight and a stair-step shape of runtime growth respectively, which match the expected numbers of comm

[1] This part of latency is a fixed constant for all topologies and schedules, so it is omitted earlier.



(a) Total-Hop Latency T_L (Relative error: avg 1.71%, max 6.21%)

(b) BW Runtime T_B (Relative error: avg 0.47%, max 1.32%)

Figure A.3: Linear regression results.

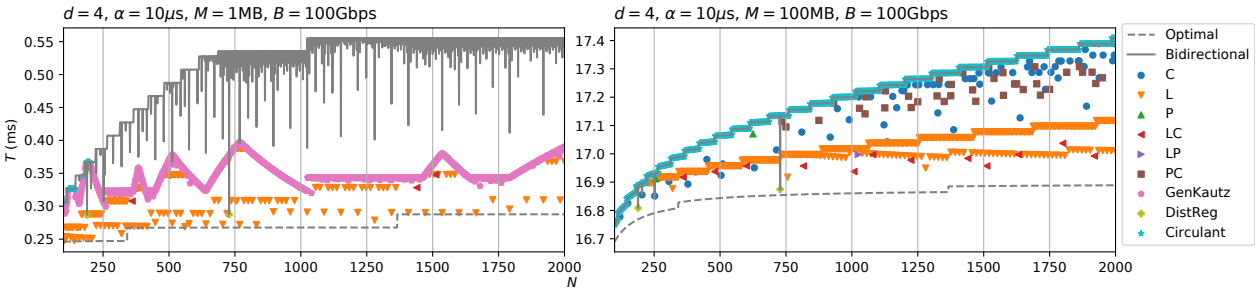


Figure A.4: The minimum allreduce runtimes at different N for $d = 4$, $\alpha = 10$us, and $M/B = 1$MB/100Gbps, 100MB/100Gbps. "L", "P", and "C" stand for line graph, Cartesian power, and Cartesian product (of different graphs) respectively. For example, "LC" means the runtime is achieved by a topology whose construction involves line graph expansion and Cartesian product. "GenKautz", "DistReg", and "Circulant" stand for generalized Kautz graph (§A.6.2), distance regular graph (§A.6.3), and circulant graph (§A.6.4) respectively. The figures also show the best bidirectional topology known at different Ns. Degree expansion does not show up due to target $d = 4$ being relatively small.

steps $2(N-1)$ and $2\lfloor N/2 \rfloor$ respectively. For BW runtime, we get an estimate $1/B \approx 1.018 \times 10^{-4}$us/byte or $B \approx 79$Gbps with low errors (average and maximum relative errors of 0.47% and 1.32% respectively). As one can see from Figure A.3b, all three topologies follow the fitted curve $2\frac{1\text{GB}}{B} \cdot \frac{N-1}{N} = 2T_B^*(N)$ since they are all BW-optimal. However, there is a gap between $B \approx 79$Gbps and the hardware theoretical bandwidth 4x25Gbps=100Gbps. Besides inevitable loss of bandwidth in actual communication, the gap can also be explained by the fact that computational cost of reduction also accounts for part of $1/B$ as discussed in §A.3.4.

A.1.3 Pareto-Efficiency Analysis

There could exist multiple Pareto-efficient topologies at given N and d. For different α and M/B, the Pareto-efficient topology with minimum allreduce runtime is also different. To see how N affects the best choice of

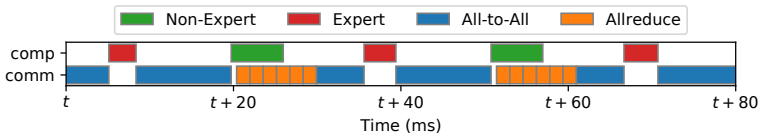


Figure A.5: Example of a training timeline for Switch Transformers.

topologies, we use topology finder (§3.5.4) to generate Pareto-efficient topologies at $d = 4$ for N up to 2000 and pick the best one based on specific values of α and M/B.

Figure A.4 shows two examples of such analysis. At $M = 1$MB, total-hop latency is more important than BW runtime. Thus, we see that generalized Kautz graph is the most popular one, being the best topology at many Ns. On the contrary, at $M = 100$MB, BW performance becomes the dominant factor, and thus circulant graph becomes the most popular one. Line graphs are also popular in both settings; however, line graph expansion requires target N to be divisible by some power of d, so it does not work for any N.

A.1.4 Details of Simulated Distributed Training

We simulate distributed ML training by first collecting actual compute times for model layers, running the models on an NVIDIA A100-SXM-80GB GPU, and then adding communication times according to the specific parallelism, e.g., data or expert parallelism. The communication time is calculated using α-β model for allreduce (§A.1.2) and multi-commodity flow for all-to-all (§A.1.5), assuming $\alpha = 10$us, $B = 100$Gbps over $d = 4$. Our simulation is designed to match the compute-communication overlap pattern of PyTorch Distributed Data Parallel [75]. As in PyTorch DDP, we bucket gradients that are ready for allreduce during backward propagation. Once the gradient volume reaches a predefined bucket capacity, an allreduce is performed. While a large bucket size results in less latency overhead, a small bucket size enhances compute-communication overlap. We choose the best bucket size by comparing the iteration times of bucket sizes {1MB, 10MB, 100MB, 1GB}. To ensure overlap, computation and communication are handled as independent streams, with the communication stream executing one collective at a time.

For the simulated training of Mixture-of-Experts (MoE) models, we follow the standard practice of expert parallelism [40, 119, 70, 73], where experts are sharded across all nodes while non-expert layers are replicated. All-to-all communications are needed before the expert layers to route tokens to the nodes of the assigned experts, and afterward to return tokens to the original nodes for the continuation of the forward/backward pass, thus blocking the computation stream. Furthermore, all-to-all and allreduce are not allowed to be overlapped

as they occupy the same network bandwidth [73]. Figure A.5 shows a timeline example of the simulated expert-parallel training. For simplicity, we assume a uniform token distribution among the experts, as MoE models are trained to balance expert load [165, 130, 40]. Consequently, the all-to-all communication is uniform across the nodes. We use the multi-commodity flow formulation (A.1) to compute the all-to-all communication time.

All hyperparameters, including sequence lengths and global batch sizes, are chosen according to the original paper of Switch Transformers [40]. The topology degree is fixed at 4, and the topology sizes are chosen such that the local batch size at each node is ≥ 1 and not so large as to run out of GPU memory. Table A.1 includes all the Pareto-efficient topologies used in the simulation. For each model and topology size, we choose the topology that results in the smallest iteration time.

A.1.5 All-to-All Throughput

The problem of deriving the throughput of all-to-all communication on a topology can be nicely formulated as a multi-commodity flow (MCF) problem [12, 153, 65, 47, 143]. In an all-to-all MCF, each pair of nodes $(s,t) \in V_G^2$ acts as the source and sink of a commodity. The objective is to simultaneously route f units of flows from each s to t such that f is maximized, with flow allocation subject to flow conservation and edge capacities. In [12], the authors have devised an efficient LP formulation to compute the optimal f:

$$\begin{aligned} \text{maximize} \quad & f \\ \text{subject to} \quad & \sum_s y_{s,(u,v)} \leq 1, && \forall u,v \\ & f + \sum_v y_{s,(u,v)} \leq \sum_w y_{s,(w,u)}, && \forall s,u : s \neq u \\ & y_{s,(u,v)} \geq 0. && \forall s,u,v \end{aligned} \tag{A.1}$$

In LP (A.1), we assume the capacity/bandwidth of each link is 1 unit. Therefore, if the bandwidth of each link is B/d, then fB/d represents the rate at which every node can send to every other node simultaneously.

A.1.6 Unidirectional to Bidirectional

Unidirectional topologies are technically feasible on optical testbeds. The optical cable contains two fibers, one for each direction, and the fabric can link them to two distinct end-hosts, thus enabling unidirectional topologies at no additional hardware cost.

However, in our evaluation, we only use bidirectional topologies. While unidirectional topologies can be realized by configuring the patch panel in simplex mode, the requisite overlay routing for the reverse path

traffic (acks, etc.) is currently only supported using routing rules performed by the host kernel as opposed to the NIC, leading to unpredictable RTTs. Therefore, we can functionally validate unidirectional topologies on our testbed, but we cannot accurately evaluate their performance. Note that newer NICs [19, 104] do support hardware offloading for these rules, which we will examine in future work.

While this paper considers unidirectional topologies a lot, many of the techniques can be conveniently applied to bidirectional topologies as well. For example, BFB schedule generation, degree expansion, and Cartesian product can all be used on bidirectional topologies by replacing each bidirectional edge with two opposite unidirectional edges. The resulting degree expanded and Cartesian product topologies still have unidirectional edges in opposite pairs. Although line graph expansion only works within unidirectional topologies, there is a way to convert unidirectional topology and schedule to bidirectional ones with zero performance sacrifice. In this section, we will show how to convert a reverse-symmetric (see Definition 6) d-regular unidirectional topology G and its allgather schedule A to a $2d$-regular bidirectional topology G' and its schedule A' such that $T_L(A) = T_L(A')$ and $T_B(A) = T_B(A')$.

Let $g : V_G \to V_{G^T}$ be the isomorphism from G to G^T, then it is trivial to see that $g(A)$ (see Definition 7) is an allgather schedule for G^T. Observe that $G' = G \cup G^T$ is a $2d$-regular bidirectional topology. Consider both A and $g(A)$ as allgather schedules for bidirectional topology G'. Schedules A and $g(A)$ use disjoint sets of edges, because they use opposite directions. Thus, we can divide each shard into two halves. Let one half follow schedule A and the other half follow $g(A)$. Let such a schedule be A'.

It is trivial to see that $T_L(A) = T_L(A')$. As for $T_B(A) = T_B(A')$, it follows the fact that the total data size is halved for each of A and $g(A)$, but the bandwidth per edge is also halved due to the doubling of degree. Note that if A is BW-optimal, then A' is BW-optimal; however, A' is not necessarily Moore optimal if A is Moore optimal.

A.2 Reduce-Scatter & Allgather

We use tuple $((v,C),(u,w),t)$ to denote that u sends v's chunk C to w at comm step t. Node v is the source and destination node of chunk C in allgather and reduce-scatter respectively. A communication schedule is thus a collection of tuples.

Definition 4 (Allgather)**.** *An algorithm (G,A) is an allgather algorithm if for arbitrary $x \in S$ and distinct $u,v \in V_G$, there exists a sequence in A:*

$$((v,C_1),(w_0,w_1),t_1),((v,C_2),(w_1,w_2),t_2),\ldots,((v,C_n),(w_{n-1},w_n),t_n),$$

where $w_0 = v$, $w_n = u$, $t_1 < t_2 < \ldots < t_n$, *and* $x \in C_1 \cap C_2 \cap \cdots \cap C_n$.

This sequence serves to broadcast x from v to u. A reduce-scatter algorithm has the same definition except $w_0 = u$, $w_n = v$. In reduce-scatter, we assume any chunk received by a node is immediately reduced with the node's local chunk.

In this paper, many of the techniques are discussed under allgather only. We will show that anything holds in either reduce-scatter or allgather has an equivalent version for the other collective operation. To do so, we use the concept of *transpose graph* from graph theory and define *reverse schedule*. We say a schedule A is *for* topology G if every $((v,C),(u,w),t) \in A$ satisfies $u,v,w \in V_G$ and $(u,w) \in E_G$.

Definition 5 (Reverse Schedule)**.** *Suppose A is a schedule for G. A reverse schedule A^T of A is a schedule for transpose graph G^T such that $((v,C),(u,w),t_{\max} - t + 1) \in A^T$ iff $((v,C),(w,u),t) \in A$, where $t_{\max}$ is the max comm step in A.*

It is trivial to see that $T_L(A) = T_L(A^T)$ and $T_B(A) = T_B(A^T)$. Note that $(u,w) \in E_{G^T}$ if and only if $(w,u) \in E_G$ by definition of transpose graph.

Theorem 1. *If A is a reduce-scatter/allgather schedule for G, then A^T is an allgather/reduce-scatter schedule for G^T.*

Theorem 1 has the following two corollaries:

Corollary 1.1. *Suppose $G \mapsto f(G)$ is a function to construct reduce-scatter/allgather schedule given graph G, then $G \mapsto f(G^T)^T$ is a function to construct allgather/reduce-scatter schedule given graph G.*

Corollary 1.2. *Suppose $(G,A) \mapsto (f(G), f(A))$ is a mapping within reduce-scatter/allgather algorithms, then $(G,A) \mapsto (f(G^T)^T, f(A^T)^T)$ is a mapping within allgather/reduce-scatter algorithms.*

For example, the line graph expansion in §3.5.1 can be seen as a mapping within allgather, and the BFB linear program (3.1) can be seen as a function to construct allgather schedule. Thus, Corollary 1.1 and 1.2 have shown that they both have equivalent versions in reduce-scatter.

In undirected topology, it is well-known that reduce-scatter and allgather are a pair of dual operations such that one can be transformed into another by reversing the communication in schedule [23]. It is similar for directed topology but with extra requirement and more complicated transformation. We define the following property for directed graphs:

Definition 6 (Reverse-Symmetry)**.** *A digraph G is reverse-symmetric if it is isomorphic to its own transpose graph G^T.*

In graph theory, there is a similar concept called *skew-symmetric graph*. Reverse-symmetry is a weaker condition than skew-symmetry.

We define a way to transform the schedule for G into a schedule for G^T based on graph isomorphism:

Definition 7 (Schedule Isomorphism)**.** *Suppose G and G' are isomorphic. Let $f : V_G \to V_{G'}$ be the graph isomorphism and A be a schedule for G, then $f(A)$ is a schedule for G' that $((f(v),C),(f(u),f(w)),t) \in f(A)$ iff $((v,C),(u,w),t) \in A$.*

Theorem 2. *Suppose G is reverse-symmetric. Let G^T be the transpose graph, and let $f : V_{G^T} \to V_G$ be the isomorphism from G^T to G. If (G,A) is a reduce-scatter/allgather algorithm, then $(G, f(A^T))$ is an allgather/reduce-scatter algorithm with $T_L(f(A^T)) = T_L(A)$ and $T_B(f(A^T)) = T_B(A)$.*

Theorem 2 establishes that given any reverse-symmetric topology, if we have either reduce-scatter or allgather, then we can construct both reduce-scatter and allgather. Since allreduce can be achieved by applying a reduce-scatter followed by an allgather, we only need one of reduce-scatter and allgather to construct a complete allreduce algorithm. Furthermore, if the reduce-scatter or allgather algorithm has runtime T, then the resulting allreduce algorithm has runtime $2T$.

Most of our base topologies are reverse-symmetric (Table A.3). In addition, all of our expansion techniques also preserve reverse-symmetry. Thus, one can almost always use Theorem 2 to derive reduce-scatter and allreduce schedules from allgather schedule on our synthesized topologies. For non-reverse-symmetric topologies like generalized Kautz graph, one can apply Corollary 1.1 or 1.2 to construct reduce-scatter and allgather separately.

A.3 Topology-Schedule Optimality

Because our cost model is only concerned with total-hop latency and BW runtime, the optimality of reduce-scatter/allgather algorithm is only related to total-hop latency optimality and BW optimality in this paper. Note that we also consider topology as a dimension that can be optimized, so optimality is discussed in the space of all topology-schedule combinations, i.e., *algorithms* by our definition.

A.3.1 Total-Hop Latency Optimality

Definition 8 (Total-Hop Latency Optimal)**.** *Given an N-node degree-d reduce-scatter/allgather algorithm (G,A), if any other N-node degree-d reduce-scatter/allgather algorithm (G',A') satisfies $T_L(A') \geq T_L(A)$, then (G,A) is total-hop latency optimal.*

Because in reduce-scatter/allgather, every node needs to send a shard of data to every other node, the number of comm steps is lower bounded by the graph diameter:

Theorem 3. *Every reduce-scatter/allgather algorithm (G,A) satisfies $T_L(A) \geq \alpha \cdot D(G)$, where $D(G)$ is the diameter of G.*

Because we can always construct a BFB schedule A for topology G with $T_L(A) = \alpha \cdot D(G)$, it follows the corollary:

Corollary 3.1. *An N-node degree-d reduce-scatter/allgather algorithm (G,A) is total-hop latency optimal if and only if*

$$T_L(A) = \alpha \cdot D(G) = \alpha \cdot \min\{D(G') : |V_{G'}| = N, \deg(G') = d\}.$$

The minimum diameter of a directed graph given a number of vertices and degree is still an open question. One can check *degree/diameter problem* [96] for more information. However, as a close upper bound of number of vertices given degree and diameter, the *Moore bound* for digraph is sufficient to tell the total-hop latency optimality in most cases.

Definition 9 (Moore Bound)**.** *Let G be any degree-d digraph of diameter k. The Moore bound is an upper bound on the number of vertices in G:*

$$M_{d,k} = \sum_{i=0}^{k} d^i = \frac{d^{k+1}-1}{d-1}.$$

Definition 10 (Moore Optimal)**.** *Let (G,A) be an N-node degree-d reduce-scatter/allgather algorithm with $T_L(A) = k\alpha$, then (G,A) is Moore optimal if $N > M_{d,k-1}$.*

Because for any degree-d digraph G, $D(G) \geq k$ must be true as long as $|V_G| > M_{d,k-1}$, Moore optimality is a stronger condition than total-hop latency optimality. We define a function T_L^* such that $T_L^*(N,d)$ equals the Moore optimal total-hop latency of N-node degree-d reduce-scatter/allgather algorithms.

A.3.2 Bandwidth Optimality

Definition 11 (Bandwidth Optimal)**.** *Given an N-node degree-d reduce-scatter/allgather algorithm (G,A), if any other N-node degree-d reduce-scatter/allgather algorithm (G',A') satisfies $T_B(A') \geq T_B(A)$, then (G,A) is BW-optimal.*

In reduce-scatter/allgather, each node needs to send/receive at least $M \cdot \frac{N-1}{N}$ amount of data. Thus, the following holds:

Theorem 4. *$\frac{M}{B} \cdot \frac{N-1}{N}$ is a lower bound of $T_B(A)$ for any N-node reduce-scatter/allgather algorithm (G,A).*

Note that one can always construct a ring of degree d by sending d parallel edges from one node to the next node. The trivial ring reduce-scatter/allgather schedule has $\frac{M}{B} \cdot \frac{N-1}{N}$ BW runtime. Therefore, we have:

Corollary 4.1. *An N-node reduce-scatter/allgather algorithm (G,A) is BW-optimal if and only if $T_B(A) = \frac{M}{B} \cdot \frac{N-1}{N}$.*

We define a function T_B^* such that $T_B^*(N) = \frac{M}{B} \cdot \frac{N-1}{N}$ is the optimal BW runtime of N-node reduce-scatter/allgather algorithms. From Corollary 4.1, we have the following necessary and sufficient condition for BW optimality:

Theorem 5. *An allgather algorithm (G,A) is BW-optimal if and only if:*

1. *$\frac{1}{B/d}\sum_{((v,C),(u,w))\in A_t} |C| = T_B(A_t)$ for all $(u,v) \in E_G$ and $t \in \{1,\ldots,t_{\max}\}$. A_t is the subschedule of A at comm step t.*
2. *Pick any distinct $u,v \in V_G$. For each $x \in S$, there exists a unique $((v,C),(w,u),t) \in A$ such that $x \in C$.*

Condition 1 ensures that at each comm step, every link of topology G has equal workload, so no link finishes early and results in waste of bandwidth. Condition 2 ensures that no piece of data is received twice by some node, so no duplicated send exists.

A.3.3 Allreduce Optimality

In this paper, we construct an allreduce algorithm through a reduce-scatter followed by allgather. In such construction, the lower bound of allreduce algorithm is $2(T_L^*(N,d) + T_B^*(N))$. To compare this with the lower bound of any allreduce construction, in [110], the authors have proved that $2T_B^*(N)$ is indeed the lower bound of BW runtime of any allreduce algorithm. As for total-hop latency, a reduce-scatter followed by allgather has

at least $2D(G)$ number of comm steps, so $2T_L^*(N,d)$ is the lower bound of total-hop latency. Although one can use all-to-all to construct an allreduce with number of comm steps equal to one diameter $D(G)$ (lower bound being $T_L^*(N,d)$ instead of $2T_L^*(N,d)$), the lower bound of BW runtime for all-to-all is $\frac{M}{B}\cdot(N-1) = N\cdot T_B^*(N)$, which is much worse than $2T_B^*(N)$.

There is also another way of constructing allreduce: reduce followed by broadcast. In such an approach, the number of comm steps can be twice the radius of G instead of twice the diameter. However, the Moore bound for graph diameter also applies to graph radius, so $2T_L^*(N,d)$ is still a lower bound of allreduce via reduce+broadcast. By Theorem 16, the total-hop latency optimal allreduce via reduce+broadcast is at most 2α lower than the total-hop latency of generalized Kautz graph can do with reduce-scatter plus allgather. Furthermore, reduce+broadcast is usually poor in BW performance.

A.3.4 Computational Cost

In this paper, we omit the computational cost of reduction operation in performance analysis. While this approach is commonly adopted in previous literature [145, 21, 128, 124], we give a formal reasoning why this approach is legitimate. It is not only because computational cost is generally orders of magnitude lower than network cost, but also because computational cost can be incorporated into network cost.

Assume a cost model where computation and network communication do not overlap at each node.[2] In particular, at each comm step of reduce-scatter, the computation to reduce chunks happens immediately after the node receives all chunks and before the node starts to send out chunks for the next comm step. We adopt notations from [23], where γ denotes the computational time cost per size of data. Like total-hop latency and BW runtime, we also let $T_C(A)$ be the total time spent on computation by schedule A. As argued in [23], a lower bound of computational cost is $T_C \geq M\cdot\gamma\cdot\frac{N-1}{N}$ for both reduce-scatter and allreduce, which is identical to the BW optimality of reduce-scatter and half of that of allreduce. The following theorem shows that BW runtime of a schedule can act as an upper bound for the computational time.

Theorem 6. *Given a reduce-scatter algorithm (G,A), suppose $T_B(A) = \frac{M}{B}\cdot y$, then $T_C(A) \leq M\cdot\gamma\cdot y$.*

The rationale behind Theorem 6 is that the amount of computation for any node at a given comm step equals the amount of data the node receives during that comm step. Thus, **as we balance network transmission, it naturally leads to a more balanced computation.** With Theorem 6, if the BW runtime of

[2] Otherwise, the computational cost would be even more negligible.

some allreduce schedule A is $T_B(A) = 2\frac{M}{B} \cdot y$, then $T_B(A) + T_C(A) \leq M \cdot (\frac{2}{B} + \gamma) \cdot y$. We can thus simply define $B' = (\frac{1}{B} + \frac{\gamma}{2})^{-1}$, and then $2\frac{M}{B'} \cdot y$ can represent the sum of BW runtime and computational runtime altogether. The value of y is all that matters. The following corollary shows that if an algorithm is BW-optimal, then such representation is exact.

Corollary 6.1. *If allreduce algorithm (G,A) is BW-optimal, i.e., $T_B(A) = 2\frac{M}{B} \cdot \frac{N-1}{N}$, then $T_C(A) = M \cdot \gamma \cdot \frac{N-1}{N}$ and $T_B(A) + T_C(A) = 2M \cdot (\frac{1}{B} + \frac{\gamma}{2}) \cdot \frac{N-1}{N}$.*

When profiling a testbed, one can simply derive the value of $\frac{1}{B} + \frac{\gamma}{2}$ using BW-optimal topologies and use it as the new $1/B$ to apply the results of this paper. While it is still possible for two schedules with the same BW runtime to have different computational runtimes, such difference is bounded by the aforementioned theorems and orders of magnitude smaller than BW runtime.

A.4 Optimality of Expansion Techniques

In this section, we provide formal definitions and detailed performance analysis of expansion techniques.

A.4.1 Line Graph Expansion

Definition 12 (Line Graph)**.** *Given a directed graph (or digraph) G, each edge $(u,v) \in E_G$ corresponds to a vertex uv in the line graph $L(G)$. For every uv, vw pair in $V_{L(G)}$, there exists an edge $(uv, vw) \in E_{L(G)}$.*

In the case of multiedges between u and v, the line graph also contains multiple vertex uv.

Definition 1 (Schedule of Line Graph)**.** *Given an allgather schedule A_G for topology G, let $A_{L(G)}$ be the schedule for line graph $L(G)$ containing:*

1. *$((v'v, S), (v'v, vu), 1)$ for each edge $(v'v, vu) \in E_{L(G)}$ with $v'v \neq vu$.* **[Insert the 1st comm step in $A_{L(G)}$.]**
2. *$((v'v, C), (uw, ww'), t+1)$ for each $((v,C),(u,w),t) \in A_G$ and $v'v \neq ww'$.* **[Adapt A_G to form $A_{L(G)}$.]**

The following theorem gives the performance of the expanded schedule:

Theorem 7. *Given a d-regular topology G, if (G,A_G) is an N-node allgather algorithm, then $(L(G), A_{L(G)})$ is a dN-node allgather algorithm satisfying:*

$$T_L(A_{L(G)}) = T_L(A_G) + \alpha, \tag{A.2}$$

$$T_B(A_{L(G)}) \leq T_B(A_G) + \frac{M}{B} \cdot \frac{1}{N}. \tag{A.3}$$

From Theorem 7, one can see that the performance of the expanded schedule depends on that of the base schedule. Note that T_B also depends on M and B. For simplicity, we write $T_B(A_G)$ instead of $T_B(A_G, M, B)$ when there is no ambiguity. Theorem 7 makes an implicit assumption that $T_B(A_G, M, B) = \tau(M/B)$ for some constant τ. This assumption, suggesting that T_B scales linearly with data size and inversely with bandwidth, should hold for any reasonably designed schedule.

Consequently, if we apply line graph expansion n times, the performance of the expanded schedule is:

Corollary 7.1. *Given a d-regular topology G, if (G, A_G) is an N-node allgather algorithm with $T_B(A_G, M, B) = \tau(M/B)$ for some constant τ, then $(L^n(G), A_{L^n(G)})$ is a d^nN-node allgather algorithm satisfying:*

$$T_L(A_{L^n(G)}) = T_L(A_G) + n\alpha, \tag{A.4}$$

$$T_B(A_{L^n(G)}) \leq T_B(A_G) + \frac{M}{B} \cdot \frac{d}{d-1} \left(\frac{1}{N} - \frac{1}{d^n N} \right). \tag{A.5}$$

In terms of the optimality of line graph expansion:

Theorem 8. *$(L^n(G), A_{L^n(G)})$ is Moore optimal if and only if (G, A_G) is Moore optimal.*

Theorem 9. *If (G, A_G) is BW-optimal with N nodes, then $T_B(A_{L^n(G)})/T_B^*(d^nN) \leq 1 + [(d-1)N]^{-1}$ for all n.*

As mentioned in the main text, by Theorem 9, the key metric for the quality of base graph is how large it is while achieving both Moore and BW optimality. Currently, our largest such base graph that works for any even degree is Hamming graph $H(2, 1+d/2)$, which has $(1+d/2)^2 = \Theta(d^2)$ number of nodes. The corresponding line graph expanded topology is always Moore optimal and at most $O(1/d^3)$ away from BW optimality by Theorem 9.

Line graph expansion is closely related to BFB schedule for two reasons: (1) most of our base topologies like complete bipartite graph and Hamming graph use BFB schedule as the base schedule, and (2) the line graph expansion of BFB schedule is still a BFB schedule. To see the performance bound in Theorem 7 is tight, we have the following results in the context of BFB schedule:

Theorem 10. *Let A_G be a BFB allgather schedule for d-regular topology G with $|N^+(u)| > 1$ for all $u \in V_G$, then the expanded schedule $A_{L(G)}$ is a BFB allgather schedule for $L(G)$. In particular, if A_G is the optimal BFB schedule for G, then $A_{L(G)}$ is the optimal BFB schedule for $L(G)$ satisfying:*

$$T_B(A_{L(G)}) = T_B(A_G) + \frac{M}{B} \cdot \frac{1}{N}. \tag{A.6}$$

Corollary 10.1. *Let A_G be a BFB allgather schedule for d-regular topology G with $|N^+(u)|>1$ for all $u \in V_G$, then the expanded schedule $A_{L^n(G)}$ is a BFB allgather schedule for $L^n(G)$. In particular, if A_G is the optimal BFB schedule for G, then $A_{L^n(G)}$ is the optimal BFB schedule for $L^n(G)$ satisfying:*

$$T_B(A_{L^n(G)}) = T_B(A_G) + \frac{M}{B} \cdot \frac{d}{d-1}\left(\frac{1}{N} - \frac{1}{d^n N}\right).$$

A.4.2 Degree Expansion

Definition 13 (Degree Expanded Topology)**.** *Given an N-node d-regular topology G without self-loops, construct the degree expanded nN-node nd-regular topology $G*n$:*

1. *For each vertex $v \in V_G$, add $v_1, \ldots, v_n$ to V_{G*n},*
2. *For each edge $(u,v) \in E_G$, add (u_i, v_j) to E_{G*n} for all i, j including $i = j$.*

Definition 2 (Degree Expanded Schedule)**.** *Given an allgather schedule A_G for G, construct A_{G*n} for $G*n$:*

1. *For all i,j including $i=j$ and for each $((v,C),(u,w),t) \in A_G$, add $((v_j,C),(u_j,w_i),t)$ to A_{G*n};*
2. *Divide shard S into equal-sized chunks $C_1, \ldots, C_{nd}$. Given $u_i, u_j \in V_{G*n}$ with $i \neq j$, add $((u_i,C_\alpha),(v_\alpha,u_j),t_{\max}+1)$ to A_{G*n} for each $(v_1,u_j),\ldots,(v_{nd},u_j) \in E_{G*n}$, where $t_{\max}$ is the max comm step in A_G.*

Theorem 11. *Given a d-regular topology G without self loops, if (G,A_G) is an N-node allgather algorithm with $T_B(A_G,M,B) = \tau(M/B)$ for some constant τ, then $(G*n, A_{G*n})$ is an nN-node allgather algorithm satisfying:*

$$T_L(A_{G*n}) = T_L(A_G) + \alpha, \tag{A.7}$$

$$T_B(A_{G*n}) = T_B(A_G) + \frac{M}{B} \cdot \frac{n-1}{nN}. \tag{A.8}$$

Corollary 11.1. *If (G,A_G) is BW-optimal and $T_B(A_G,M,B) = \tau(M/B)$ for some τ, then $(G*n, A_{G*n})$ is BW-optimal.*

Degree expansion preserves BW optimality. As for total-hop latency of degree expanded topology, observe that $T_L^*(N,d) = \Theta(\log_d N)$ and $\log_{nd} nN < \log_d N$, so T_L^* decreases as we apply degree expansion. Since T_L increases in degree expansion, Moore optimality is not preserved.

A.4.3 Cartesian Product Expansion

Definition 3 (Cartesian Product)**.** *The Cartesian product digraph $G_1 \square G_2$ of digraphs G_1 and G_2 has vertex set $V_{G_1} \times V_{G_2}$ with vertex $\mathbf{u} = (u_1, u_2)$ connected to $\mathbf{v} = (v_1, v_2)$ iff either $(u_1, v_1) \in E_{G_1}$ and $u_2 = v_2$; or $u_1 = v_1$ and $(u_2, v_2) \in E_{G_2}$.*

Definition 3 generalizes to Cartesian product of multiple digraphs: $G_1 \square G_2 \square G_3 = (G_1 \square G_2) \square G_3$. The Cartesian product of n identical digraphs is denoted as Cartesian power $G^{\square n}$.

Definition 14 (Schedule of Cartesian Power)**.** *Given an allgather schedule A_G for topology G and $n \in \mathbb{N}$, construct the schedule $A_{G^{\square n}}$ for $G^{\square n}$:*

1. *Construct the schedule $A^{(1)}$ as follows:*
2. *For $j = 1, \ldots, n$, for each $((w, C), (u, v), t) \in A_G$, add*

$$(((\mathbf{x}, w, \mathbf{z}), C), ((\mathbf{y}, u, \mathbf{z}), (\mathbf{y}, v, \mathbf{z})), t + (j-1)t_{\max})$$

   to $A^{(1)}$ for all $\mathbf{x}, \mathbf{y} \in V_G^{j-1}$ and $\mathbf{z} \in V_G^{n-j}$. $t_{\max}$ is the max comm step in A_G.
3. *Similarly, construct $A^{(i)}$ for $i = 2, \ldots, n$ that each vertex $\mathbf{v}$ in $A^{(1)}$ is shifted by $i-1$ to $(\mathbf{v}[n-i+2:n], \mathbf{v}[1:n-i+1])$.*
4. *Divide each shard into n equal-sized subshards. Construct schedule $A_{G^{\square n}}$ such that $A^{(i)}$ performs allgather over the i-th subshards of all nodes.*

Theorem 12. *Given a d-regular topology G, if (G, A_G) is an N-node allgather algorithm with $T_B(A_G, M, B) = \tau(M/B)$ for some constant τ, then $G^{\square n}$ is an nd-regular topology, and $(G^{\square n}, A_{G^{\square n}})$ is an N^n-node allgather algorithm satisfying:*

$$T_L(A_{G^{\square n}}) = n \cdot T_L(A_G), \tag{A.9}$$

$$T_B(A_{G^{\square n}}) = T_B(A_G) \cdot \frac{N}{N-1} \cdot \frac{N^n - 1}{N^n}. \tag{A.10}$$

We then have the following corollary:

Corollary 12.1. *If (G, A_G) is BW-optimal and $T_B(A_G, M, B) = \tau(M/B)$ for some τ, then $(G^{\square n}, A_{G^{\square n}})$ is BW-optimal.*

Like degree expansion, Cartesian power expansion does not preserve Moore optimality.

We use BFB schedule generation when dealing with Cartesian product of distinct topologies:

Theorem 13. *Let $G_1, G_2, \ldots, G_n$ be topologies that*

1. *$G_1, \ldots, G_n$ are nontrivial simple digraphs;*
2. *Every G_i has BW-optimal BFB allgather schedule.*

Then, the optimal BFB allgather schedule, i.e. the schedule generated by BFB LP (3.1), for $G_1 \Box \ldots \Box G_n$ is also BW-optimal. The total-hop latency of the schedule equals $\alpha \cdot D(G_1 \Box \ldots \Box G_n) = \alpha \cdot \sum_i D(G_i)$.

The BFB schedule generation can also be used when individual topologies do not have BW-optimal BFB schedules; however, in such a case, we do not have performance bound for the schedule of the Cartesian product.

A.5 BFB Schedule Generation

The LP formulation for u_2 in Figure 3.5 is:

$$
\begin{aligned}
\text{minimize} \quad & U_{u_2,t} \\
\text{subject to} \quad & x_{v_1,(w_1,u_2),t} \le U_{u_2,t}, \\
& x_{v_1,(w_2,u_2),t} + x_{v_2,(w_2,u_2),t} \le U_{u_2,t}, \\
& x_{v_2,(w_3,u_2),t} \le U_{u_2,t}, \\
& x_{v_1,(w_1,u_2),t} + x_{v_1,(w_2,u_2),t} = 1, \\
& x_{v_2,(w_2,u_2),t} + x_{v_2,(w_3,u_2),t} = 1, \\
& 0 \le x_{v,(w,u_2),t} \le 1. \qquad \forall v, w
\end{aligned}
$$

Definition 15 (BFB schedule)**.** *An allgather schedule A for G is a BFB schedule if A satisfies: $((v,C),(w,u),t) \in A$ only if $d(v,u) = d(v,w) + 1 = t$.*

Theorem 14. *A schedule A for G is a BFB allgather schedule if and only if the following are satisfied:*

1. *If $((v,C),(w,u),t) \in A$, then $d(v,u) = d(v,w) + 1 = t$;*
2. *For any distinct $u, v \in V_G$, the collection of chunks $\mathcal{C}_v = \{C \mid ((v,C),(w,u),t) \in A\}$ satisfies $S = \bigcup_{C \in \mathcal{C}_v} C$.*

Condition 1 ensures the schedule follows the breadth-first broadcast order. Condition 2 ensures every node receives the entire shard from every other node and thus a valid allgather.

A.5.1 Optimality

Theorem 15. *If A is a BFB schedule for G, then the total-hop latency $T_L(A) = \alpha \cdot D(G)$.*

There may exist many BFB schedules for a given topology G. They all have the same T_L but may have different T_Bs. Thus, the optimal BFB schedule is the one with the lowest T_B. Since every BFB schedule can be expressed as a solution to linear program (3.1), we have the following result:

Theorem 16. *Given any topology G, linear program (3.1) gives the optimal BFB schedule of G.*

An important implication of Theorem 16 is that **if we can show a BW-optimal BFB schedule exists for a topology G, then linear program (3.1) is guaranteed to generate one.** This has become an important tool for us to prove that BFB schedule generation can always generate BW-optimal schedules for some families of topologies (see §A.6). For the rest of this section, we show conditions that, if met by a topology, ensure it has a BW-optimal BFB schedule.

The following theorem shows the necessary and sufficient conditions for a BFB allgather schedule to be BW-optimal:

Theorem 17. *Suppose (G,A) is a BFB allgather schedule. (G,A) is BW-optimal if and only if:*

1. *There exists a sequence $N_1^-, N_2^-, \ldots, N_{D(G)}^- \in \mathbb{N}$ such that for any $x \in \mathbb{N}$ and $u \in V_G$, $|N_x^-(u)| = N_x^-$.*
2. *For any $(w,u) \in E_G$, $\sum_{((v,C),(w,u)) \in A_t} |C| = \frac{M}{N}|N_t^-(u)|/d = \frac{M}{N}N_t^-/d$.*

We assume G is d-regular. Condition 1 and 2 together ensure that at each comm step, all links have perfectly balanced workloads. In Theorem 13, we have already proven that *a Cartesian product graph has BW-optimal BFB schedule if it is the product of graphs that each have a BW-optimal BFB schedule.* Here, Theorem 17 also leads to the following sufficient condition for a bidirectional topology to have a BW-optimal BFB schedule:

Theorem 18. *There exists a BW-optimal BFB schedule for undirected graph G if for every distance x, two of the following constants exist:*

1. *$N_x = |N_x(u)|$ for any $u \in V_G$;*
2. *$a_x = |N_x(u) \cap N_{x-1}(w)|$ for any $u \in V_G$ and $w \in N(u)$;*
3. *$b_x = |N(u) \cap N_{x-1}(v)|$ for any $u \in V_G$ and $v \in N_x(u)$.*

Moreover, if two of N_x, a_x, b_x exist, then the third one must also exist with $N_x = da_x/b_x$.

Note that in undirected graphs, we have $N_x^+(u) = N_x^-(u) = N_x(u)$. To understand these constants, N_x is the number of data shards u needs to receive at comm step x; a_x is the number of data shards that can be transmitted by each link (w,u) at comm step x; b_x is the number of link (w,u)s that each data shard can use to transmit the data to u at comm step x. These three constants collectively ensure that links are perfectly balanced with each link transmitting $\frac{M}{N}N_x/d = \frac{M}{N}a_x/b_x$ amount of data at comm step x.

Now, we give a *necessary and sufficient condition* for any topology to have a BW-optimal BFB schedule. The condition is derived based on the observation that the BFB optimization problem is equivalent to a job scheduling problem. In each comm step t, for each node u, we have a set of jobs $\{j_1, j_2, \ldots, j_m\}$ (data from the source nodes $v \in N_t^-(u)$) and a set of processors $\{p_1, p_2, \ldots, p_d\}$ (links from in-neighbors $w \in N^-(u)$). There exists a map f from any job to a set of processors that j_i can only be scheduled to the processors in $f(j_i)$ (in-neighbor ws satisfying $d(v,u) = d(v,w) + 1 = t$). Assuming jobs can be arbitrarily divided into subjobs for parallel execution on multiple processors, the problem is how to schedule these jobs to processors so that workloads are balanced across all processors. We have the following result:

Theorem 19. *The workloads can be balanced if and only if there exists no subset $J \subseteq \{j_1, j_2, \ldots, j_m\}$ such that*

$$\frac{|J|}{\left|\bigcup_{j \in J} f(j)\right|} > \frac{m}{d}.$$

Note that there is an independent scheduling problem for each comm step t and node u. Therefore, topology G has a BW-optimal BFB schedule if and only if:

1. At each comm step t, $|N_t^-(u)|$ is the same for all $u \in V_G$.
2. The scheduling problem w.r.t. each t and u satisfies the condition in Theorem 19.

A.5.2 Discrete Chunked BFB Schedule

The BFB LP (3.1) makes an assumption that shards can be divided arbitrarily and infinitesimally. However, to compile the schedule into an executable form, one may need a *discrete chunked schedule*, where each shard is divided into a fixed number of equal-sized chunks. In practice, $x_{v,(w,u),t}$s are usually solved to be rational numbers. We can divide each shard into a number of chunks equal to the LCM of $x_{v,(w,u),t}$s' denominators so that each $x_{v,(w,u),t}$ represents some integer number of chunks. This approach has worked for us in evaluations. However, there exists the case where each shard of the data can only be divided into P equal chunks (i.e., the

whole data M can only be divided into PN equal chunks). In such a case, we show that *we can approximate the optimal discrete chunked BFB schedule in polynomial time*.

Consider the following integer program given u,t:

$$\begin{aligned} \min \quad & W_{u,t} \\ \text{s.t.} \quad & \sum_v y_{v,(w,u),t} \le W_{u,t}, \quad \forall w \in N^-(u) \\ & \sum_w y_{v,(w,u),t} = P, \quad \forall v \in N_t^-(u) \\ & y_{v,(w,u),t} \in \{0,1,\ldots,P\}, \quad \forall w,v. \end{aligned} \tag{A.11}$$

Compared with (3.1), one can easily see that the optimal solution of (A.11) gives the optimal BFB allgather schedule when each shard of the data can only be divided into P chunks. One can also easily solve the LP relaxation of (A.11) in polynomial time. Let T_B^{OPT} be the optimal BW runtime of the schedule obtained by directly solving integer program (A.11). Suppose the LP relaxation gives a schedule with BW runtime T_B^{LP}, then it holds that $T_B^{\text{LP}} \le T_B^{\text{OPT}}$.

Let $y^{\text{LP}}_{v,(w,u),t}$s be the solution to the LP relaxation of (A.11). We can obtain an integer solution $y_{v,(w,u),t}$s of (A.11) by rounding $y^{\text{LP}}_{v,(w,u),t}$s up or down to integers. For each v, we have

$$\sum_w \left\lfloor y^{\text{LP}}_{v,(w,u),t} \right\rfloor \le P \le \sum_w \left\lceil y^{\text{LP}}_{v,(w,u),t} \right\rceil.$$

Thus, it is trivial to round $y^{\text{LP}}_{v,(w,u),t}$s to integer $y_{v,(w,u),t}$s that $\sum_w y_{v,(w,u),t} = P$ and $y_{v,(w,u),t} < y^{\text{LP}}_{v,(w,u),t} + 1$. We give the following approximation bound for the resulting schedule:

Theorem 20. *Rounding LP gives a solution with BW runtime* $T_B \le T_B^{OPT} + \frac{M}{B} \cdot \frac{d(d^{D(G)}-1)}{(d-1)PN}$. *In addition, if topology* G *is Moore optimal, then* $T_B \le T_B^{OPT} + \frac{M}{B} \cdot \frac{d}{P}$.

The cost $\frac{M}{B} \cdot \frac{d}{P}$ is negligible since P can easily be hundreds or even thousands while degree d is usually a small integer.

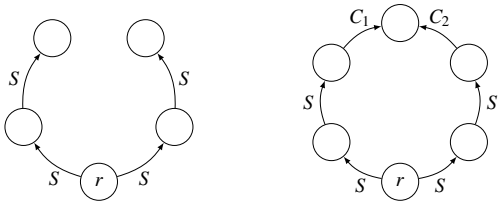


Figure A.6: The broadcast paths of ring BFB allgather schedule. The left and right figures respectively show the broadcast patterns for odd- and even-sized bidirectional rings. Edges of the rings are omitted. C_1 and C_2 are two halves of shard S.

A.5.3 Heterogeneous BFB Schedule

The BFB LP (3.1) assumes a homogeneous network. It turns out that with little modification, (3.1) can become an LP for heterogeneous network too:

$$\begin{aligned} \min \quad & U_{u,t} \\ \text{s.t.} \quad & \alpha_{w,u} + \tfrac{M/N}{B_{w,u}} \sum_v x_{v,(w,u),t} \le U_{u,t}, \quad \forall w \in N^-(u) \\ & \sum_w x_{v,(w,u),t} = 1, \quad \forall v \in N_t^-(u) \\ & 0 \le x_{v,(w,u),t} \le 1, \quad \forall w, v. \end{aligned} \tag{A.12}$$

$\alpha_{w,u}$ and $B_{w,u}$ are the hop latency and bandwidth of link (w,u). In some cases, the α of some link (w,u) is so high that $\alpha_{w,u}$ alone dominates $U_{u,t}$ in (A.12) even though $\sum_v x_{v,(w,u),t} = 0$. This is problematic because one should not pay $\alpha_{w,u}$ if link (w,u) is not used. However, such a scenario can be easily detected after solving LP (A.12). One can avoid the issue by simply removing link (w,u) and solving the LP again.

A.6 Generative Topologies

In this section, we introduce several topologies for which applying BFB schedule generation yields high-performance communication schedules.

A.6.1 Bidirectional Ring

Ring is the most common topology for allreduce. The traditional schedule on ring is to make each shard go a full circle to do reduce-scatter/allgather. In a bidirectional ring, one can simply make half the shard go clockwise and the other half go counterclockwise to utilize both directions of the links. Such a reduce-scatter/allgather schedule is BW-optimal but poor in total-hop latency with $T_L = (N-1)\alpha$. With BFB schedule generation, we discovered a new ring reduce-scatter/allgather schedule that achieves half the total-hop latency ($T_L = \lfloor N/2 \rfloor \alpha$) while maintaining BW optimality. From each node, the BFB allgather schedule broadcasts the

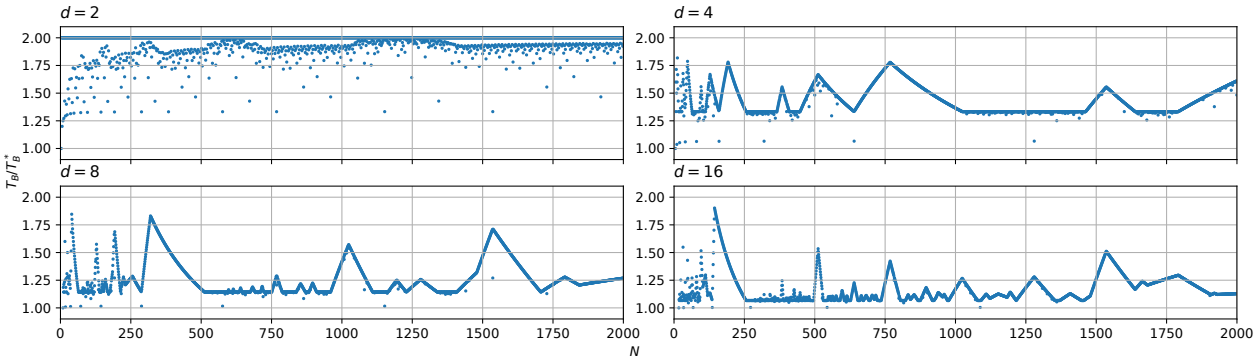


Figure A.7: T_B/T_B^* of generalized Kautz graph $\Pi_{d,N}$ up to $N = 2000$. As shown, the BW runtime of $\Pi_{d,N}$ is less than or equal to $2T_B^*$ at all times for $d = 2,4,8,16$. In particular, the higher the degree is, the closer T_B is to optimal. As for total-hop latency, Theorem 21 shows that $T_L \leq T_L^*(N,d) + \alpha$.

entire shard clockwise and counterclockwise in parallel. Thus, each direction only needs to go half a circle instead of a full circle. If N is even, then the farthest node across the ring receives each half of the shard from each of its two neighbors in the end. Figure A.6 shows examples in odd- and even-sized rings respectively.

A.6.2 Generalized Kautz Graph

Generalized Kautz graph [54, 15] is a low-T_L unidirectional topology that can be constructed for every N and d.

Definition 16 (Generalized Kautz Graph)**.** *The $\Pi_{d,m}$ digraph has the set of integers modulo m as vertex set. Its arc set A is defined as follows:*

$$A = \{(x,y) \mid y \equiv -dx - a, 1 \leq a \leq d\}.$$

If $m = d^{n+1} + d^n$, then $\Pi_{d,m} = K(d,n)$, where $K(d,n)$ is the Kautz graph $L^n(K_{d+1})$.

We apply BFB schedule generation to generalized Kautz graph. The resulting schedule is not always Moore optimal, but the following theorem shows that it is at most one α away from Moore optimality, i.e., $T_L \leq T_L^*(N,d) + \alpha$:

Theorem 21. *Suppose $D(\Pi_{d,m}) = k$, then $m > M_{d,k-2}$.*

Remember Moore optimality is stricter than total-hop latency optimality, so it is possible that generalized Kautz graph is total-hop latency optimal. The special case, Kautz graph $K(d,n)$, is always Moore optimal and is, in fact, the largest known digraph in *degree/diamter problem* for any degree $d > 2$ [96].

As for BW performance, from Figure A.7, one can see that generalized Kautz graph is also close to BW optimality, especially at higher degrees.

A.6.3 Distance-Regular Graph

In graph theory, distance-regular graphs are a family of highly symmetric undirected graphs. We can show that there exists a BW-optimal BFB schedule for any distance-regular graph, and thus LP (3.1) can always generate one. We borrow the following definition from [9]:

Definition 17 (Distance-Regular Graph)**.** *A connected graph G is distance-regular if for any vertices $x, y \in V_G$ and integers i, j, the number of vertices at distance i from x and distance j from y depends only on i, j and $d(x,y)$.*

In other words, there exists a constant $s^h_{i,j}$ for every h, i, j such that $s^h_{i,j} = |N_i(x) \cap N_j(y)|$ whenever $x, y \in V_G$ satisfy $d(x,y) = h$. Thus, we can apply Theorem 18 with $N_x = s^0_{x,x}$, $a_x = s^1_{x,x-1}$, and $b_x = s^x_{1,x-1}$.

The significance of distance-regular graph is not only about BW optimality. Many of distance-regular graphs have low diameters, so their schedules are not only BW-optimal but also close to, and in some cases exactly, Moore optimal. Table A.2 gives examples of distance-regular graphs at $d = 4$. In addition, many of the base graphs mentioned in this paper are also distance-regular like complete bipartite graphs (Figure 3.1) and Hamming graphs. One can refer to [36] for a repository of distance-regular graphs.

A.6.4 Circulant Graph

Circulant graph is a well-studied topology in both graph theory and network design. Many popular network topologies like shifted ring, chordal ring, and loop network are either subcategories of or closely related to circulant graphs. The definition of circulant graph is as follows:

Definition 18. *The circulant graph $C(n, \{a_1, \ldots, a_k\})$ is a bidirectional graph with vertex set $\{0, 1, \ldots, n-1\}$ and each node i is adjacent to nodes $i \pm a_1, \ldots, i \pm a_k \pmod n$.*

Note that in this paper, we only consider connected circulant graphs, and $C(n, \{a_1, \ldots, a_k\})$ is connected if and only if $\gcd(n, a_1, \ldots, a_k) = 1$ [98, 90]. It is easy to see that $C(n, \{a_1, \ldots, a_k\})$ is an n-node $2k$-regular topology.

We have found that the BFB schedule generation seems to give BW-optimal schedules for all circulant graphs. In particular, we have the following conjecture:

Conjecture 1. *For any circulant graph $C(n,\{a_1,\ldots,a_k\})$, there exists a BW-optimal BFB schedule.*

While we leave a complete proof or disproof of this conjecture for future work, we have proved the conjecture holds when $k=2$, which corresponds to the graph having degree 4.

Circulant graph revolutionized our Pareto frontier of topologies since it can be constructed for every N and even value d. It can provide a BW-optimal topology if our expansion techniques fail to produce one at some N and d. Since all circulant graphs seem to be BW-optimal, the question is what choices of $a_1,\ldots,a_k$ result in minimum total-hop latency, or equivalently, minimum diameter for a given n and k. While this remains largely an open question in graph theory [98], the case of $k=2$ has been solved in [17]:

Theorem 22. *Given $n>6$ and $m=\lceil(-1+\sqrt{2n-1})/2\rceil$, circulant graph $C(n,\{m,m+1\})$ has a diameter equal to m, which is the minimum diameter over all circulant graphs $C(n,\{a_1,a_2\})$.*

We can certainly use multiedge to apply this construction for any even degree that is ≥ 4. The resulting topology has $\Theta(\sqrt{N})$ diameter, which is a significant improvement in terms of total-hop latency when BW optimality is required. Previously, the only topology that is known to be BW-optimal for any N and d is ring, which has $\Theta(N)$ diameter.

A.7 Proofs

Theorem 7. *Given a d-regular topology G, if (G,A_G) is an N-node allgather algorithm, then $(L(G),A_{L(G)})$ is a dN-node allgather algorithm satisfying:*

$$T_L(A_{L(G)}) = T_L(A_G) + \alpha, \tag{A.2}$$

$$T_B(A_{L(G)}) \leq T_B(A_G) + \frac{M}{B}\cdot\frac{1}{N}. \tag{A.3}$$

Proof. Let $v'v,uw$ be arbitrary two distinct vertices in $L(G)$. We want to show there exists a sequence in $A_{L(G)}$ going from $v'v$ to uw like in Definition 4 for any $x\in S$. If $u=v$, then $((v'v,S),(v'v,uw),1)$ at the first comm step suffices. If $u\neq v$, because A_G is allgather, there exists a sequence in A_G:

$$((v,C_1),(v,w_1),t_1),((v,C_2),(w_1,w_2),t_2),\ldots,((v,C_n),(w_{n-1},u),t_n),$$

where $t_1<t_2<\cdots<t_n$ and $x\in C_1\cap C_2\cap\cdots\cap C_n$. Thus, by Definition 1, there exists a sequence in $A_{L(G)}$:

$$((v'v,S),(v'v,vw_1),1),((v'v,C_1),(vw_1,w_1w_2),t_1+1),\ldots,((v'v,C_n),(w_{n-1}u,uw),t_n+1),$$

as desired. The new algorithm $(L(G), A_{L(G)})$ has dM total data length, because the number of nodes has grown d-fold while the size of a shard remains the same.

As for $T_L(A_{L(G)})$ and $T_B(A_{L(G)})$, equality (A.2) trivially follows the Definition 1. Let $[A_{L(G)}]_t$ and $[A_G]_t$ be the subschedules of $A_{L(G)}$ and A_G at comm step t. Given $v \in V_G$, because G is d-regular, we have $|\{v'v \mid v'v \in V_{L(G)}\}| = |\{(v', v) \mid (v', v) \in E_G\}| = d$. Given any edge (uw, ww') and t, there are at most d number of $((v'v, C), (uw, ww'), t+1) \in A_{L(G)}$ for each $((v, C), (u, w), t) \in A_G$ by Definition 1. Thus, given (uw, ww'),

$$\sum_{((v'v,C),(uw,ww'))\in[A_{L(G)}]_{t+1}} |C| \le \sum_{((v,C),(u,w))\in[A_G]_t} d \cdot |C|.$$

It follows that $T_B([A_{L(G)}]_{t+1}, dM, B) \le d \cdot T_B([A_G]_t, M, B)$ and hence $\sum_{t=2}^{t_{\max}+1} T_B([A_{L(G)}]_t, dM, B) \le d \cdot T_B(A_G, M, B)$. For the first comm step, we have

$$T_B([A_{L(G)}]_1, dM, B) = \frac{|S|}{B/d} = \frac{M/N}{B/d}.$$

Assuming $T_B(A_G, M, B) = \tau(M/B)$ for some constant τ, we have $d \cdot T_B(A_G, M, B) = T_B(A_G, dM, B)$. It follows that

$$T_B(A_{L(G)}, dM, B) = \sum_{t=1}^{t_{\max}+1} T_B([A_{L(G)}]_t, dM, B) \le \frac{M/N}{B/d} + d \cdot T_B(A_G, M, B) = T_B(A_G, dM, B) + \frac{dM}{B} \cdot \frac{1}{N}.$$

Replacing dM by M gives (A.3) as desired. □

Corollary 7.1. *Given a d-regular topology G, if (G, A_G) is an N-node allgather algorithm with $T_B(A_G, M, B) = \tau(M/B)$ for some constant τ, then $(L^n(G), A_{L^n(G)})$ is a d^nN-node allgather algorithm satisfying:*

$$T_L(A_{L^n(G)}) = T_L(A_G) + n\alpha, \tag{A.4}$$

$$T_B(A_{L^n(G)}) \le T_B(A_G) + \frac{M}{B} \cdot \frac{d}{d-1}\left(\frac{1}{N} - \frac{1}{d^nN}\right). \tag{A.5}$$

Theorem 9. *If (G, A_G) is BW-optimal with N nodes, then $T_B(A_{L^n(G)})/T_B^*(d^nN) \le 1 + [(d-1)N]^{-1}$ for all n.*

Proof. If (G, A_G) is BW-optimal, then $T_B(A_G) = \frac{M}{B} \cdot \frac{N-1}{N}$ and

$$T_B(A_{L^n(G)}) \le \frac{M}{B}\left[1 + \frac{1}{d-1}\left(\frac{1}{N} - \frac{d}{d^nN}\right)\right]. \tag{A.13}$$

It is trivial to see that (A.13)$/T_B^*(d^nN) \nearrow 1 + [(d-1)N]^{-1}$ as $n \to \infty$. □

Theorem 1. *If A is a reduce-scatter/allgather schedule for G, then A^T is an allgather/reduce-scatter schedule for G^T.*

Proof. Suppose (G,A) is a reduce-scatter algorithm. For arbitrary $x \in S$ and distinct $u, v \in V_G$, there exists a sequence of tuples in A:

$$((v,C_1),(u,w_1),t_1),((v,C_2),(w_1,w_2),t_2),\ldots,((v,C_n),(w_{n-1},v),t_n),$$

where $t_1 < t_2 < \cdots < t_n$ and $x \in C_1 \cap C_2 \cap \cdots \cap C_n$. It follows that there exists a sequence of tuples in A^T:

$$((v,C_n),(v,w_{n-1}),t'_n),\ldots,((v,C_2),(w_2,w_1),t'_2),((v,C_1),(w_1,u),t'_1).$$

where $t'_i = t_{\max} - t_i + 1$, so $t'_n < \cdots < t'_2 < t'_1$. Since u, v, x are abitrary, A^T is an allgather schedule on G^T. One can similarly show that if (G,A) is an allgather algorithm, then (G^T, A^T) is a reduce-scatter algorithm. □

Corollary 1.1. *Suppose $G \mapsto f(G)$ is a function to construct reduce-scatter/allgather schedule given graph G, then $G \mapsto f(G^T)^T$ is a function to construct allgather/reduce-scatter schedule given graph G.*

Corollary 1.2. *Suppose $(G,A) \mapsto (f(G), f(A))$ is a mapping within reduce-scatter/allgather algorithms, then $(G,A) \mapsto (f(G^T)^T, f(A^T)^T)$ is a mapping within allgather/reduce-scatter algorithms.*

Theorem 2. *Suppose G is reverse-symmetric. Let G^T be the transpose graph, and let $f : V_{G^T} \to V_G$ be the isomorphism from G^T to G. If (G,A) is a reduce-scatter/allgather algorithm, then $(G, f(A^T))$ is an allgather/reduce-scatter algorithm with $T_L(f(A^T)) = T_L(A)$ and $T_B(f(A^T)) = T_B(A)$.*

Proof. By definition of $f(A^T)$,

$$\begin{aligned}((f(v),C),(f(w),f(u)),t_{\max} - t + 1) \in f(A^T) &\Leftrightarrow ((v,C),(w,u),t_{\max} - t + 1) \in A^T \\ &\Leftrightarrow ((v,C),(u,w),t) \in A.\end{aligned}$$

Note that $(u,w) \in E_G \Leftrightarrow (w,u) \in E_{G^T} \Leftrightarrow (f(w),f(u)) \in E_G$, so $f(A^T)$ is a valid schedule for G.

Suppose (G,A) is a reduce-scatter algorithm. For any $x \in S$ and distinct $u, v \in V_G$, there exists a sequence of tuples in A:

$$((v,C_1),(u,w_1),t_1),((v,C_2),(w_1,w_2),t_2),\ldots,((v,C_n),(w_{n-1},v),t_n),$$

where $t_1 < t_2 < \cdots < t_n$ and $x \in C_1 \cap C_2 \cap \cdots \cap C_n$. It follows that there exists a sequence of tuples in $f(A^T)$:

$$((f(v),C_n),(f(v),f(w_{n-1})),t'_n),((f(v),C_{n-1}),(f(w_{n-1}),f(w_{n-2})),t'_{n-1}),\ldots,((f(v),C_1),(f(w_1),f(u)),t'_1),$$

where $t'_i = t_{\max} - t_i + 1$, and $x \in C_n \cap C_{n-1} \cap \cdots \cap C_1$. Because f is a bijection, $(G, f(A^T))$ is an allgather algorithm. $T_L(A) = T_L(f(A^T))$ and $T_B(A) = T_B(f(A^T))$ are trivial, and one can similarly prove that if (G,A) is an allgather algorithm, then $(G, f(A^T))$ is a reduce-scatter algorithm. □

Theorem 3. *Every reduce-scatter/allgather algorithm* (G,A) *satisfies* $T_L(A) \geq \alpha \cdot D(G)$*, where* $D(G)$ *is the diameter of* G*.*

Proof. The proof is mentioned in text. □

Corollary 3.1. *An* N*-node degree-*d *reduce-scatter/allgather algorithm* (G,A) *is total-hop latency optimal if and only if*

$$T_L(A) = \alpha \cdot D(G) = \alpha \cdot \min\{D(G') : |V_{G'}| = N, \deg(G') = d\}.$$

Theorem 4. $\frac{M}{B} \cdot \frac{N-1}{N}$ *is a lower bound of* $T_B(A)$ *for any* N*-node reduce-scatter/allgather algorithm* (G,A)*.*

Proof. The proof is mentioned in text. □

Corollary 4.1. *An* N*-node reduce-scatter/allgather algorithm* (G,A) *is BW-optimal if and only if* $T_B(A) = \frac{M}{B} \cdot \frac{N-1}{N}$*.*

Theorem 5. *An allgather algorithm* (G,A) *is BW-optimal if and only if:*

1. $\frac{1}{B/d}\sum_{((v,C),(u,w))\in A_t} |C| = T_B(A_t)$ *for all* $(u,v) \in E_G$ *and* $t \in \{1,\ldots,t_{\max}\}$*.* A_t *is the subschedule of* A *at comm step* t*.*
2. *Pick any distinct* $u,v \in V_G$*. For each* $x \in S$*, there exists a unique* $((v,C),(w,u),t) \in A$ *such that* $x \in C$*.*

Proof. If $T_B(A) = T_B^*(N) = \frac{M}{B} \cdot \frac{N-1}{N}$, then the amount of data received by each vertex must be equal to $M \cdot \frac{N-1}{N}$, and the ingress bandwidth B must be fully utilized. If condition 1 does not hold, then some link (w,u) is not fully utilized. If condition 2 does not hold, then the amount of data received by some node is greater than $M \cdot \frac{N-1}{N}$.

If both 1 and 2 hold, then every vertex receives exactly $M \cdot \frac{N-1}{N}$ in total and bandwidth are fully utilized. Thus, $T_B(A) = T_B^*(N)$ and (G,A) is BW-optimal. □

Theorem 6. *Given a reduce-scatter algorithm* (G,A)*, suppose* $T_B(A) = \frac{M}{B} \cdot y$*, then* $T_C(A) \leq M \cdot \gamma \cdot y$*.*

Proof. At any comm step t, suppose the BW runtime is $T_B(A_t) = \frac{M}{B} \cdot y_t$. It follows at comm step t, the amount of data each node receives is at most $B \cdot T_B(A_t) = M \cdot y_t$, so $T_C(A_t) \leq M \cdot \gamma \cdot y_t$. The theorem trivially follows $y = \sum_t y_t$. □

Corollary 6.1. *If allreduce algorithm* (G,A) *is BW-optimal, i.e.,* $T_B(A) = 2\frac{M}{B} \cdot \frac{N-1}{N}$*, then* $T_C(A) = M \cdot \gamma \cdot \frac{N-1}{N}$ *and* $T_B(A) + T_C(A) = 2M \cdot (\frac{1}{B} + \frac{\gamma}{2}) \cdot \frac{N-1}{N}$*.*

Theorem 8. *$(L^n(G), A_{L^n(G)})$ is Moore optimal if and only if (G, A_G) is Moore optimal.*

Proof. Suppose $T_L(A_G) = \alpha k$. Thus, (G, A_G) is Moore optimal if and only if

$$N > M_{d,k-1} = \sum_{i=0}^{k-1} d^i = \frac{d^k}{d-1} - \frac{1}{d-1}. \tag{A.14}$$

$(L^n(G), A_{L^n(G)})$ is Moore optimal if and only if

$$d^n N > M_{d,k+n-1} \Leftrightarrow N > \frac{d^k}{d-1} - \frac{1}{d^n(d-1)}. \tag{A.15}$$

Because (A.15) $-$ (A.14) < 1 and (A.14) is an integer, (A.14) and (A.15) are equivalent. □

Theorem 10. *Let A_G be a BFB allgather schedule for d-regular topology G with $|N^+(u)| > 1$ for all $u \in V_G$, then the expanded schedule $A_{L(G)}$ is a BFB allgather schedule for $L(G)$. In particular, if A_G is the optimal BFB schedule for G, then $A_{L(G)}$ is the optimal BFB schedule for $L(G)$ satisfying:*

$$T_B(A_{L(G)}) = T_B(A_G) + \frac{M}{B} \cdot \frac{1}{N}. \tag{A.6}$$

Proof. It is trivial to see that $A_{L(G)}$ is a BFB allgather schedule on $L(G)$. For the sake of contradiction, suppose there exists a BFB schedule $A'_{L(G)}$ that $T_B(A'_{L(G)}) < T_B(A_G) + \frac{M}{B} \cdot \frac{1}{N}$. Let $x^*_{v'v,(wu,uu'),t}$s be the solution of BFB LP (3.1) corresponding to $A'_{L(G)}$. We build a schedule A'_G by constructing a solution of (3.1) such that

$$x_{v,(w,u),t} = \frac{1}{d} \sum_{v' \in N^-(v)} x^*_{v'v,(wu,uu'),t+1},$$

where $u' \in N^+(u) \setminus \{v\}$ is arbitrary. To verify the construction is a valid solution, given any $u \in V_G$ and $v \in N_t^-(w)$, the equality of (3.1) follows:

$$\sum_w x_{v,(w,u),t} = \frac{1}{d} \sum_{v'} \sum_w x^*_{v'v,(wu,uu'),t+1} = \frac{1}{d} \sum_{v'} \sum_{wu} x^*_{v'v,(wu,uu'),t+1} = \frac{1}{d} \sum_{v'} 1 = \frac{1}{d} \cdot d = 1.$$

The third equality follows the equality constraint in (3.1). Now, given $(w,u) \in E_G$, observe that

$$\sum_v x_{v,(w,u),t} = \frac{1}{d} \sum_v \sum_{v'} x^*_{v'v,(wu,uu'),t+1} = \frac{1}{d} \sum_{v'v} x^*_{v'v,(wu,uu'),t+1} \le \frac{1}{d} U^*_{uu',t+1}.$$

Thus, $U_{u,t} = \max_w \sum_v x_{v,(w,u),t} \le \frac{1}{d} U^*_{uu',t+1}$ and hence

$$\max_{u \in V_G} U_{u,t} \le \frac{1}{d} \max_{uu' \in V_{L(G)}} U^*_{uu',t+1}.$$

Note that $U^*_{uu',1} = 1$ for all $uu' \in V_{L(G)}$, as each node must send the full shard to every neighbor at the 1st comm step in any BFB allgather schedule. By (3.2), we have

$$T_B(A'_G) \leq T_B(A'_{L(G)}) - \frac{M/(dN)}{B/d} = T_B(A'_{L(G)}) - \frac{M}{B} \cdot \frac{1}{N} < T_B(A_G),$$

contradicting A_G being the optimal BFB schedule. Thus, combined with inequality (A.3), we have proven $A_{L(G)}$ being optimal as well as the equality (A.6). □

Corollary 10.1. *Let A_G be a BFB allgather schedule for d-regular topology G with $|N^+(u)| > 1$ for all $u \in V_G$, then the expanded schedule $A_{L^n(G)}$ is a BFB allgather schedule for $L^n(G)$. In particular, if A_G is the optimal BFB schedule for G, then $A_{L^n(G)}$ is the optimal BFB schedule for $L^n(G)$ satisfying:*

$$T_B(A_{L^n(G)}) = T_B(A_G) + \frac{M}{B} \cdot \frac{d}{d-1} \left(\frac{1}{N} - \frac{1}{d^n N} \right).$$

Theorem 11. *Given a d-regular topology G without self loops, if (G, A_G) is an N-node allgather algorithm with $T_B(A_G, M, B) = \tau(M/B)$ for some constant τ, then $(G * n, A_{G*n})$ is an nN-node allgather algorithm satisfying:*

$$T_L(A_{G*n}) = T_L(A_G) + \alpha, \tag{A.7}$$

$$T_B(A_{G*n}) = T_B(A_G) + \frac{M}{B} \cdot \frac{n-1}{nN}. \tag{A.8}$$

Proof. Let u_i, v_j be arbitrary two distinct vertices in $G * n$. Suppose $u \neq v$ in G, then for any $x \in S$, there exists a sequence in A_G:

$$((v, C_1), (v, w^{(1)}), t_1), ((v, C_2), (w^{(1)}, w^{(2)}), t_2), \ldots, ((v, C_n), (w^{(n-1)}, u), t_n),$$

where $t_1 < t_2 < \cdots < t_n$ and $x \in C_1 \cap C_2 \cap \cdots \cap C_n$. By Definition 2, there exists a sequence in A_{G*n}:

$$((v_j, C_1), (v_j, w_j^{(1)}), t_1), ((v_j, C_2), (w_j^{(1)}, w_j^{(2)}), t_2), \ldots, ((v_j, C_n), (w_j^{(n-1)}, u_i), t_n),$$

as desired. Now, suppose $u = v$ in G. By previous proof, the shard of v_j reaches every in-neighbor u'_α of u_i by the end of comm step $t_{\max}$ since $u' \neq v$. Then, the last comm step $t_{\max} + 1$ added in step 2 of Definition 2 delivers the shard to u_i with each edge (u'_α, u_i) delivering $1/nd$ of a shard. Thus, A_{G*n} is a complete allgather.

In step 1 of Definition 2, we have $T_B([A_{G*n}]_t, nM, nB) = T_B([A_G]_t, M, B)$ and hence $\sum_{t=1}^{t_{\max}} T_B([A_{G*n}]_t, nM, nB) = T_B(A_G, M, B)$. The nM and nB are due to the fact that both the number of nodes and degree have grown n-fold. Thus,

$$T_B(A_{G*n}, M, B) = T_B(A_{G*n}, nM, nB)$$

$$
\begin{aligned}
&= T_B(A_G, M, B) + T_B([A_{G*n}]_{t_{\max}+1}, nM, nB) \\
&= T_B(A_G, M, B) + (n-1) \cdot \frac{(nM)/(nN)}{nd} \cdot \frac{1}{nB/(nd)} \\
&= T_B(A_G, M, B) + \frac{M}{B} \cdot \frac{n-1}{nN}.
\end{aligned}
$$

The first equality follows the assumption that $T_B(A_G, M, B) = \tau(M/B)$ for some constant τ. □

Corollary 11.1. *If (G, A_G) is BW-optimal and $T_B(A_G, M, B) = \tau(M/B)$ for some τ, then $(G*n, A_{G*n})$ is BW-optimal.*

Theorem 12. *Given a d-regular topology G, if (G, A_G) is an N-node allgather algorithm with $T_B(A_G, M, B) = \tau(M/B)$ for some constant τ, then $G^{\square n}$ is an nd-regular topology, and $(G^{\square n}, A_{G^{\square n}})$ is an N^n-node allgather algorithm satisfying:*

$$T_L(A_{G^{\square n}}) = n \cdot T_L(A_G), \tag{A.9}$$

$$T_B(A_{G^{\square n}}) = T_B(A_G) \cdot \frac{N}{N-1} \cdot \frac{N^n - 1}{N^n}. \tag{A.10}$$

Proof. We will show that $A^{(1)}$ is a valid allgather schedule. Since $A^{(i)}$s are simply starting at different dimensions, this also shows that $A^{(i)}$s and hence $A_{G^{\square n}}$ are all valid allgather schedules for $G^{\square n}$.

Let $\mathbf{u}$ be arbitrary vertex in $G^{\square n}$. For any $x \in S$, we will show that schedule $A^{(1)}$ broadcasts x from $\mathbf{u}$ to all vertices in $G^{\square n}$. At $j = 1$, $A^{(1)}$ performs an allgather over vertices $\{(v_1, \mathbf{u}[2:n]) \mid v_1 \in V_G\}$ which induce a subgraph of $G^{\square n}$ isomorphic to G. Thus, x has been broadcast to all vertices in $\{(v_1, \mathbf{u}[2:n]) \mid v_1 \in V_G\}$. At $j = 2$, $A^{(1)}$ performs an allgather over vertices $\{(v_1, v_2, \mathbf{u}[3:n]) \mid v_2 \in V_G\}$ for each v_1. By the end of $j = 2$, x has been broadcast to all vertices in $\{(v_1, v_2, \mathbf{u}[3:n]) \mid v_1, v_2 \in V_G\}$. By the end of $j = n$, x has been broadcast to all vertices in $\{\mathbf{v} \mid \mathbf{v} \in V_G^n\} = V_{G^{\square n}}$. Since $\mathbf{u}$ and x are arbitrary, $A^{(1)}$ is a valid allgather schedule for $G^{\square n}$.

As for performance, (A.9) is trivial. To prove (A.10), observe that at each j in $A^{(1)}$, allgather A_G is performed with a data size $N^{j-1}M/n$ over the subgraph induced by $\{(\mathbf{y}, v, \mathbf{z}) \mid v \in V_G\}$ for each $\mathbf{y} \in V_G^{j-1}, \mathbf{z} \in V_G^{n-j}$. The bandwidth of each node within the subgraph is $1/n$ of that in $G^{\square n}$. It follows that

$$
\begin{aligned}
T_B(A^{(1)}, N^{n-1}M/n, nB) &= \sum_{j=1}^{n} T_B(A_G, N^{j-1}M/n, B) \\
&= \sum_{j=1}^{n} \frac{N^{j-1}}{n} T_B(A_G, M, B) \\
&= \frac{N^n - 1}{n(N-1)} T_B(A_G, M, B).
\end{aligned}
$$

Therefore,

$$\begin{aligned} T_B(A_{G^{\square n}}, M, B) &= \frac{n}{N^{n-1}} T_B(A_{G^{\square n}}, N^{n-1}M, nB) \\ &= \frac{n}{N^{n-1}} T_B(A^{(1)}, N^{n-1}M/n, nB) \\ &= \frac{n}{N^{n-1}} \cdot \frac{N^n - 1}{n(N-1)} T_B(A_G, M, B) \\ &= T_B(A_G, M, B) \cdot \frac{N}{N-1} \cdot \frac{N^n - 1}{N^n}. \end{aligned}$$

□

Corollary 12.1. *If (G, A_G) is BW-optimal and $T_B(A_G, M, B) = \tau(M/B)$ for some τ, then $(G^{\square n}, A_{G^{\square n}})$ is BW-optimal.*

Theorem 13. *Let $G_1, G_2, \ldots, G_n$ be topologies that*

1. *$G_1, \ldots, G_n$ are nontrivial simple digraphs;*
2. *Every G_i has BW-optimal BFB allgather schedule.*

Then, the optimal BFB allgather schedule, i.e. the schedule generated by BFB LP (3.1), for $G_1 \square \ldots \square G_n$ is also BW-optimal. The total-hop latency of the schedule equals $\alpha \cdot D(G_1 \square \ldots \square G_n) = \alpha \cdot \sum_i D(G_i)$.

Proof. To prove the theorem, it is sufficient to show that if G_1 and G_2 have BW-optimal BFB schedules, then $G_1 \square G_2$ has a BW-optimal BFB schedule. By Theorem 16, let $x^*_{v_1,(w_1,u_1),t_1}$s and $x^*_{v_2,(w_2,u_2),t_2}$s be the solutions of (3.1) on G_1 and G_2 respectively. Let $\mathbf{u} = (u_1, u_2), \mathbf{v} = (v_1, v_2)$. Define $r \in [0,1]$, which we will decide later. We construct a solution of (3.1) for $G_1 \square G_2$ such that:

$$\begin{aligned} x_{\mathbf{v},((w_1,u_2),\mathbf{u}),t_1+t_2} &= \begin{cases} r \cdot x^*_{v_1,(w_1,u_1),t_1} & \text{if } u_2 \neq v_2, \\ x^*_{v_1,(w_1,u_1),t_1} & \text{if } u_2 = v_2, \end{cases} \\ x_{\mathbf{v},((u_1,w_2),\mathbf{u}),t_1+t_2} &= \begin{cases} (1-r) \cdot x^*_{v_2,(w_2,u_2),t_2} & \text{if } u_1 \neq v_1, \\ x^*_{v_2,(w_2,u_2),t_2} & \text{if } u_1 = v_1. \end{cases} \end{aligned} \tag{A.16}$$

First of all, because $d_{G_1 \square G_2}(\mathbf{v}, \mathbf{u}) = d_{G_1}(v_1, u_1) + d_{G_2}(v_2, u_2)$, it is easy to verify that (A.16) gives a BFB schedule. In addition, for any distinct $\mathbf{u}, \mathbf{v} \in G_1 \square G_2$ with $u_1 \neq v_1$ and $u_2 \neq v_2$,

$$\sum_{\mathbf{w}} x_{\mathbf{v},(\mathbf{w},\mathbf{u}),t_1+t_2} = r \sum_{w_1} x^*_{v_1,(w_1,u_1),t_1} + (1-r) \sum_{w_2} x^*_{v_2,(w_2,u_2),t_2}$$

$$= r + (1-r)$$
$$= 1$$

satisfying the equality in (3.1). The $u_1 = v_1$ or $u_2 = v_2$ case is trivial. Because G_1 and G_2 have BW-optimal BFB schedule, by Theorem 17, for any $(w_1, u_1) \in E_{G_1}$,

$$\sum_{v_1 \in N_t^{-G_1}(u_1)} x^*_{v_1,(w_1,u_1),t} = \frac{N_t^{-G_1}}{d_1}, \tag{A.17}$$

where $N_t^{-G_1}(u_1)$ is $N_t^-(u_1)$ in G_1. Define $N_{t_1,t_2}^{-G_1 \square G_2}(\mathbf{u}) = N_{t_1}^{-G_1}(u_1) \times N_{t_2}^{-G_2}(u_2)$, then it holds that

$$N_t^{-G_1 \square G_2}(\mathbf{u}) = \bigcup_{t_1=0}^{t} N_{t_1,t-t_1}^{-G_1 \square G_2}(\mathbf{u}) = N_t^{-G_1}(u_1) \times \{u_2\} \cup \{u_1\} \times N_t^{-G_2}(u_2) \cup \bigcup_{t_1=1}^{t-1} N_{t_1,t-t_1}^{-G_1 \square G_2}(\mathbf{u}).$$

Thus, (A.17) gives

$$\sum_{\mathbf{v} \in N_t^{-G_1 \square G_2}(\mathbf{u})} x_{\mathbf{v},((w_1,u_2),\mathbf{u}),t} = \frac{N_t^{-G_1}}{d_1} + r \sum_{t_1=1}^{t-1} \frac{N_{t_1}^{-G_1} N_{t-t_1}^{-G_2}}{d_1}. \tag{A.18}$$

for any $((w_1,u_2),\mathbf{u}) \in E_{G_1 \square G_2}$. For G_2, one can similarly get

$$\sum_{\mathbf{v} \in N_t^{-G_1 \square G_2}(\mathbf{u})} x_{\mathbf{v},((u_1,w_2),\mathbf{u}),t} = \frac{N_t^{-G_2}}{d_2} + (1-r) \sum_{t_2=1}^{t-1} \frac{N_{t-t_2}^{-G_1} N_{t_2}^{-G_2}}{d_2}. \tag{A.19}$$

The value of r is the solution to (A.18) = (A.19):

$$\frac{N_t^{-G_1}}{d_1} + r \sum_{t_1=1}^{t-1} \frac{N_{t_1}^{-G_1} N_{t-t_1}^{-G_2}}{d_1} = \frac{N_t^{-G_2}}{d_2} + (1-r) \sum_{t_2=1}^{t-1} \frac{N_{t-t_2}^{-G_1} N_{t_2}^{-G_2}}{d_2}.$$

To see there is always a solution $r \in [0,1]$, we have $N_t^{-G_1} \le d_1 \cdot N_{t-1}^{-G_1}$ and $N_t^{-G_2} \le d_2 \cdot N_{t-1}^{-G_2}$, so

$$\frac{N_t^{-G_1}}{d_1} - \frac{N_t^{-G_2}}{d_2} \le \frac{N_t^{-G_1}}{d_1} \le N_{t-1}^{-G_1} \le \sum_{t_2=1}^{t-1} \frac{N_{t-t_2}^{-G_1} N_{t_2}^{-G_2}}{d_2},$$
$$\frac{N_t^{-G_2}}{d_2} - \frac{N_t^{-G_1}}{d_1} \le \frac{N_t^{-G_2}}{d_2} \le N_{t-1}^{-G_2} \le \sum_{t_1=1}^{t-1} \frac{N_{t_1}^{-G_1} N_{t-t_1}^{-G_2}}{d_1}.$$

The last inequality follows that because G_2 is nontrivial simple digraph, $N_1^{-G_2} = d_2$ and hence $N_{t-1}^{-G_1} = N_{t-1}^{-G_1} N_1^{-G_2} / d_2$. Note that $a + rb = c + (1-r)d$ always has a solution $r \in [0,1]$ if $a - c \le d$ and $c - a \le b$. With (A.18) = (A.19), by Theorem 17, we have constructed a BW-optimal solution of (3.1) for $G_1 \square G_2$. The theorem trivially follows by induction. □

Theorem 14. *A schedule A for G is a BFB allgather schedule if and only if the following are satisfied:*

1. *If $((v,C),(w,u),t) \in A$, then $d(v,u) = d(v,w) + 1 = t$;*
2. *For any distinct $u, v \in V_G$, the collection of chunks $\mathcal{C}_v = \{C \mid ((v,C),(w,u),t) \in A\}$ satisfies $S = \bigcup_{C \in \mathcal{C}_v} C$.*

Proof. Let v_0, v_k be arbitrary two distinct vertices in V_G with $d(v_0, v_k) = k$. For any $x \in S$, we want to show that there exists a path taking x from v_0 to v_k. At comm step k, conditions 1 and 2 guarantee that there exists $v_{k-1} \in N^-(v_k)$ and $((v_0, C_k), (v_{k-1}, v_k), k) \in A$ such that $d(v_0, v_{k-1}) = k-1$ and $x \in C_k$. At comm step $k-1$, similarly, it is guaranteed that there exists $v_{k-2} \in N^-(v_{k-1})$ and $((v_0, C_{k-1}), (v_{k-2}, v_{k-1}), k-1) \in A$ such that $d(v_0, v_{k-2}) = k-2$ and $x \in C_{k-1}$. Thus, we have a sequence of tuples in A:

$$((v_0, C_1), (v_0, v_1), 1), ((v_0, C_2), (v_1, v_2), 2), \ldots, ((v_0, C_k), (v_{k-1}, v_k), k),$$

where $x \in C_1 \cap C_2 \cap \cdots \cap C_k$ as desired. In the other direction, if condition 1 fails, then A is not a BFB schedule; if condition 2 fails, then A is not a valid allgather schedule. □

Theorem 15. *If A is a BFB schedule for G, then the total-hop latency $T_L(A) = \alpha \cdot D(G)$.*

Proof. The proof is trivial. □

Theorem 16. *Given any topology G, linear program (3.1) gives the optimal BFB schedule of G.*

Proof. The proof is mentioned in text. □

Theorem 17. *Suppose (G,A) is a BFB allgather schedule. (G,A) is BW-optimal if and only if:*

1. *There exists a sequence $N_1^-, N_2^-, \ldots, N_{D(G)}^- \in \mathbb{N}$ such that for any $x \in \mathbb{N}$ and $u \in V_G$, $|N_x^-(u)| = N_x^-$.*
2. *For any $(w,u) \in E_G$, $\sum_{((v,C),(w,u)) \in A_t} |C| = \frac{M}{N} |N_t^-(u)|/d = \frac{M}{N} N_t^-/d$.*

Proof. At comm step t, each vertex needs to receive shards from vertices in $N_t^-(u)$. By condition 1 of Theorem 5, each in-edge of vertex u receives equal amount of data, so each in-edge receives $\frac{M}{N}|N_t^-(u)|/d$. In addition, condition 1 of Theorem 5 also forces every edge in G receiving equal amount of data at any given comm step, so BW optimality is achieved if and only if $\frac{M}{N}|N_t^-(u)|/d = \frac{M}{N}|N_t^-(v)|/d = \frac{M}{N} N_t^-/d$ for all $u, v \in V_G$. Note that condition 2 of Theorem 5 is automatically satisfied. Thus, the conditions of Theorem 17 lead to Theorem 5, and vice versa. □

Theorem 18. *There exists a BW-optimal BFB schedule for undirected graph G if for every distance x, two of the following constants exist:*

1. $N_x = |N_x(u)|$ *for any* $u \in V_G$;
2. $a_x = |N_x(u) \cap N_{x-1}(w)|$ *for any* $u \in V_G$ *and* $w \in N(u)$;
3. $b_x = |N(u) \cap N_{x-1}(v)|$ *for any* $u \in V_G$ *and* $v \in N_x(u)$.

Moreover, if two of N_x, a_x, b_x *exist, then the third one must also exist with* $N_x = da_x/b_x$.

Proof. It is easy to see $N_x = da_x/b_x$. Constant N_xs satisfy condition 1 of Theorem 17. As for 2 of Theorem 17, at comm step t, consider a BFB schedule such that for any $u, v, w \in V_G$ with $d(v,u) = d(v,w) + 1 = t$, node w sends $1/b_t$ of v's shard to u. Thus, $\sum_{((v,C),(w,u)) \in A_t} |C| = \frac{M}{N} a_t / b_t = \frac{M}{N} N_t / d$. □

Theorem 19. *The workloads can be balanced if and only if there exists no subset* $J \subseteq \{j_1, j_2, \ldots, j_m\}$ *such that*

$$\frac{|J|}{\left|\bigcup_{j \in J} f(j)\right|} > \frac{m}{d}.$$

Proof. Consider a flow network, where each j_a is connected to each $p_b \in f(j_a)$ with ∞ capacity. Source s is connected to each j_a with capacity 1, and each p_b is connected to sink t with capacity m/d. Thus, the workloads can be balanced if and only if the max flow is m. Given any subset J, consider the s-t cut $(A, \bar{A})$ that $A = s + J + f(J)$. The cut has capacity $m - |J| + \frac{m}{d}|f(J)|$, which is less than m if and only if the inequality is true. □

Theorem 20. *Rounding LP gives a solution with BW runtime* $T_B \leq T_B^{OPT} + \frac{M}{B} \cdot \frac{d(d^{D(G)}-1)}{(d-1)PN}$. *In addition, if topology G is Moore optimal, then* $T_B \leq T_B^{OPT} + \frac{M}{B} \cdot \frac{d}{P}$.

Proof. For any (w,u) at comm step t, since $|N_{t-1}^-(w)| \leq d^{t-1}$,

$$\sum_v y_{v,(w,u),t} < \sum_v 1 + y^{\text{LP}}_{v,(w,u),t} \leq d^{t-1} + \sum_v y^{\text{LP}}_{v,(w,u),t}.$$

Thus, we have $W_{u,t} \leq W^{\text{LP}}_{u,t} + d^{t-1}$. By (3.2),

$$T_B - T_B^{\text{OPT}} \leq T_B - T_B^{\text{LP}} \leq \frac{M/N}{B/d} \cdot \frac{1}{P} \sum_{t=1}^{D(G)} d^{t-1} = \frac{M}{B} \cdot \frac{d(d^{D(G)}-1)}{(d-1)PN}.$$

Note that we need to divide (3.2) by P, because $y_{v,(w,u),t} \in [0,P]$ in (A.11) while $x_{v,(w,u),t} \in [0,1]$ in (3.1). If G is Moore optimal (i.e., $N > M_{d,D(G)-1}$), it follows that

$$T_B - T_B^{\text{OPT}} < \frac{M}{B} \cdot \frac{d(d^{D(G)}-1)}{(d-1)PM_{d,D(G)-1}} = \frac{M}{B} \cdot \frac{d}{P}.$$

□

Theorem 21. *Suppose $D(\Pi_{d,m}) = k$, then $m > M_{d,k-2}$.*

Proof. From [54], we know that $k \leq \lceil \log_d m \rceil$. Then,

$$m \geq d^{k-1} > \frac{d^{k-1}-1}{d-1} = M_{d,k-2}.$$

□

Theorem 22. *Given $n > 6$ and $m = \lceil (-1+\sqrt{2n-1})/2 \rceil$, circulant graph $C(n,\{m,m+1\})$ has a diameter equal to m, which is the minimum diameter over all circulant graphs $C(n,\{a_1,a_2\})$.*

Proof. See [17]. □

A.8 Supplementary Tables and Figures

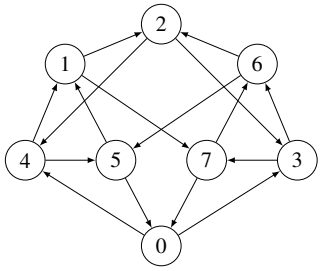


Figure A.8: Diamond Topology $(N = 8, d = 2)$**.**

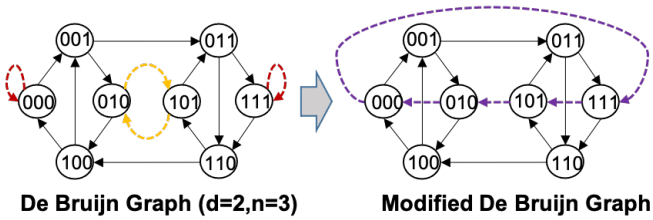


Figure A.9: An Example of Modified de Bruijn Graph $(N = 8, d = 2)$**.** The modification rewires the self loops and 2-cycles in de Bruijn graph to form a single long cycle without violating degree constraint.

Topology	**Allreduce**		**All-to-All**	
$N=32$, $d=4$	T_L	T_B	$D(G)$	MCF
$L(K_{4,4})$	3α	$1.000M/B$	3	5.71e−2
DistReg$(4,32)$	4α	$0.969M/B$	4	5.26e−2
Theoretical Bound	3α	$0.969M/B$	3	5.80e−2
$N=64$, $d=4$	T_L	T_B	$D(G)$	MCF
$\Pi_{4,64}$	3α	$1.312M/B$	3	2.17e−2
$L($DBJMod$(4,2))$	4α	$1.000M/B$	4	2.21e−2
Diamond$^{\square 2}$	6α	$0.984M/B$	6	1.87e−2
Theoretical Bound	3α	$0.984M/B$	3	2.42e−2
$N=128$, $d=4$	T_L	T_B	$D(G)$	MCF
$L^2(K_{4,4})$	4α	$1.031M/B$	4	9.89e−3
$L($DistReg$(4,32))$	5α	$1.000M/B$	5	9.26e−3
BiRing$(2,8)\square$UniRing$(1,4)^{\square 2}$	10α	$0.992M/B$	10	5.21e−3
Theoretical Bound	4α	$0.992M/B$	4	1.00e−2
$N=256$, $d=4$	T_L	T_B	$D(G)$	MCF
DBJ$(4,4)$	4α	$1.328M/B$	4	4.04e−3
$L^2($DBJMod$(4,2))$	5α	$1.016M/B$	5	4.10e−3
$L($Diamond$^{\square 2})$	7α	$1.000M/B$	7	3.62e−3
DBJMod$(2,4)^{\square 2}$	10α	$0.996M/B$	8	2.94e−3
Theoretical Bound	4α	$0.996M/B$	4	4.39e−3
$N=512$, $d=4$	T_L	T_B	$D(G)$	MCF
$L^3(K_{4,4})$	5α	$1.039M/B$	5	1.88e−3
$L^2($DistReg$(4,32))$	6α	$1.008M/B$	6	1.78e−3
$L($BiRing$(1,4)^{\square 3}\square$UniRing$(1,2))$	11α	$1.000M/B$	11	1.12e−3
UniRing$(1,4)^{\square 3}\square$UniRing$(1,8)$	16α	$0.998M/B$	16	5.58e−4
Theoretical Bound	5α	$0.998M/B$	5	1.90e−3
$N=1024$, $d=4$	T_L	T_B	$D(G)$	MCF
$\Pi_{4,1024}$	5α	$1.332M/B$	5	8.01e−4
$L^3(C(16,\{3,4\}))$	6α	$1.020M/B$	6	8.12e−4
$L^2($Diamond$^{\square 2})$	8α	$1.004M/B$	8	7.34e−4
$L($DBJMod$(2,4)^{\square 2})$	11α	$1.000M/B$	9	6.18e−4
$($UniRing$(1,4)\square$UniRing$(1,8))^{\square 2}$	20α	$0.999M/B$	20	2.79e−4
Theoretical Bound	5α	$0.999M/B$	5	8.57e−4

Table A.1: Pareto-efficient topologies at $N\in\{32,64,128,256,512,1024\}$, $d=4$. The results are generated from the topology finder (§3.5.4). For notations of the topologies, see Table 3.3 and A.3. For distance regular graphs (DistReg), see Table A.2. The MCF values are computed using LP (A.1).

Graph Name	N	T_L	T_L^*	$T_L - T_L^*$	T_L^{**}	$T_L - T_L^{**}$
Octahedron J(4,2)	6	2	2	0	2	0
Paley graph P9≅H(2,3)	9	2	2	0	2	0
K5,5-I	10	3	2	1	2	1
Distance-3 graph of Heawood graph	14	3	2	1	2	1
Line graph of Petersen graph	15	3	2	1	2	1
4-cube Q4≅H(4,2)	16	4	2	2	2	2
Line graph of Heawood graph	21	3	2	1	3	0
Incidence graph of PG(2,3)	26	3	3	0	3	0
Incidence graph of AG(2,4) minus a parallel class	32	4	3	1	3	1
Odd graph O4	35	3	3	0	3	0
Line graph of Tutte's 8-cage	45	4	3	1	3	1
Doubled Odd Graph D(O4)	70	7	3	4	4	3
Incidence graph of GQ(3,3)	80	4	3	1	4	0
Line graph of Tutte's 12-cage	189	6	4	2	5	1
Incidence graph of GH(3,3)	728	6	5	1	6	0

Table A.2: Examples of distance-regular graphs at $d = 4$ [36]. T_L^{**} is the bidirectional Moore optimality.

Topology	Notation	Degree	Size	Reverse-Symmetric	Bandwidth Optimal	Moore Optimal	BFB Schedule	Self-Loop	MultiEdge
Complete	K_m	$m-1$	m	✓	✓	✓	✓	×	×
Complete Bipartite (Fig 3.1)	$K_{d,d}$	d	$2d$	✓	✓	✓	✓	×	×
Hamming	$H(n,q) = K_q^{\square n}$	$n(q-1)$	q^n	✓	✓	$T_L = n$	✓	×	×
Kautz	$K(d,n) = L^n(K_{d+1})$	d	$d^n(d+1)$	✓	when $n=0$	✓	✓	×	×
Generalized Kautz (§A.6.2)	$\Pi_{d,m}$	d	$m \geq d+1$	×	when $m = d+1$	$T_L \leq T_L^* + 1$	✓	when m mod $(d+1) \neq 0$	×
Circulant (§A.6.4)	$C(m, \{a_1, \ldots, a_d\})$	d	m	✓	✓	×	✓	×	×
Directed Circulant		d	$d+2$	✓	✓	✓	✓	×	×
Bidirectional Ring	BiRing(d,m)	even d	$m \geq 3$	✓	✓	$T_L = \lfloor m/2 \rfloor$	✓	×	when $d > 2$
Unidirectional Ring	UniRing(d,m)	d	m	✓	✓	$T_L = m-1$	✓	×	when $d > 1$
Diamond (Fig A.8)		2	8	×	✓	✓	×	×	×
de Bruijn	DBJ(d,n)	d	d^n	✓	when $n \leq 1$	✓	✓	✓	×
Modified de Bruijn (Fig A.9)	DBJMod$(2,3)$	2	8	✓	✓	$T_L = 4$	×	×	×
	DBJMod$(2,4)$	2	16	×	✓	$T_L = 5$	×	×	×
	DBJMod$(3,2)$	3	9	×	✓	$T_L = 3$	×	×	×
	DBJMod$(4,2)$	4	16	×	✓	$T_L = 3$	×	×	×
Distance-Regular Graphs (§A.6.3)	DistReg(d,m)	d	m	✓	✓		✓	×	×

Table A.3: Summary of Important Topologies.

Appendix B

ForestColl: Throughput-Optimal Collective Communications on Heterogeneous Network Fabrics

In this appendix, we provide detailed mathematical analysis and proofs to supplement the main text. To summarize,

- §B.1 provides a summary of notations used in the main text and appendix.
- §B.2 discusses related works beyond schedule generation.
- §B.3 describes the implementation and parallelization of schedule generation algorithm.
- §B.4 shows a dilemma for step schedules to reach optimality.
- §B.5 elaborates on the mathematical details of the algorithm.
- §B.6 proves that every part of our algorithm runs in polynomial time.
- §B.7 describes a linear program to construct optimal allreduce schedule.
- §B.8 provides proofs of all theorems in this paper.

B.1 Notations

To ensure rigorous mathematical reasoning, we introduce the following notations:

- $G = (V = V_s \cup V_c, E)$: the input topology as a directed graph. V_s and V_c represent the switch nodes and compute nodes, respectively.
- $\vec{G}_x$: the auxiliary network constructed for optimality binary search. Defined in §B.5.1.
- $G(\{Ub_e\})$: a graph obtained by multiplying each link bandwidth b_e of G by U. Defined in §B.5.1.
- G^{ef}: a graph obtained by splitting off edge e and f of G. Defined in §B.5.2.

- $G^* = (V_c, E^*)$: the graph after removing all switch nodes from G using edge splitting technique. Defined in §B.5.2.
- $\widehat{D}_{(u,w),v}, \widehat{D}_{(w,t),v}$: the auxiliary networks for edge splitting (computing γ). Defined in §B.5.2.
- $\overline{D}$: the auxiliary network for spanning tree construction (computing μ). Defined in §B.5.3.
- M: total size of the data across all nodes.
- N: the number of compute nodes, i.e., $N = |V_c|$.
- b_e: the bandwidth of link e.
- k: the number of out-trees rooted at each compute node.
- x: the total bandwidth of the out-trees rooted at each compute node.
- y: the bandwidth utilized by each out-tree.
- U: equal to $1/y$. Used to scale edge capacities.
- γ: the maximum capacity we can safely replace (or split off) $(u,w),(w,t)$ with (u,t). Defined in Theorem 28.
- μ: the maximum capacity we can add an edge into a tree. Defined in (B.3) of §B.5.3.
- S, S^*: a cut represented as a vertex subset. S^* denotes the bottleneck cut, where $\frac{|S^* \cap V_c|}{B^+_G(S^*)} \geq \frac{|S \cap V_c|}{B^+_G(S)}$ for all $S \subset V, S \not\supseteq V_c$.
- $B^+_G(S), B^-_G(S)$: the exiting bandwidth and entering bandwidth of a vertex set S on a graph G, i.e., sum of the bandwidths of all links exiting/entering S.
- $F(x,y;G)$: the maxflow from node x to y in graph G.
- $c(A,B;G)$: the total capacity from vertex set A to B in graph G, i.e., the sum of the capacities of directed edges going from A to B.
- $\lambda(x,y;G)$: the edge connectivity from x to y in graph G. In integer-capacity graph, $\lambda(x,y;G) = F(x,y;G)$.
- $d^+(v), d^-(v)$: the in-degree and out-degree of node v. In integer-capacity graph, $d^+(v), d^-(v)$ are total ingress and egress capacity of v.
- $T_{u,i}$: the i-th out-tree rooted at node u.
- $R_{u,i}, \mathcal{V}(T_{u,i})$: the vertex set of the out-tree $T_{u,i}$.
- $\mathcal{E}(T_{u,i})$: the edge set of the out-tree $T_{u,i}$.

Method	Optimality	Generality	Scalability
SCCL [21]	Fixed-Chunk	Non-Switch	NP-hard
TACCL [128]	Heuristic	Yes	NP-hard
TE-CCL [81]	Heuristic	Yes	NP-hard
TACOS [156]	Greedy	Yes	Yes
BFB [160]	Conditional	Non-Switch	Yes
Blink [145]	Single-Root	Non-Switch	Yes
MultiTree [53]	Greedy	Yes	Yes
TTO [69]	Mesh-Only	Non-Switch	Yes
SyCCL [22]	Heuristic	Yes	NP-hard
ForestColl	Yes	Yes	Yes

Table B.1: Summary of related work. "Fixed-Chunk": SCCL's optimality is for a fixed number of data chunks with optimal chunking unknown. "Heuristic/Greedy": these methods rely on heuristic/greedy methods without optimality guarantees. "Conditional": BFB's optimality is conditioned on the underlying topology. "Single-Root": Blink focuses on single-root reduce/broadcast instead of reduce-scatter/allgather. "Mesh-Only": TTO supports only mesh topologies. "Non-Switch": these methods are limited to direct-connect topologies without switches. "NP-hard": these methods rely on NP-hard optimization formulations.

- $m(R_{u,i}), g(x,y)$: notations for spanning tree construction. Defined in Theorem 31.

B.2 Other Related Work

Other Optimizations of Collective Communications: Beyond schedule generation, other works optimize collective comms in ways orthogonal to ForestColl. TopoOpt [152], Rail-only [150], and BFB [160] optimize the underlying network topology, where ForestColl's optimality for any topology can be beneficial. C-Cube [29] explores efficient comm on logical trees, such as overlapping reduction and broadcast, but does not mention tree construction. BlueConnect [28] proposes a collective algorithm for single hierarchical switching fabrics but is otherwise inapplicable. Recent works [80, 148, 33, 68, 125, 27] explore prototype hardware/protocol designs for in-network multicast/aggregation under simplistic topologies. ForestColl is compatible with these designs, extending them to more complex topologies. Other works also optimize ML comm through network transport [147], packet scheduling [112, 162], and multi-tenancy [157, 118].

Hybrid Parallelism: A major line of work focuses on designing hybrid parallelism strategies to co-optimize the use of comp, comm, and memory resources. Megatron-LM [131, 99] showcases how to combine data, tensor, and pipeline parallelisms to speed up LLM training. FlexFlow [59] and Alpa [163] propose automated search for hybrid parallelism strategies, with Alpa specifically aiming to minimize comm cost. nnScaler [78] incorporates domain experts into the search process for more parallelization opportunities.

Comp-Comm Overlap: Another line of research aims to enhance the overlap between comp and comm in ML training. Some implementations of parallelisms, such as PyTorch DDP [75], FSDP [161], and Domino [146], exploit overlap opportunities within a single parallelism. More complicated approaches, including CoCoNet [57], Syndicate [88], and Centauri [26], optimize the scheduling of comp and comm operations to achieve overlap in hybrid parallelism. Recent works like CoCoNet [57], [149], T3 [111], and Flux [25] overlap at the finest scale by fusing comp and comm kernels.

ForestColl complements both hybrid parallelism and comp-comm overlap. In hybrid parallelism, comm cost plays a pivotal role in overall performance [59, 163, 78]. By speeding up comm, ForestColl alleviates comm bottlenecks and enables more optimizations for comp and memory access. Its adaptability to any topology—including subsets of a topology (§4.6.2)—enables more possibilities for hybrid parallelism. While comp-comm overlap aims to hide comm cost, the massive traffic required by LLMs means that comm cost remains substantial [146, 25, 149, 111]. Besides, resource contention on GPUs (e.g., comp units, memory bandwidth) between comp and comm operations underscores that overlap is not cost-free [149, 1]. ForestColl helps reduce the non-overlapped comm cost and enhance comm efficiency in overlap (§4.6.4).

B.3 Implementation of Schedule Generation

In this section, we describe our implementation of ForestColl's schedule generation algorithm, focusing on how we parallelize key components to leverage multicore processors. The algorithm is implemented in Java with $\sim$1100 LOC. For maxflow, we use the push–relabel algorithm [46] provided by JGraphT library [94]. Table B.2 shows the breakdown of schedule generation time for 1024-GPU topologies in §4.6.5.

Topology	Optimality Binary Search	Switch Node Removal	Spanning Tree Construction	Total Time
1024-GPU A100	2.2s	979s	1209s	36.5min
1024-GPU MI250	3.8s	550s	1708s	37.7min

Table B.2: Breakdown of schedule generation time for 1024-GPU topologies in §4.6.5. Runtimes were measured on a 128-core 2.2GHz CPU.

Optimality binary search is the fastest stage of the schedule generation algorithm. In Algorithm 1, we parallelize the computation of maxflows from node s to each compute node c. With this parallelization, ForestColl can derive the optimal throughput of 1024-GPU topologies within seconds. For switch node removal, we similarly parallelize the computation of γ in Algorithm 2, which also requires independent

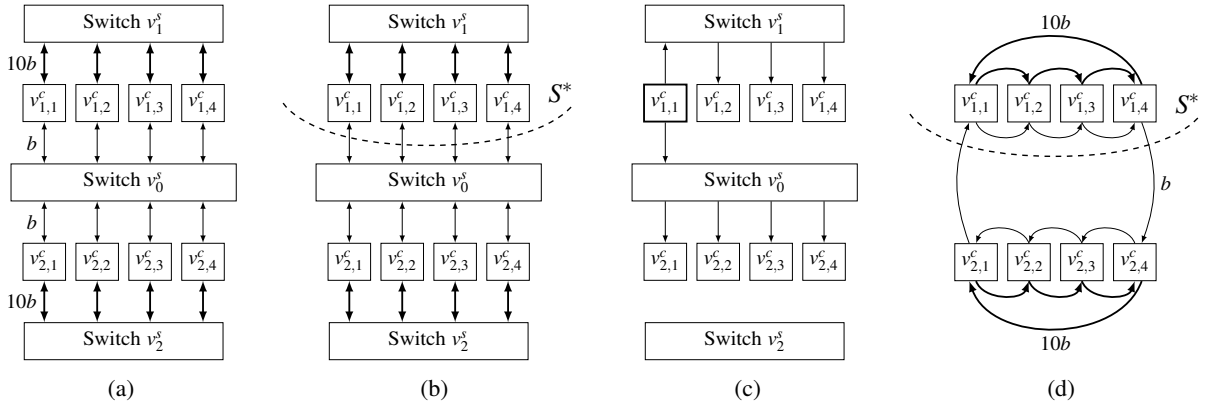


Figure B.1: An 8-compute-node switch topology in 2-box setting. The thick links have 10x the bandwidth of the thin ones. Figure (a) shows the original switch topology. Figure (b) shows the bottleneck cut in this topology. Figure (c) shows a spanning tree rooted at $v_{1,1}^c$ with switch-node broadcast. Figure (d) shows a suboptimal way of transforming the switch topology into a direct-connect logical topology (resulting in 4x worse optimal performance).

maxflow computations for each compute node. The runtime of this stage depends on the amount of switches in the network; for example, switch node removal on the A100 topology takes longer than on the MI250 due to the additional NVSwitches.

Parallelizing spanning tree construction is more challenging. The main bottleneck lies in computing μ in Algorithm 4. This step is difficult to parallelize because it requires only a single maxflow computation, and the push–relabel algorithm in JGraphT is not parallelized. The challenge is compounded by the sequential nature of edge additions: for each edge e_i, we must compute μ, decide whether to add or skip e_i, and then move on to e_{i+1}. As a result, constructing kN spanning trees requires computing μ for $\Omega(kN^2)$ times. To address this, we take inspiration from branch prediction. Specifically, thread j speculatively computes μ for edge e_{i+j} under the assumption that edges $e_i, \ldots, e_{i+j-1}$ are all added. If all consecutive edges can indeed be added, they are incorporated in the wall-clock time of a single μ computation. To further accelerate the process, we also assign threads to test subsequent edges under the assumption that e_i is rejected, ensuring that the next eligible edge is immediately available.

B.4 Minimality-or-Saturation Dilemma

In this section, we discuss why we need a tree-flow schedule instead of an ordinary step schedule to achieve throughput optimality. We show that in certain situations, a tree-flow schedule is *the only possible way* to achieve optimality. As shown in optimality ($\star$), the performance of a topology is bounded by a bottleneck cut

$(S^*, \overline{S^*})$. Suppose we want to achieve the performance bound given by the bottleneck cut, i.e., $(M/N)|S^* \cap V_c|/B_G^+(S^*)$, then the schedule must satisfy two requirements: (a) the bandwidth of the bottleneck cut, i.e., $B_G^+(S^*)$, must be saturated at all times, and (b) only the minimum amount of data required, i.e., $(M/N)|S^* \cap V_c|$, is transmitted through the bottleneck cut.

Consider the switch topology in Figure B.1a. The topology has 8 compute nodes and 3 switch nodes. The eight compute nodes are in two boxes. Each box has a switch v_1^s or v_2^s providing $10b$ egress/ingress bandwidth for each compute node in the box. The 8 compute nodes are also connected to a global switch v_0^s, providing b egress/ingress bandwidth for each compute node. It is easy to check that the bottleneck cut in this topology is a box cut $S^* = \{v_1^s, v_{1,1}^c, v_{1,2}^c, v_{1,3}^c, v_{1,4}^c\}$ shown in Figure B.1b. The cut provides a communication time lower bound of $(M/N)(4/4b)$. In comparison, a single-compute-node cut provides a much lower communication time lower bound $(M/N)(1/11b)$.

Suppose we want to achieve the lower bound by bottleneck cut S^*. Let C be the last chunk sent through the cut to box 2, and suppose it is sent to $v_{2,1}^c$. The first thing to try is to saturate the bandwidth. It means that the schedule terminates right after C is sent, leaving no idle time for $B_G^+(S^*)$. Then, at least one of $v_{2,2}^c, v_{2,3}^c, v_{2,4}^c$ must get C directly from box 1 because they have no time to get it from $v_{2,1}^c$. This violates minimality, however, because chunk C got sent through the bottleneck cut at least twice.

Suppose we want to achieve minimality. Then, $v_{2,1}^c$ has to broadcast C to $v_{2,2}^c, v_{2,3}^c, v_{2,4}^c$ within the box. However, because C is the last chunk sent through the cut by assumption, the cut bandwidth $B_G^+(S^*)$ is idle during the broadcast. The saturation requirement is violated. Thus, we are in a minimality-or-saturation dilemma that we cannot achieve both at the same time. However, we can do infinitely close by making chunk C infinitesimally small. By doing so, we transmit minimum data required, and we also make the idle time of bottleneck cut close to 0. In step schedule, one always needs to specify C as a fixed fraction of the total data, so it is impossible to achieve optimality in such a case. In contrast, the size of one send/recv can be arbitrarily small in tree-flow schedule. Therefore, a tree-flow schedule is the only way to achieve throughput optimality.

B.5 Algorithm Design

Let $G = (V = V_s \cup V_c, E)$ be an arbitrary network topology. We will compute an allgather schedule that reaches the optimal communication time ($\star$). We make two trivial assumptions about the topology: (a) all link bandwidths are integers and (b) G is *Eulerian*, i.e., the total egress bandwidth equals the total ingress bandwidth for any node. For (a), when bandwidths are rational numbers, one can always scale them up to

become integers. For (b), we use $B_G^+(v)$ and $B_G^-(v)$ to denote the total egress and ingress bandwidth of node v respectively. Since G is Eulerian, we have $B_G^+(v) = B_G^-(v)$ for all $v \in V$ and, consequently, $B_G^+(S) = B_G^-(S)$ for any $S \subseteq V$.

In summary, the algorithm contains three parts:

- §B.5.1: Conduct a binary search to compute the optimal communication time ($\star$). The binary search uses a network flow based oracle to test if a certain value is $\geq$ or $<$ than the true value of optimality ($\star$).
- §B.5.2: Transform the switch topology into a direct-connect logical topology by using *edge splitting* to remove switch nodes. The transformation is done without compromising optimal performance. This part can be skipped if the input topology is already direct-connect.
- §B.5.3: Construct spanning trees in direct-connect topology to achieve optimal performance. These spanning trees can then be mapped back to the original topology by reversing edge splitting, which determines the routing of communications between compute nodes.

The algorithm design is centered on earlier graph theoretical results on constructing edge-disjoint out-trees in directed graph [10, 136, 37, 13, 126]. A key observation leading to this algorithm is that *given a set of out-trees, there are at most U out-trees congested on any edge of G, if and only if, the set of out-trees is edge-disjoint in a multigraph topology obtained by duplicating each of G's edges U times.*

Another core design of our algorithm relies on *edge splitting*, also a technique from graph theory [10, 41, 56]. It is used to transform the switch topology into a direct-connect topology so that one can construct compute-node-only spanning trees. Previous works such as TACCL [128] and TACOS [156] attempt to do this by "unwinding" switch topologies into predefined logical topologies, such as rings. However, their transformations often result in a loss of performance compared to the original switch topology. For example, the previous works may unwind all switches in Figure B.1a into rings, resulting in Figure B.1d. However, it makes the bottleneck cut S^* worse in that the egress bandwidth of S^* becomes b instead of $4b$, causing the optimality ($\star$) to be $(M/N)(4/b)$ (4x worse). In contrast, our *edge splitting* strategically removes switch nodes without sacrificing any overall performance. Our transformation generates a direct-connect topology in Figure B.2b, which has the same optimality as Figure B.1a.

In this paper, we make extensive use of network flow between different pairs of nodes. For any flow network D, we use $F(x,y;D)$ to denote the value of maxflow from x to y in D. For disjoint A,B, let $c(A,B;D)$ be the total capacity from A to B in D. By min-cut theorem, $F(x,y;D) \leq c(A,\bar{A};D)$ if $x \in A, y \in \bar{A}$, and there

exists an x-y cut $(A^*, \overline{A^*})$ that $F(x, y; D) = c(A^*, \overline{A^*}; D)$.

B.5.1 Optimality Binary Search

In this section, we will show a way to compute the optimality ($\star$). Let $\{b_e\}_{e \in E}$ be the link bandwidths of G. By assumption, $\{b_e\}_{e \in E}$ are in $\mathbb{Z}_+$ and represented as capacities of edges in G. For any $x \in \mathbb{Q}$, we define $\vec{G}_x$ to be the flow network that (a) a source node s is added and (b) an edge (s, u) is added with capacity x for every vertex $u \in V_c$. Now, we have the following theorem:

Theorem 23. $\min_{v \in V_c} F(s, v; \vec{G}_x) \geq |V_c| x$ *if and only if* $1/x \geq \max_{S \subset V, S \not\supseteq V_c} |S \cap V_c| / B_G^+(S)$.

The implication of Theorem 23 is that we can do a binary search to get $1/x^* = \max_{S \subset V, S \not\supseteq V_c} |S \cap V_c| / B_G^+(S)$. The following initial range is trivial

$$\frac{N-1}{\min_{v \in V_c} B_G^-(v)} \leq \max_{S \subset V, S \not\supseteq V_c} \frac{|S \cap V_c|}{B_G^+(S)} \leq N - 1.$$

The lower bound corresponds to a partition containing all nodes except the compute node with minimum ingress bandwidth. The upper bound is due to the fact that $|S \cap V_c| \leq N-1$ and $B_G^+(S) \geq 1$. Starting with the initial range, one can then continuously test if $\min_{v \in V_c} F(s, v; \vec{G}_x) \geq |V_c| x$ for some midpoint x to do a binary search. To find the exact $1/x^*$, let $S^* = \arg\max_{S \subset V, S \not\supseteq V_c} |S \cap V_c| / B_G^+(S)$, then $1/x^*$ equals a fractional number with $B_G^+(S^*)$ as its denominator. Observe that $|S^* \cap V_c| \leq N-1$ and $|S^* \cap V_c| / B_G^+(S^*) \geq (N-1) / \min_{v \in V_c} B_G^-(v)$, so $B_G^+(S^*) \leq \min_{v \in V_c} B_G^-(v)$. Therefore, the denominator of $1/x^*$ is bounded by $\min_{v \in V_c} B_G^-(v)$. Now, we use the following proposition: Given two unequal fractional numbers a/b and c/d with $a, b, c, d \in \mathbb{Z}_+$, if denominators $b, d \leq X$ for some $X \in \mathbb{Z}_+$, then $|a/b - c/d| \geq 1/X^2$. The proposition implies that if $1/x^* = a/b$ for some $b \leq \min_{v \in V_c} B_G^-(v)$, then any $c/d \neq 1/x^*$ with $d \leq \min_{v \in V_c} B_G^-(v)$ satisfies $|c/d - 1/x^*| \geq 1/\min_{v \in V_c} B_G^-(v)^2$. Thus, one can run binary search until the range is smaller than $1/\min_{v \in V_c} B_G^-(v)^2$. Then, $1/x^*$ can be computed exactly by finding the fractional number closest to the midpoint with a denominator not exceeding $\min_{v \in V_c} B_G^-(v)$. This can be done with the continued fraction algorithm or a simple brute-force search if $\min_{v \in V_c} B_G^-(v)$ is not astronomical.

At this point, we have already known the optimality of communication time given a topology G. For the remainder of this section, we will show that there exists a family of spanning trees that achieves this optimality. First of all, we have assumed that G's links have the set of bandwidths $\{b_e\}_{e \in E}$. For the simplicity of notation, we use $G(\{c_e\})$ to denote the same topology as G but with the set of bandwidths $\{c_e\}_{e \in E}$ instead. $\vec{G}_x(\{c_e\})$ is also defined accordingly. When $\{c_e\}_{e \in E}$ are integers, we say a family of out-trees $\mathcal{F}$ is *edge-disjoint* in

$G(\{c_e\})$ if the number of trees using any edge $e \in E$ is less than or equal to c_e, i.e., $\sum_{T\in\mathcal{F}} \mathbb{I}[e \in T] \le c_e$ for all $e \in E$. The intuition behind this edge-disjointness is that *the integer capacity c_e represents the number of multiedges from the tail to the head of e.*

Now, we find $U \in \mathbb{Q}, k \in \mathbb{N}$ such that $U/k = 1/x^*$ and $Ub_e \in \mathbb{Z}_+$ for all $e \in E$. For simplicity of schedule, we want k to be as small as possible. The following proposition shows how to find such U,k: Given $\{b_e\}_{e\in E} \subset \mathbb{Z}_+$ and $1/x^* \in \mathbb{Q}$, let p/q be the simplest fractional representation of $1/x^*$, i.e., $p/q = 1/x^*$ and $\gcd(p,q) = 1$. Suppose $k \in \mathbb{N}$ is the smallest such that there exists $U \in \mathbb{Q}$ satisfying $U/k = 1/x^*$ and $Ub_e \in \mathbb{Z}_+$ for all $e \in E$, then $U = p/\gcd(q, \{b_e\}_{e\in E})$ and $k = Ux^*$. In Figure B.1a's example, we have $1/x^* = |S^* \cap V_c|/B^+_G(S^*) = 4/4b = 1/b$ and thus $U = 1/b, k = 1$.

Consider the digraph $G(\{Ub_e\})$. Each edge of $G(\{Ub_e\})$ has integer capacity. We will show that there exists a family of edge-disjoint out-trees $\{T_{u,i}\}_{u\in V_c, i\in[k]}$ in $G(\{Ub_e\})$ with $T_{u,i}$ rooted at u and $\mathcal{V}(T_{u,i}) \supseteq V_c$. Here, $[k] = \{1,2,\ldots,k\}$ and $\mathcal{V}(T_{u,i})$ denotes the vertex set of $T_{u,i}$. We use the following theorem proven by Bang-Jensen et al. [10]:

Theorem 24 (Bang-Jensen et al. [10])**.** *Let $D = (V,E)$ be a digraph with a special node s called a root, and let $T' = \{v \mid v \in V - s, d^-(v) < d^+(v)\}$. Assume that $\lambda(s,v;D) \ge n (\ge 1)$ for all $v \in T'$. Then there is a family $\mathcal{F}$ of n edge-disjoint out-trees rooted at s so that every $v \in V$ belongs to at least $\min(n, \lambda(s,v;D))$ members of $\mathcal{F}$.*

Because we see integer capacity as the number of multiedges, here, the total in-degree $d^-(v)$ and out-degree $d^+(v)$ are simply the total ingress and egress capacity of v in $G(\{Ub_e\})$. $\lambda(x,y;D)$ denotes the edge-connectivity from x to y in D, i.e., $\lambda(x,y;D) = \min_{x\in A, y\in\bar{A}} c(A,\bar{A};D)$. By min-cut theorem, $\lambda(x,y;D)$ is also equal to the maxflow from x to y. Theorem 24 leads to the following:

Theorem 25. *Given an integer-capacity Eulerian digraph $D = (V_s \cup V_c, E)$ and $k \in \mathbb{N}$, there exists a family of edge-disjoint out-trees $\{T_{u,i}\}_{u\in V_c, i\in[k]}$ in D with $T_{u,i}$ rooted at u and $\mathcal{V}(T_{u,i}) \supseteq V_c$ if and only if $\min_{v\in V_c} F(s,v;\vec{D}_k) \ge |V_c|k$.*

Consider the flow network $\vec{G}_k(\{Ub_e\})$. It is trivial to see that each edge in $\vec{G}_k(\{Ub_e\})$ has exactly U times the capacity as in $\vec{G}_{x^*}$, including the edges incident from s. Thus, we have

$$\begin{aligned}\min_{v\in V_c} F(s,v;\vec{G}_k(\{Ub_e\})) &= U \cdot \min_{v\in V_c} F(s,v;\vec{G}_{x^*}) \\ &\ge U \cdot |V_c| x^*\end{aligned}$$

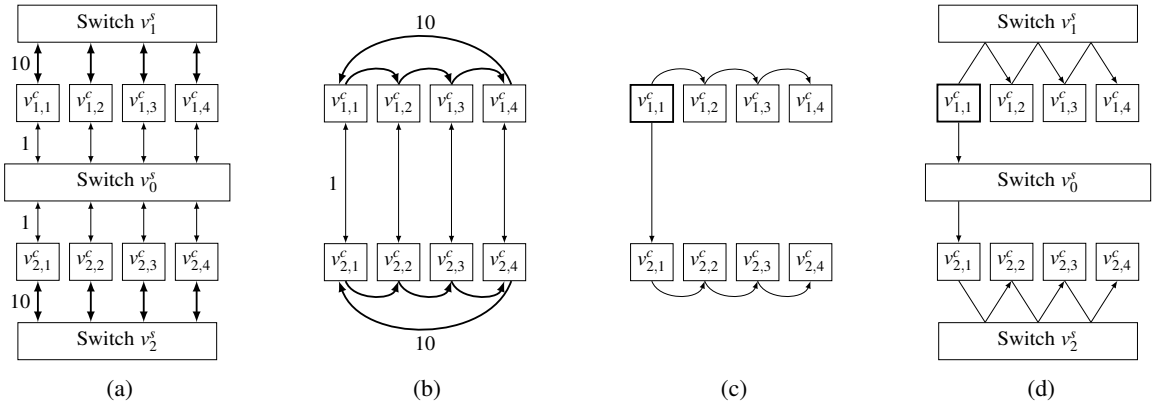


Figure B.2: Different stages of the topology in schedule construction. Figure (a) shows the topology of $G(\{Ub_e\})$. Note that the link capacities no longer have b as a multiplier. Figure (b) shows the topology G^* after edge splitting removes all switch nodes. Figure (c) shows a spanning tree constructed in G^*. Figure (d) shows the routings in G corresponding to the spanning tree.

$$= |V_c|k.$$

By Theorem 25, there exists a family of edge-disjoint out-trees $\{T_{u,i}\}_{u\in V_c, i\in[k]}$ in $G(\{Ub_e\})$ with $T_{u,i}$ rooted at u and $\mathcal{V}(T_{u,i}) \supseteq V_c$. Observe that for any edge $e \in E$, at most Ub_e number of trees from $\{T_{u,i}\}_{u\in V_c, i\in[k]}$ use edge e. For allgather, we make each tree broadcast $1/k$ of the root's data shard, then the communication time is

$$T_{\text{comm}} \leq \max_{e\in E} \frac{M}{Nk} \cdot \frac{Ub_e}{b_e} = \frac{M}{N} \cdot \frac{U}{k} = \frac{M}{N} \cdot \frac{1}{x^*} = \frac{M}{N} \max_{S\subset V, S\not\supseteq V_c} \frac{|S\cap V_c|}{B_G^+(S)}$$

reaching the optimality ($\star$) given topology G.

At this point, one may be tempted to construct and use $\{T_{u,i}\}_{u\in V_c, i\in[k]}$ to perform allgather. However, because $T_{u,i}$ can be an arbitrary tree in $G(\{Ub_e\})$, it may force switch nodes to broadcast like v_0^s, v_1^s in Figure B.1c. In the following section, we introduce a way to remove switch nodes from $G(\{Ub_e\})$, while preserving the existence of out-trees with the same communication time. Afterward, we construct out-trees in the compute-node-only topology and map the communications back to $G(\{Ub_e\})$. This yields a schedule that achieves the same optimal performance but avoids switch-node broadcast.

B.5.2 Edge Splitting

To remove the switch nodes from $G(\{Ub_e\})$, we apply a technique called *edge splitting*. Consider a vertex w and two incident edges $(u,w),(w,t)$. The operation of edge splitting is to replace $(u,w),(w,t)$ by a direct edge (u,t) while maintaining edge-connectivities in the graph. In our context, w is a switch node. We continuously

Algorithm 3: Remove Switch Nodes

Input: Integer-capacity Eulerian digraph $D = (V_s \cup V_c, E)$ and $k \in \mathbb{N}$.
Output: Direct-connect digraph $D^* = (V_c, E^*)$ and path recovery table `routing`.
begin
  Initialize table `routing`
  foreach switch node $w \in V_s$ **do**
    foreach egress edge $f = (w,t) \in E$ **do**
      foreach ingress edge $e = (u,w) \in E$ **do**
        Compute γ as in (B.1).
        if $\gamma > 0$ **then**
          Decrease f's and e's capacity by γ. Remove e if its capacity reaches 0.
          Increase capacity of (u,t) by γ. Add the edge if $(u,t) \notin E$.
          `routing`$[(u,t)][w] \leftarrow$ `routing`$[(u,t)][w] + \gamma$
          if f's capacity reaches 0 **then break**
      // *Edge f should have 0 capacity at this point.*
      Remove edge f from D.
    // *Node w should be isolated at this point.*
    Remove node w from D.
  return the latest D as D^* and table `routing`

split off one capacity of an incoming edge to w and one capacity of an outgoing edge from w until w is isolated and can be removed from the graph. Because the edge-connectivities are maintained, we are able to show that $\min_{v \in V_c} F(s,v;\vec{G}_k(\{Ub_e\})) \geq |V_c|k$ is maintained in the process. Thus, by Theorem 25, the existence of spanning trees with the same optimal performance is also preserved.

We start with the following theorem from Bang-Jensen et al. [10]. The theorem was originally proven by Frank [41] and Jackson [56].

Theorem 26 (Bang-Jensen et al. [10])**.** *Let $D = (V + w, E)$ be a directed Eulerian graph, that is, $d^-(x) = d^+(x)$ for every node x of D. Then, for every edge $f = (w,t)$ there is an edge $e = (u,w)$ such that $\lambda(x,y;D^{ef}) = \lambda(x,y;D)$ for every $x,y \in V$, where D^{ef} is the resulting graph obtained by splitting off e and f in D.*

In our case, we are only concerned with edge-connectivity from s. We allow $\lambda(x,y;D^{ef}) \neq \lambda(x,y;D)$ as long as $\min_{v \in V_c} F(s,v;\vec{D}_k^{ef}) = \min_{v \in V_c} \lambda(s,v;\vec{D}_k^{ef}) \geq |V_c|k$ holds after splitting. Theorem 26 is used to derive the following theorem:

Theorem 27. *Given an integer-capacity Eulerian digraph $D = (V_s \cup V_c, E)$ and $k \in \mathbb{N}$ with $\min_{v \in V_c} F(s,v;\vec{D}_k) \geq |V_c|k$, for every edge $f = (w,t)$ $(w \in V_s)$ there is an edge $e = (u,w)$ such that $\min_{v \in V_c} F(s,v;\vec{D}_k^{ef}) \geq |V_c|k$.*

Note that here, f and e each represent one of the multiedges (or one capacity) between w,t and u,w,

respectively. Observe that edge splitting does not affect a graph being Eulerian. Thus, in $G(\{Ub_e\})$, we can iteratively replace edges $e=(u,w), f=(w,t)$ by (u,t) for each switch node $w \in V_s$, while maintaining $\min_{v \in V_c} F(s,v;\vec{G}_k^{ef}(\{Ub_e\})) \geq |V_c|k$. The resulting graph will have all nodes in V_s isolated. By removing V_s, we get a graph $G^* = (V_c, E^*)$ having compute nodes only. Because of Theorem 25, there exists a family of edge-disjoint out-trees in G^* that achieves the same optimal performance.

While one can split off one capacity of $(u,w),(w,t)$ at a time, this becomes inefficient if the capacities of edges are large. Here, we introduce a way to split off $(u,w),(w,t)$ by maximum capacity at once. Given edges $(u,w),(w,t) \in E$, we construct a flow network $\widehat{D}_{(u,w),v}$ from $\vec{D}_k$ for each $v \in V_c$ that $\widehat{D}_{(u,w),v}$ connects $(u,s),(u,t),(v,w)$ with ∞ capacity. Similarly, we construct a flow network $\widehat{D}_{(w,t),v}$ that connects $(w,s),(u,t),(v,t)$ with ∞ capacity.

Theorem 28. *Given an integer-capacity Eulerian digraph $D=(V_s \cup V_c, E)$ and $k \in \mathbb{N}$ with $\min_{v \in V_c} F(s,v;\vec{D}_k) \geq |V_c|k$, the maximum capacity that $e=(u,w), f=(w,t)$ can be split off with the resulting graph D^{ef} satisfying $\min_{v \in V_c} F(s,v;\vec{D}_k^{ef}) \geq |V_c|k$ is*

$$\gamma = \min\left\{ c(u,w;D)\ ,\ c(w,t;D)\ ,\ \min_{v \in V_c} F(u,w;\widehat{D}_{(u,w),v}) - |V_c|k\ ,\ \min_{v \in V_c} F(w,t;\widehat{D}_{(w,t),v}) - |V_c|k \right\}. \tag{B.1}$$

Based on Theorem 28, we are able to develop Algorithm 3. Note that the runtime of Algorithm 3 does not depend on the capacities of the digraph. One should also note that we update a table `routing` while splitting. After edge splitting, we are ready to construct spanning trees that only use compute nodes for broadcast. `routing` is then used to convert the spanning trees back to paths in G that use switch nodes for send/receive between compute nodes.

Figure B.2 gives an example of edge splitting. In Figure B.2a, within each box $i \in \{1,2\}$, we split off 10 capacity of $(v_{i,j}^c, v_i^s), (v_i^s, v_{i,(j \bmod 4)+1}^c)$ for $j=1,2,3,4$ to form a ring topology. Across boxes, we split off 1 capacity of $(v_{i,j}^c, v_0^s), (v_0^s, v_{(i \bmod 2)+1,j}^c)$ for $j=1,2,3,4$. The resulting topology Figure B.2b has compute nodes only, and the optimal communication time is still $(M/N)(4/4b)$ if bandwidth multiplier b is added.

B.5.3 Spanning Tree Construction

At this point, we have a digraph $G^* = (V_c, E^*)$ with only compute nodes. In this section, we construct k out-trees from every node that span all nodes V_c in G^*. We start by showing the existence of spanning trees with the following theorem in Tarjan [136]. The theorem was originally proven by Edmonds [37].

Algorithm 4: Spanning Tree Construction

Input: Integer-capacity digraph $D^* = (V_c, E^*)$ and $k \in \mathbb{N}$.
Output: Spanning tree $(R_{u,i}, \mathcal{E}(R_{u,i}))$ for each $u \in V_c, i \in [n_u]$. Subgraph $(R_{u,i}, \mathcal{E}(R_{u,i}))$s satisfy $\forall u \in V_c : \sum_{i=1}^{n_u} m(R_{u,i}) = k$ and $\forall e \in E^* : \sum\{m(R_{u,i}) \mid e \in \mathcal{E}(R_{u,i})\} \le c(e; D^*)$.

```
begin
    Initialize R_{u,1} = {u}, E(R_{u,1}) = ∅, m(R_{u,1}) = k, n_u = 1 for all u ∈ V_c.
    Initialize g(e) = c(e; D*) for all e ∈ E*.
    while there exists R_{u,i} ≠ V_c do
        while R_{u,i} ≠ V_c do
            Pick an edge (x, y) in D* that x ∈ R_{u,i}, y ∉ R_{u,i}.
            Compute μ as in (B.4).
            if μ = 0 then continue
            if μ < m(R_{u,i}) then
                n_u ← n_u + 1
                Create a new copy R_{u,n_u} = R_{u,i}, E(R_{u,n_u}) = E(R_{u,i}), m(R_{u,n_u}) = m(R_{u,i}) − μ.
                m(R_{u,i}) ← μ
            E(R_{u,i}) ← E(R_{u,i}) + (x, y)
            R_{u,i} ← R_{u,i} + y
            g(x, y) ← g(x, y) − μ.
            Remove (x, y) if g(x, y) reaches 0.
```

Theorem 29 (Tarjan [136])**.** *For any integer-capacity digraph $D = (V, E)$ and any sets $R_i \subseteq V$, $i \in [k]$, there exist k edge-disjoint spanning out-trees T_i, $i \in [k]$, rooted respectively at R_i, if and only if for every $S \neq V$,*

$$c(S, \bar{S}; D) \ge |\{i \mid R_i \subseteq S\}|. \tag{B.2}$$

A spanning out-tree is *rooted at* R_i if for every $v \in V - R_i$, there is exactly one directed path from a vertex in R_i to v within the acyclic subgraph of out-tree. To see there exists a family of edge-disjoint spanning out-trees $\{T_{u,i}\}_{u \in V_c, i \in [k]}$ in G^*, observe that each $T_{u,i}$ is rooted at $R_{u,i} = \{u\}$, so $|\{(u,i) \mid R_{u,i} \subseteq S\}| = |S|k$ for any $S \subset V_c, S \neq V_c$. We show the following theorem:

Theorem 30. *Given an integer-capacity digraph $D = (V_c, E)$ and $k \in \mathbb{N}$, $c(S, \bar{S}; D) \ge |S|k$ for all $S \subset V_c, S \neq V_c$ if and only if* $\min_{v \in V_c} F(s, v; \vec{D}_k) \ge |V_c|k$.

Since we ensured $\min_{v \in V_c} F(s, v; \vec{G}_k^*) \ge |V_c|k$, condition (B.2) is satisfied. Spanning tree construction essentially involves iteratively expanding each $R_{u,i} = \mathcal{V}(T_{u,i})$ from $\{u\}$ to V_c by adding edges to $T_{u,i}$, while maintaining condition (B.2). Tarjan [136] has proposed such an algorithm. For each $T_{u,i}$, the algorithm continuously finds an edge (x, y) with $x \in R_{u,i}, y \notin R_{u,i}$ that adding this edge to $T_{u,i}$ does not violate (B.2). It is proven that such an edge is guaranteed to exist. However, the runtime of the algorithm quadratically depends on the total number of spanning trees, i.e., Nk in our case. This becomes problematic when k is large,

as k can get up to $\min_{v \in V_c} B_G^-(v) / \gcd(\{b_e\}_{e \in E})$. Fortunately, Bérczi & Frank [13] have proposed a strongly polynomial time algorithm based on Schrijver [126]. The runtime of the algorithm does not depend on k at all. In particular, the following theorem has been shown:

Theorem 31 (Bérczi & Frank [13])**.** *Let $D = (V, E)$ be a digraph, $g : E \to \mathbb{Z}_+$ a capacity function, $\mathcal{R} = \{R_1, \ldots, R_n\}$ a list of root-sets, $\mathcal{U} = \{U_1, \ldots, U_n\}$ a set of convex sets with $R_i \subseteq U_i$, and $m : \mathcal{R} \to \mathbb{Z}_+$ a demand function. There is a strongly polynomial time algorithm that finds (if there exist) $m(\mathcal{R})$ out-trees so that $m(R_i)$ of them are spanning U_i with root-set R_i and each edge $e \in E$ is contained in at most $g(e)$ out-trees.*

In our context, we start with $\mathcal{R} = \{R_u \mid u \in V_c\}$ and $R_u = \{u\}, U_u = V_c, m(R_u) = k$. We define $\mathcal{E}(R_i)$ to be the edge set of the $m(R_i)$ out-trees corresponding to R_i, so $\mathcal{E}(R_u) = \emptyset$ is initialized. Given $\mathcal{R} = \{R_1, \ldots, R_n\}$, we pick an $R_i \neq V_c$, say R_1. Then, we find an edge (x, y) such that $x \in R_1, y \notin R_1$ and (x, y) can be added to $\mu : 0 < \mu \leq \min\{g(x, y), m(R_1)\}$ copies of the $m(R_1)$ out-trees without violating (B.2). If $\mu = m(R_1)$, then we directly add (x, y) to $\mathcal{E}(R_1)$ and $R_1 = R_1 + y$. If $\mu < m(R_1)$, then we add a copy R_{n+1} of R_1 that $\mathcal{E}(R_{n+1}) = \mathcal{E}(R_1), m(R_{n+1}) = m(R_1) - \mu$. We revise $m(R_1)$ to μ, add (x, y) to $\mathcal{E}(R_1)$, and $R_1 = R_1 + y$. Finally, we update $g(x, y) = g(x, y) - \mu$. Now, given $\mathcal{R} = \{R_1, \ldots, R_{n+1}\}$, we can apply the step repeatedly until $R_i = V_c$ for all $R_i \in \mathcal{R}$. According to Bérczi & Frank [13], μ is defined as follows:

$$\mu = \min \left\{ g(x, y) \ , \ m(R_1) \ , \ \min\{c(S, \bar{S}; D) - p(S; D) : x \in S, y \in \bar{S}, R_1 \not\subseteq S\} \right\} \tag{B.3}$$

where $p(S; D) = \sum\{m(R_i) \mid R_i \subseteq S\}$. Neither Bérczi & Frank [13] nor Schrijver [126] explicitly mentioned how to compute μ in polynomial time. Therefore, we describe a method for doing so. We construct a flow network $\overline{D}$ such that (a) a node s_i is added for each R_i except $i = 1$, (b) connect x to each s_i with capacity $m(R_i)$, and (c) connect each s_i to every vertex in R_i with ∞ capacity. We then show the following result:

Theorem 32. *For any edge (x, y) in D with $x \in R_1, y \notin R_1$,*

$$\mu = \min \left\{ g(x, y) \ , \ m(R_1) \ , \ F(x, y; \overline{D}) - \textstyle\sum_{i \neq 1} m(R_i) \right\}. \tag{B.4}$$

Thus, μ can be calculated by computing a single maxflow from x to y in $\overline{D}$. The complete algorithm is described in Algorithm 4. The resulting $\mathcal{R}$ can be indexed as $\mathcal{R} = \bigcup_{u \in V_c} \{R_{u,1}, \ldots, R_{u,n_u}\}$, where $R_{u,i}$ corresponds to $m(R_{u,i})$ number of identical out-trees rooted at u and specified by edge set $\mathcal{E}(R_{u,i})$. We have $\sum_{i=1}^{n_u} m(R_{u,i}) = k$ for all u. Thus, $\mathcal{R}$ can be decomposed into $\{T_{u,i}\}_{u \in V_c, i \in [k]}$. However, since all spanning trees within $R_{u,i}$ are identical, the allgather schedule can simply be specified in terms of $\mathcal{E}(R_{u,i})$ and $m(R_{u,i})$.

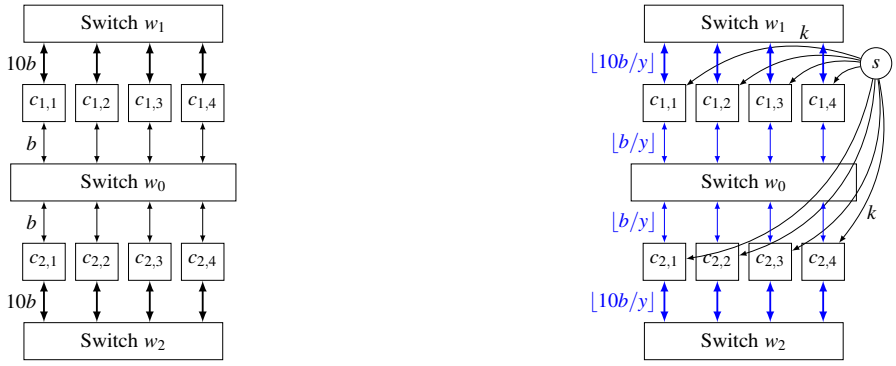


Figure B.3: The auxiliary network for fixed-k binary search. (a) shows the original topology. (b) shows the auxiliary network ForestColl uses to binary search for the optimal y (the bandwidth utilized by each tree) given a fixed k (the number of trees rooted at each compute node).

After construction, we have edge-disjoint spanning trees $\{T_{u,i}\}_{u\in V_c, i\in[k]}$ in G^*. Each of the edge (u,v) in $T_{u,i}$ may correspond to a path $u\to w_1\to\ldots\to w_n\to v$ in G with $w_1,\ldots,w_n$ being switch nodes. In other words, edges in $T_{u,i}$ only specify the source and destination of send/recv between compute nodes. Thus, one needs to use the `routing` in Algorithm 3 to recover the paths in G. For any edge (u,t) in G^*, `routing`$[(u,t)][w]$ denotes the amount of capacity from u to t that is going through $(u,w),(w,t)$. It should be noted that `routing` may be recursive, meaning that $(u,w),(w,t)$ may also go through some other switches. Because each capacity of (u,t) corresponds to one capacity of a path from u to t in G, the resulting schedule in G has the same performance in G^*, achieving the optimal performance ($\star$).

In Figure B.2's example, we construct a spanning tree like B.2c for each compute node. By reversing the edge splitting with `routing`, the spanning tree becomes the schedule in B.2d. Note that the corresponding schedule of a spanning tree in G^* is not necessarily a tree in G. For example, the schedule in B.2d visits switches v_1^s, v_2^s multiple times. By reversing the edge splitting for all spanning trees, we obtain a complete allgather schedule that achieves the optimal communication time of $(M/N)(4/4b)$.

One may be tempted to devise a way to construct spanning trees with low heights. This has numerous benefits such as lower latency at small data sizes and better convergence towards optimality. Although there is indeed potential progress to be made in this direction, constructing edge-disjoint spanning trees of minimum height has already been proven to be an NP-complete problem [14].

Algorithm 5: Fixed-k Binary Search

Input: A directed graph $G = (V_s \cup V_c, E)$ and k, the number of trees rooted at each compute node.
Output: y^*, the maximum bandwidth of each tree.
begin
  $l \leftarrow \frac{(N-1)k}{\min_{v \in V_c} B_G^-(v)}$ // *a lower bound of* $\frac{1}{y^*}$
  $r \leftarrow (N-1)k$ // *an upper bound of* $\frac{1}{y^*}$
  while $r - l \geq 1/\max_{e \in E} b_e^2$ **do**
    $\frac{1}{y} \leftarrow (l+r)/2$
    Add node s to G.
    foreach compute node $c \in V_c$ **do**
      Add an edge from s to c with capacity k.
    foreach link $e \in E$ with bandwidth b_e **do**
      Adjust the capacity of e to $\lfloor b_e/y \rfloor$.
    if the maxflow from s to each $c \in V_c$ is Nk **then**
      $r \leftarrow \frac{1}{y}$ // *case* $\frac{1}{y} \geq \frac{1}{y^*}$
    else
      $l \leftarrow \frac{1}{y}$ // *case* $\frac{1}{y} < \frac{1}{y^*}$
  Find the unique fractional number $\frac{p}{q} \in [l, r]$ such that denominator $q \leq \max_{e \in E} b_e$.
  return $\frac{q}{p}$ as y^*

B.5.4 Fixed-k Optimality

A potential problem of our schedule is that k, the number of spanning trees per root, depends linearly on link bandwidths, potentially reaching up to $\min_{v \in V_c} B_G^-(v)/\gcd(\{b_e\}_{e \in E})$. Although the runtime of spanning tree construction does not depend on k, in practice, one may want to reduce k to simplify the schedule. In this section, we offer a way to construct a schedule with the best possible performance for a fixed k. We start with the following theorem:

Theorem 33. *Given $U \in \mathbb{R}_+$ and $k \in \mathbb{N}$, a family of out-trees $\{T_{u,i}\}_{u \in V_c, i \in [k]}$ with $T_{u,i}$ rooted at u and $\mathcal{V}(T_{u,i}) \supseteq V_c$ achieves $\frac{M}{Nk} \cdot U$ communication time if and only if it is edge-disjoint in $G(\{\lfloor U b_e \rfloor\}_{e \in E})$.*

To test the existence of edge disjoint $\{T_{u,i}\}_{u \in V_c, i \in [k]}$ in $G(\{\lfloor U b_e \rfloor\}_{e \in E})$, by Theorem 25, we can simply test whether $\min_{v \in V_c} F(s, v; \vec{G}_k(\{\lfloor U b_e \rfloor\})) \geq |V_c| k$ holds. The following theorem provides a method for binary search to find the lowest communication time for the given k.

Theorem 34. *Let $\frac{M}{Nk} \cdot U^*$ be the lowest communication time that can be achieved with k out-trees per $v \in V_c$. Then, there exists a family of edge-disjoint out-trees $\{T_{u,i}\}_{u \in V_c, i \in [k]}$ in $G(\{\lfloor U b_e \rfloor\}_{e \in E})$ with $T_{u,i}$ rooted at u and $\mathcal{V}(T_{u,i}) \supseteq V_c$ if and only if $U \geq U^*$.*

The initial range is:

$$\frac{(N-1)k}{\min_{v\in V_c} B_G^-(v)} \le U^* \le (N-1)k.$$

Observe that there must exists $b_e \in E$ such that $U^* b_e \in \mathbb{Z}_+$; otherwise, U^* can be further decreased. Thus, the denominator of U^* must be less than or equal to $\max_{e\in E} b_e$. Similar to optimality binary search, by Proposition B.5.1, one can run binary search until the range is smaller than $1/\max_{e\in E} b_e^2$. Then, U^* can be determined exactly by computing the fractional number that is closest to the midpoint, while having a denominator less than or equal to $\max_{e\in E} b_e$. Algorithm 5 and Figure B.3 show the pseudocode and auxiliary network for binary search, respectively, with $y = 1/U$. After having U^*, one can simply apply edge splitting and spanning tree construction to $G(\{\lfloor U^* b_e \rfloor\}_{e\in E})$ to derive the pipeline schedule. *Note that $G(\{\lfloor U^* b_e \rfloor\}_{e\in E})$ is not necessarily Eulerian. If $G(\{\lfloor U^* b_e \rfloor\}_{e\in E})$ is not Eulerian, then edge splitting cannot be applied. However, in cases where G is bidirectional, $G(\{\lfloor U^* b_e \rfloor\}_{e\in E})$ is guaranteed to be Eulerian.*

The following theorem gives a bound on how close $\frac{M}{Nk}\cdot U^*$ is to optimality ($\star$):

Theorem 35. *Let $\frac{M}{Nk}\cdot U^*$ be the lowest communication time that can be achieved with k out-trees per $v \in V_c$. Then,*

$$\frac{M}{Nk}\cdot U^* \le \frac{M}{N} \max_{S\subset V, S\not\supseteq V_c} \frac{|S\cap V_c|}{B_G^+(S)} + \frac{M}{Nk}\cdot \frac{1}{\min_{e\in E} b_e}.$$

B.6 Time Complexity Analysis

In this section, we analyze the runtime complexity of different parts of the algorithm. To summarize, all parts run in polynomial time. One might be concerned by the time complexity like N^8. However, the runtime bounds are very loose—for example, using $O(N^2)$ to bound the number of edges, whereas realistic network topologies are typically much sparser. The bounds also assume single-core execution, while §B.3 describes various techniques to extensively parallelize the algorithm in practice. *The purpose of this analysis is to show that the algorithm runs in polynomial time, rather than to provide tight runtime bounds.* We leave proofs of tighter runtime bounds for future work. The empirical runtime performance of ForestColl is evaluated in §4.6.5.

Optimality Binary Search The key part of optimality binary search is to compute $\min_{v\in V_c} F(s, v; \vec{G}_x)$, which involves computing maxflow from s to every compute node in V_c. Assuming the use of preflow-push algorithm [46] to solve network flow, the time complexity to compute $\min_{v\in V_c} F(s, v; \vec{G}_x)$ is $O(N|V|^2|E|)$. Note that in practice, one can compute the maxflow from s to each $v \in V_c$ in parallel to significantly speed up the

computation. As for how many times $\min_{v\in V_c} F(s,v;\vec{G}_x)$ is computed, observe that the binary search terminates when range is smaller than $1/\min_{v\in V_c} B_G^-(v)^2$. The initial range of binary search is bounded by interval $(0,N)$, so the binary search takes at most $\lceil \log_2(N \min_{v\in V_c} B_G^-(v)^2)\rceil$ iterations. Because $\min_{v\in V_c} B_G^-(v) < |V| \max_{e\in E} b_e$, the total runtime complexity is $O(N|V|^2|E|(\log|V| + \log\max_{e\in E} b_e))$.

Edge Splitting In Algorithm 3, while we possibly add more edges to the topology, the number of edges is loosely bounded by $O(|V|^2)$. Thus, computing γ in Theorem 28 takes $O(N|V|^4)$, and γ is computed at most $O(|V_s||V|^4)$ times in the nested foreach loop. The total runtime can be loosely bounded by $O(N|V_s||V|^8)$.

Spanning Tree Construction Upon completion of Algorithm 3, G^* has N vertices and hence $O(N^2)$ number of edges. In Algorithm 4, μ only needs one maxflow to be computed. The runtime is thus $O(N^4)$. Bérczi & Frank [13] proved that μ only needs to be computed $O(mn^2)$ times, where m and n are the number of edges and vertices respectively. Thus, the runtime of Algorithm 4 can be loosely bounded by $O(N^8)$.

Fixed-k Optimality The runtime of this part is similar to optimality binary search, with the exception that the binary search takes at most $\lceil \log_2(Nk \max_{e\in E} b_e^2)\rceil$ iterations instead. The total runtime complexity is $O(N|V|^2|E|(\log N + \log k + \log\max_{e\in E} b_e))$.

B.7 Allreduce Linear Program

Generating an allreduce schedule is similar to generating an allgather schedule, as we can also use spanning tree packing. For allreduce, data flows through spanning in-trees to be reduced at root nodes and then is broadcast through spanning out-trees. One can apply the algorithm introduced in the main text to generate optimal out-trees and then reverse them for the in-trees. While this approach always yields the optimal allreduce schedule in our work so far, theoretically, allreduce can be further optimized from two perspectives:

1. Each node can be the root of a variable number of spanning trees instead of equal number in allgather.
2. Congestion between spanning in-trees and out-trees may be further optimized.

In this section, we introduce a linear program designed to optimize allreduce schedules, addressing both perspectives. This linear program formulation automatically determines: for (i), the number of trees rooted at each node, and for (ii), the bandwidth allocation of each edge for the reduce in-trees and broadcast out-trees, respectively.

The linear program works by formulating maxflow and edge splitting as linear program constraints. Given a graph G, suppose we want the maxflow from s to t in G being $\geq L$, i.e., $F(s,t;G) \geq L$. This constraint can

be expressed in terms of linear program constraints:

$$\begin{aligned}
\text{s.t.} \quad & \sum_{u\in N_G^-(v)} f^{s,t}_{(u,v)} \geq \sum_{w\in N_G^+(v)} f^{s,t}_{(v,w)}, && \forall v \in V(G), v \notin \{s,t\} \\
& \sum_{u\in N_G^-(t)} f^{s,t}_{(u,t)} \geq \sum_{w\in N_G^+(t)} f^{s,t}_{(t,w)} + L, \\
& 0 \leq f^{s,t}_{(u,v)} \leq c_{(u,v)}. && \forall (u,v) \in E(G)
\end{aligned}$$

As long as a solution exists for the set of decision variables $\{f^{s,t}_{(u,v)} \mid (u,v) \in E_G\}$, the maxflow from s to t is $\geq L$. Recall from §4.5.2 that our objective is to maximize x while ensuring that the maxflow from s to any $v \in V_c$ is $\geq Nx$. Here, because of (i), we assign a distinct x_v for each $v \in V_c$. Consequently, the optimization problem shifts to maximize $\sum_{v\in V_c} x_v$ without causing any $F(s,v;\vec{G}) < \sum_{v\in V_c} x_v$. The resulting allreduce communication time is

$$T_{\text{comm}} = M \Big/ \sum_{v\in V_c} x_v.$$

To optimize (ii), we introduce variables $c^{\text{RE}}_{(u,v)}$ and $c^{\text{BC}}_{(u,v)}$ to reserve the bandwidth of each link (u,v) for reduce in-trees and broadcast out-trees, respectively, with $c^{\text{RE}}_{(u,v)} + c^{\text{BC}}_{(u,v)} = b_{(u,v)}$. Thus, $c^{\text{RE}}_{(u,v)}$s and $c^{\text{BC}}_{(u,v)}$s induce two separate graph Gs, on which we can apply the spanning tree construction to derive in-trees and out-trees, respectively. The linear program formulation is as follows:

$$\begin{aligned}
\max \quad & \textstyle\sum_{v\in V_c} x_v \\
\text{s.t.} \quad & F(s,t;\vec{G}) \geq \textstyle\sum_{v\in V_c} x_v, && \forall t \in V_c \\
& \qquad \text{w.r.t. } 0 \leq f^{\text{s,t}}_{(s,v)} \leq x_v \text{ and } 0 \leq f^{\text{s,t}}_{(u,v)} \leq c^{\text{BC}}_{(u,v)} \\
& F(t,s;\vec{G}) \geq \textstyle\sum_{v\in V_c} x_v, && \forall t \in V_c \\
& \qquad \text{w.r.t. } 0 \leq f^{t,s}_{(v,s)} \leq x_v \text{ and } 0 \leq f^{t,s}_{(u,v)} \leq c^{\text{RE}}_{(u,v)} \\
& c^{\text{RE}}_{(u,v)} + c^{\text{BC}}_{(u,v)} \leq b_{(u,v)}, && \forall (u,v) \in E_G \\
& c^{\text{RE}}_{(u,v)}, c^{\text{BC}}_{(u,v)} \geq 0, && \forall (u,v) \in E_G \\
& x_v \geq 0. && \forall v \in V_c
\end{aligned} \tag{B.5}$$

Note that for reduce in-trees, we constrain the maxflow from V_c to s rather than from s to V_c. Accordingly, the x_vs are also capacities from $v \in V_c$ to s. The solution to LP (B.5) yields the optimal allreduce performance.

The linear program is sufficient for switch-free topology. In a switch topology G, similar to the algorithm in the main text, we need to convert it into a switch-free topology before applying the LP. We are unable to solve the LP and then apply edge splitting technique, as the $c^{\text{RE}}_{(u,v)}$s and $c^{\text{BC}}_{(u,v)}$s do not guarantee Eulerian

capacity in the respective induced graphs. To remove switch nodes, we add a level of indirection by defining $b'_{(\alpha,\beta)}$s for all $(\alpha,\beta) \in V_c^2$ to replace the $b_{(u,v)}$s in LP (B.5). We add multi-commodity flow constraints into the LP to ensure $b'_{(\alpha,\beta)}$ commodity flow from α to β in G under the capacities $b_{(u,v)}$s. Thus, the linear program can automatically allocate switch bandwidth for compute-to-compute flows.

In ideal mathematics, we can obtain rational solutions for all variables to derive the in-trees and out-trees to reach optimality. However, in practice, modern LP solvers cannot guarantee rational solutions. We can only round down $x_v, c^{\text{RE}}_{(u,v)}, c^{\text{BC}}_{(u,v)}$s to the nearest $1/k$ and construct spanning trees, assuming one wants at most $k \cdot \sum_{v \in V_c} x_v$ trees. This approach can approximate the optimal solution as k increases. Nevertheless, the optimal objective value of LP (B.5) provided by any solver always suggests the optimal allreduce performance, and we can use it to verify the optimality of allreduce schedule derived by using the algorithm in the main text.

B.8 Proofs

Theorem 23. $\min_{v \in V_c} F(s,v;\vec{G}_x) \geq |V_c|x$ *if and only if* $1/x \geq \max_{S \subset V, S \not\supseteq V_c} |S \cap V_c|/B^+_G(S)$.

Proof. $\Rightarrow$: Suppose $1/x < \max_{S \subset V, S \not\supseteq V_c} |S \cap V_c|/B^+_G(S)$. Let $S' \subset V, S' \not\supseteq V_c$ be the set that $1/x < |S' \cap V_c|/B^+_G(S')$. Pick arbitrary $v' \in V_c - S'$. Consider the maxflow $F(s,v';\vec{G}_x)$ and s-v' cut $(A,\bar{A})$ in $\vec{G}_x$ that $A = S' + s$. We have

$$\begin{aligned} c(A,\bar{A};\vec{G}_x) &= c(S',\bar{A};\vec{G}_x) + \sum_{u \in \bar{A} \cap V_c} c(s,u;\vec{G}_x) \\ &= B^+_G(S') + |V_c - S'|x \\ &< |S' \cap V_c|x + |V_c - S'|x \\ &= |V_c|x. \end{aligned} \tag{B.6}$$

By min-cut theorem, $\min_{v \in V_c} F(s,v;\vec{G}_x) \leq F(s,v';\vec{G}_x) \leq c(A,\bar{A};\vec{G}_x) < |V_c|x$.

$\Leftarrow$: Suppose $1/x \geq \max_{S \subset V, S \not\supseteq V_c} |S \cap V_c|/B^+_G(S)$. Pick arbitrary $v' \in V_c$. Let $(A,\bar{A})$ be an arbitrary s-v' cut and $S' = V \cap A = A - s$. It follows that $1/x \geq |S' \cap V_c|/B^+_G(S')$. Thus, following (B.6),

$$\begin{aligned} c(A,\bar{A};\vec{G}_x) &= B^+_G(S') + |V_c - S'|x \\ &\geq |S' \cap V_c|x + |V_c - S'|x \\ &= |V_c|x. \end{aligned}$$

Because cut $(A,\bar{A})$ is arbitrary, we have $F(s,v';\vec{G}_x) \geq |V_c|x$. Because v' is also arbitrary, $\min_{v \in V_c} F(s,v;\vec{G}_x) \geq |V_c|x$. □

Given two unequal fractional numbers a/b and c/d with $a,b,c,d \in \mathbb{Z}_+$, if denominators $b,d \leq X$ for some $X \in \mathbb{Z}_+$, then $|a/b - c/d| \geq 1/X^2$.

Proof. Because $a/b \neq c/d$, we have $ad - bc \neq 0$. Thus,

$$\left|\frac{a}{b} - \frac{c}{d}\right| = \left|\frac{ad-bc}{bd}\right| \geq \frac{1}{bd} \geq \frac{1}{X^2}.$$

□

Given $\{b_e\}_{e\in E} \subset \mathbb{Z}_+$ and $1/x^* \in \mathbb{Q}$, let p/q be the simplest fractional representation of $1/x^*$, i.e., $p/q = 1/x^*$ and $\gcd(p,q) = 1$. Suppose $k \in \mathbb{N}$ is the smallest such that there exists $U \in \mathbb{Q}$ satisfying $U/k = 1/x^*$ and $Ub_e \in \mathbb{Z}_+$ for all $e \in E$, then $U = p/\gcd(q, \{b_e\}_{e\in E})$ and $k = Ux^*$.

Proof. Since $U/k = 1/x^*$, we have $k = Ux^*$, so finding the smallest k is to find the smallest U such that (a) $Ux^* = Uq/p \in \mathbb{N}$ and (b) $Ub_e \in \mathbb{N}$ for all $e \in E$. Suppose $U = \alpha/\beta$ and $\gcd(\alpha,\beta) = 1$. Because α,β are coprime, $Ub_e \in \mathbb{N}$ implies $\beta | b_e$ for all $e \in E$. Again, because p,q are coprime, $Uq/p \in \mathbb{N}$ implies $p|\alpha$ and $\beta|q$. Thus, the smallest such α is p, and the largest such β is $\gcd(q, \{b_e\}_{e\in E})$. The proposition immediately follows. □

Theorem 24 (Bang-Jensen et al. [10])**.** *Let $D = (V,E)$ be a digraph with a special node s called a root, and let $T' = \{v \mid v \in V - s, d^-(v) < d^+(v)\}$. Assume that $\lambda(s,v;D) \geq n (\geq 1)$ for all $v \in T'$. Then there is a family $\mathcal{F}$ of n edge-disjoint out-trees rooted at s so that every $v \in V$ belongs to at least $\min(n, \lambda(s,v;D))$ members of $\mathcal{F}$.*

Theorem 25. *Given an integer-capacity Eulerian digraph $D = (V_s \cup V_c, E)$ and $k \in \mathbb{N}$, there exists a family of edge-disjoint out-trees $\{T_{u,i}\}_{u\in V_c, i\in[k]}$ in D with $T_{u,i}$ rooted at u and $\mathcal{V}(T_{u,i}) \supseteq V_c$ if and only if $\min_{v\in V_c} F(s,v;\vec{D}_k) \geq |V_c|k$.*

Proof. $\Rightarrow$: Pick arbitrary $v \in V_c$. Given the family of edge-disjoint out-trees $\{T_{u,i}\}_{u\in V_c, i\in[k]}$, we push one unit of flow from s to v along the path from u to v within tree $T_{u,i}$ for each $u \in V_c, i \in [k]$. Thus, we have constructed a flow assignment with $|V_c|k$ amount of flow. Since $v \in V_c$ is arbitrary, we have $\min_{v\in V_c} F(s,v;\vec{D}_k) \geq |V_c|k$.

$\Leftarrow$: Suppose $\min_{v\in V_c} F(s,v;\vec{D}_k) \geq |V_c|k$. Because D is Eulerian, we have $T' = \emptyset$ and the assumption in Theorem 24 trivially satisfied for $n = |V_c|k$. Therefore, a family $\mathcal{F}$ of $|V_c|k$ edge-disjoint out-trees rooted at s exists that each $v \in V_c$ belongs to all of them. In addition, for each $v \in V_c$, since $c(s,v;\vec{D}_k) = k$ and

$d^+(s) = |V_c|k = |\mathcal{F}|$, there are exactly k out-trees in $\mathcal{F}$ in which v is the only child of root s. By removing the root s from every out-tree in $\mathcal{F}$, we have the family of edge-disjoint out-trees $\{T_{u,i}\}_{u\in V_c, i\in[k]}$ in D as desired. □

Theorem 26 (Bang-Jensen et al. [10])**.** *Let $D=(V+w,E)$ be a directed Eulerian graph, that is, $d^-(x)=d^+(x)$ for every node x of D. Then, for every edge $f=(w,t)$ there is an edge $e=(u,w)$ such that $\lambda(x,y;D^{ef}) = \lambda(x,y;D)$ for every $x,y\in V$, where D^{ef} is the resulting graph obtained by splitting off e and f in D.*

Theorem 27. *Given an integer-capacity Eulerian digraph $D=(V_s\cup V_c,E)$ and $k\in\mathbb{N}$ with $\min_{v\in V_c} F(s,v;\vec{D}_k) \geq |V_c|k$, for every edge $f=(w,t)$ $(w\in V_s)$ there is an edge $e=(u,w)$ such that $\min_{v\in V_c} F(s,v;\vec{D}_k^{ef}) \geq |V_c|k$.*

Proof. Consider the flow network $\vec{D}_k$. We construct $\vec{D}'_k$ by adding a k-capacity edge from each $v\in V_c$ back to s. It is trivial to see that $\vec{D}'_k$ is Eulerian. By Theorem 26, given $f=(w,t)$, there exists an edge $e=(u,w)$ such that $\lambda(s,v;\vec{D}_k'^{ef}) = \lambda(s,v;\vec{D}'_k)$ for all $v\in V_c$. Observe that adding edges from V_c to s does not affect the edge-connectivity from s to any $v\in V_c$, so for all $v\in V_c$,

$$F(s,v;\vec{D}_k^{ef}) = \lambda(s,v;\vec{D}_k^{ef}) = \lambda(s,v;\vec{D}_k'^{ef}) = \lambda(s,v;\vec{D}'_k) = \lambda(s,v;\vec{D}_k) = F(s,v;\vec{D}_k).$$

The theorem trivially follows. □

Theorem 28. *Given an integer-capacity Eulerian digraph $D=(V_s\cup V_c,E)$ and $k\in\mathbb{N}$ with $\min_{v\in V_c} F(s,v;\vec{D}_k)\geq |V_c|k$, the maximum capacity that $e=(u,w), f=(w,t)$ can be split off with the resulting graph D^{ef} satisfying $\min_{v\in V_c} F(s,v;\vec{D}_k^{ef})\geq |V_c|k$ is*

$$\gamma = \min\left\{ c(u,w;D)\ ,\ c(w,t;D)\ ,\ \min_{v\in V_c} F(u,w;\widehat{D}_{(u,w),v}) - |V_c|k\ ,\ \min_{v\in V_c} F(w,t;\widehat{D}_{(w,t),v}) - |V_c|k \right\}. \tag{B.1}$$

Proof. First of all, one should note that for any s-v cut $(A,\bar{A})$ with $v\in V_c$ and $A\subset V_s\cup V_c+s$, if $s,u,t\in A\wedge v,w\in\bar{A}$, then $(A,\bar{A})$ has the same capacity in $\vec{D}_k$ and $\widehat{D}_{(u,w),v}$, i.e., $c(A,\bar{A};\vec{D}_k) = c(A,\bar{A};\widehat{D}_{(u,w),v})$. Similarly, if $s,w\in A\wedge v,u,t\in\bar{A}$, then $c(A,\bar{A};\vec{D}_k) = c(A,\bar{A};\widehat{D}_{(w,t),v})$.

$\geq$: Suppose we split off $(u,w),(w,t)$ by γ times and then $F(s,v';\vec{D}_k^{ef}) < |V_c|k$ for some $v'\in V_c$. Let $(A,\bar{A})$ be the min s-v' cut in $\vec{D}_k^{ef}$ that $c(A,\bar{A};\vec{D}_k^{ef}) = F(s,v';\vec{D}_k^{ef}) < |V_c|k$. We assert that $(A,\bar{A})$ must cut through (u,w) and (w,t) such that either $s,u,t\in A\wedge v', w\in\bar{A}$ or $s,w\in A\wedge v', u,t\in\bar{A}$; otherwise, we have $F(s,v';\vec{D}_k)\leq c(A,\bar{A};\vec{D}_k) = c(A,\bar{A};\vec{D}_k^{ef}) < |V_c|k$ (note that splitting off $(u,w),(w,t)$ adds edge (u,t)). Suppose

$s,u,t \in A \wedge v', w \in \bar{A}$, then $c(A,\bar{A};\vec{D}_k) = c(A,\bar{A};\widehat{D}_{(u,w),v'})$. It is trivial to see that $c(A,\bar{A};\vec{D}_k) = c(A,\bar{A};\vec{D}_k^{ef}) + \gamma$. Thus,

$$F(u,w;\widehat{D}_{(u,w),v'}) \le c(A,\bar{A};\widehat{D}_{(u,w),v'}) = c(A,\bar{A};\vec{D}_k) = c(A,\bar{A};\vec{D}_k^{ef}) + \gamma < |V_c|k + \gamma,$$

contradicting $\gamma \le \min_{v \in V_c} F(u,w;\widehat{D}_{(u,w),v}) - |V_c|k$. For $s,w \in A \wedge v', u,t \in \bar{A}$, one can similarly show a contradiction by looking at $F(w,t;\widehat{D}_{(w,t),v'})$.

$\le$: Suppose we split off $(u,w),(w,t)$ by $\gamma' > \gamma$ times and the resulting graph is D^{ef}. It is trivial to see that γ' cannot be greater than $c(u,w;D)$ or $c(w,t;D)$. Suppose $\gamma' > F(u,w;\widehat{D}_{(u,w),v'}) - |V_c|k$ for some $v' \in V_c$. Consider the min u-w cut $(A,\bar{A})$ with $c(A,\bar{A};\widehat{D}_{(u,w),v'}) = F(u,w;\widehat{D}_{(u,w),v'})$. Because $(u,s),(u,t),(v',w)$ have ∞ capacity, we have $s,u,t \in A \wedge v', w \in \bar{A}$ and hence $c(A,\bar{A};\vec{D}_k) = c(A,\bar{A};\widehat{D}_{(u,w),v'})$. It is again trivial to see that $c(A,\bar{A};\vec{D}_k^{ef}) = c(A,\bar{A};\vec{D}_k) - \gamma'$ and $(A,\bar{A})$ being an s-v' cut in $\vec{D}_k^{ef}$. Hence,

$$F(s,v';\vec{D}_k^{ef}) \le c(A,\bar{A};\vec{D}_k^{ef}) = c(A,\bar{A};\vec{D}_k) - \gamma' = c(A,\bar{A};\widehat{D}_{(u,w),v'}) - \gamma' < |V_c|k.$$

One can show similar result for $\gamma' > F(w,t;\widehat{D}_{(w,t),v'}) - |V_c|k$. □

Theorem 29 (Tarjan [136])**.** *For any integer-capacity digraph $D = (V,E)$ and any sets $R_i \subseteq V$, $i \in [k]$, there exist k edge-disjoint spanning out-trees T_i, $i \in [k]$, rooted respectively at R_i, if and only if for every $S \neq V$,*

$$c(S,\bar{S};D) \ge |\{i \mid R_i \subseteq S\}|. \tag{B.2}$$

Theorem 30. *Given an integer-capacity digraph $D = (V_c,E)$ and $k \in \mathbb{N}$, $c(S,\bar{S};D) \ge |S|k$ for all $S \subset V_c, S \neq V_c$ if and only if* $\min_{v \in V_c} F(s,v;\vec{D}_k) \ge |V_c|k$.

Proof. $\Rightarrow$: Suppose $\min_{v \in V_c} F(s,v;\vec{D}_k) < |V_c|k$. Let v' be the vertex that $F(s,v';\vec{D}_k) < |V_c|k$. By min-cut theorem, there exists an s-v' cut $(A,\bar{A})$ in $\vec{D}_k$ such that $c(A,\bar{A};\vec{D}_k) = F(s,v';\vec{D}_k) < |V_c|k$. Let $S = V_c \cap A$, then $A = S + s$, $\bar{S} = V_c - S = V_c + s - A = \bar{A}$, and hence

$$c(S,\bar{S};D) = c(A,\bar{A};\vec{D}_k) - \sum_{u \in \bar{A}} c(s,u;\vec{D}_k) < |V_c|k - |V_c - S|k = |S|k.$$

$\Leftarrow$: Suppose there exists $S \subset V_c, S \neq V_c$ such that $c(S,\bar{S};D) < |S|k$. Pick arbitrary $v' \in V_c - S$. Consider s-v' cut $(A,\bar{A})$ such that $A = S + s$. By min-cut theorem, we have

$$F(s,v';\vec{D}_k) \le c(A,\bar{A};\vec{D}_k) = c(S,\bar{S};D) + \sum_{u \in \bar{A}} c(s,u;\vec{D}_k) < |S|k + |V_c - S|k = |V_c|k.$$

□

Theorem 31 (Bérczi & Frank [13])**.** *Let $D=(V,E)$ be a digraph, $g: E \to \mathbb{Z}_+$ a capacity function, $\mathcal{R}=\{R_1,\ldots,R_n\}$ a list of root-sets, $\mathcal{U}=\{U_1,\ldots,U_n\}$ a set of convex sets with $R_i \subseteq U_i$, and $m: \mathcal{R} \to \mathbb{Z}_+$ a demand function. There is a strongly polynomial time algorithm that finds (if there exist) $m(\mathcal{R})$ out-trees so that $m(R_i)$ of them are spanning U_i with root-set R_i and each edge $e \in E$ is contained in at most $g(e)$ out-trees.*

Theorem 32. *For any edge (x,y) in D with $x \in R_1, y \notin R_1$,*

$$\mu = \min\left\{ g(x,y)\ ,\ m(R_1)\ ,\ F(x,y;\overline{D}) - \textstyle\sum_{i\neq 1} m(R_i) \right\}. \tag{B.4}$$

Proof. For simplicity of notation, let $L = \min\{c(S,\bar{S};D) - p(S;D) : x \in S, y \in \bar{S}, R_1 \not\subseteq S\}$. We will prove (B.4) by showing that either $L = F(x,y;\overline{D}) - \sum_{i\neq 1} m(R_i)$ or $L \geq F(x,y;\overline{D}) - \sum_{i\neq 1} m(R_i) \geq m(R_1)$. Let $S \subset V_c$ be arbitrary that $x \in S, y \in \bar{S}, R_1 \not\subseteq S$, and let $A = S \cup \{s_i \mid R_i \subseteq S\}$. It follows that $(A,\bar{A})$ is an x-y cut in $\overline{D}$ and hence

$$\begin{aligned} c(S,\bar{S};D) - p(S;D) &= c(S,\bar{S};D) - \textstyle\sum\{m(R_i) \mid R_i \subseteq S\} \\ &= c(S,\bar{S};D) + \textstyle\sum\{m(R_i) \mid i \neq 1, R_i \not\subseteq S\} - \sum_{i\neq 1} m(R_i) \\ &= c(A,\bar{A};\overline{D}) - \textstyle\sum_{i\neq 1} m(R_i) \\ &\geq F(x,y;\overline{D}) - \textstyle\sum_{i\neq 1} m(R_i). \end{aligned}$$

The second equality is due to $R_1 \not\subseteq S$, so $\sum\{m(R_i) \mid R_i \subseteq S\} = \sum\{m(R_i) \mid i \neq 1, R_i \subseteq S\}$. Since S is arbitrary, we have $L \geq F(x,y;\overline{D}) - \sum_{i\neq 1} m(R_i)$.

Let $(A',\overline{A'})$ be the min x-y cut in $\overline{D}$ and $S' = A' \cap V_c$. We assert that for any $i \neq 1, R_i \subseteq S'$, we have $s_i \in A'$; otherwise, by moving s_i from $\overline{A'}$ to A', we create a cut with lower capacity, contradicting $(A',\overline{A'})$ being min-cut. We also assert that for any $i \neq 1, R_i \not\subseteq S'$, we have $s_i \in \overline{A'}$; otherwise, there exists $v \in R_i - S'$ that ∞ edge (s_i, v) crosses $(A',\overline{A'})$. Thus, we have

$$\begin{aligned} F(x,y;\overline{D}) - \textstyle\sum_{i\neq 1} m(R_i) =& c(A',\overline{A'};\overline{D}) - \textstyle\sum_{i\neq 1} m(R_i) \\ =& c(S',\overline{S'};D) + \textstyle\sum\{m(R_i) \mid i \neq 1, R_i \not\subseteq S'\} - \sum_{i\neq 1} m(R_i) \\ =& c(S',\overline{S'};D) - \textstyle\sum\{m(R_i) \mid i \neq 1, R_i \subseteq S'\}. \end{aligned} \tag{B.7}$$

Now, we consider two cases:

(a) Suppose $R_1 \not\subseteq S'$. Then, $c(S',\overline{S'};D) - p(S';D) \geq L$. By (B.7), we have

$$\begin{aligned} L &\geq F(x,y;\overline{D}) - \textstyle\sum_{i\neq 1} m(R_i) \\ &= c(S',\overline{S'};D) - \textstyle\sum\{m(R_i) \mid i \neq 1, R_i \subseteq S'\} \end{aligned}$$

$$= c(S', \overline{S'}; D) - p(S'; D)$$

Thus, $L = c(S', \overline{S'}; D) - p(S'; D) = F(x, y; \overline{D}) - \sum_{i \neq 1} m(R_i)$ and (B.4) holds.

(b) Suppose $R_1 \subseteq S'$. Because the existence of spanning trees is guaranteed, we have

$$c(S', \overline{S'}; D) \geq p(S'; D) = m(R_1) + \sum \{m(R_i) \mid i \neq 1, R_i \subseteq S'\}.$$

Hence,

$$\begin{aligned} L &\geq F(x, y; \overline{D}) - \sum_{i \neq 1} m(R_i) \\ &= c(S', \overline{S'}; D) - \sum \{m(R_i) \mid i \neq 1, R_i \subseteq S'\} \\ &\geq m(R_1). \end{aligned}$$

Thus, $\mu = \min\{g(x, y), m(R_1)\}$ and (B.4) also holds.

□

Theorem 33. *Given $U \in \mathbb{R}_+$ and $k \in \mathbb{N}$, a family of out-trees $\{T_{u,i}\}_{u \in V_c, i \in [k]}$ with $T_{u,i}$ rooted at u and $\mathcal{V}(T_{u,i}) \supseteq V_c$ achieves $\frac{M}{Nk} \cdot U$ communication time if and only if it is edge-disjoint in $G(\{\lfloor U b_e \rfloor\}_{e \in E})$.*

Proof. $\Leftarrow$: Suppose $\{T_{u,i}\}_{u \in V_c, i \in [k]}$ is edge disjoint in $G(\{\lfloor U b_e \rfloor\}_{e \in E})$, then

$$T_{\text{comm}} = \frac{M}{Nk} \cdot \max_{e \in E} \frac{1}{b_e} \sum_{T \in \{T_{u,i}\}} \mathbb{I}[e \in T] \leq \frac{M}{Nk} \cdot \max_{e \in E} \frac{\lfloor U b_e \rfloor}{b_e} \leq \frac{M}{Nk} \cdot U.$$

$\Rightarrow$: Suppose $\{T_{u,i}\}_{u \in V_c, i \in [k]}$ achieves $\frac{M}{Nk} \cdot U$ communication time, then

$$\max_{e \in E} \frac{1}{b_e} \sum_{T \in \{T_{u,i}\}} \mathbb{I}[e \in T] \leq U \quad \Longrightarrow \quad \sum_{T \in \{T_{u,i}\}} \mathbb{I}[e \in T] \leq U b_e \quad \text{for all } e \in E.$$

Since $\sum_{T \in \{T_{u,i}\}} \mathbb{I}[e \in T]$ must be an integer, the edge-disjointness trivially follows. □

Theorem 34. *Let $\frac{M}{Nk} \cdot U^*$ be the lowest communication time that can be achieved with k out-trees per $v \in V_c$. Then, there exists a family of edge-disjoint out-trees $\{T_{u,i}\}_{u \in V_c, i \in [k]}$ in $G(\{\lfloor U b_e \rfloor\}_{e \in E})$ with $T_{u,i}$ rooted at u and $\mathcal{V}(T_{u,i}) \supseteq V_c$ if and only if $U \geq U^*$.*

Proof. $\Rightarrow$: The existence of edge-disjoint $\{T_{u,i}\}_{u \in V_c, i \in [k]}$ in $G(\{\lfloor U b_e \rfloor\}_{e \in E})$ with $U < U^*$ simply contradicts $\frac{M}{Nk} \cdot U^*$ being the lowest communication time. $\Leftarrow$: Let $\{T^*_{u,i}\}_{u \in V_c, i \in [k]}$ be the family of out-trees with the lowest communication time, then by Theorem 33, it is edge-disjoint in $G(\{\lfloor U b_e \rfloor\}_{e \in E})$ for all $U \geq U^*$. □

Theorem 35. *Let $\frac{M}{Nk} \cdot U^*$ be the lowest communication time that can be achieved with k out-trees per $v \in V_c$. Then,*

$$\frac{M}{Nk} \cdot U^* \leq \frac{M}{N} \max_{S \subset V, S \not\supseteq V_c} \frac{|S \cap V_c|}{B_G^+(S)} + \frac{M}{Nk} \cdot \frac{1}{\min_{e \in E} b_e}.$$

Proof. Let $U = \max_{e \in E} \lceil k b_e / x^* \rceil / b_e$ where $1/x^* = \max_{S \subset V, S \not\supseteq V_c} |S \cap V_c| / B_G^+(S)$. For each edge (u,v) in $G(\lfloor U b_e \rfloor)$, we have

$$c(u,v; G(\lfloor U b_e \rfloor)) = \left\lfloor b_{(u,v)} \cdot \max_{e \in E} \frac{\lceil k b_e / x^* \rceil}{b_e} \right\rfloor \geq \left\lfloor b_{(u,v)} \cdot \frac{\lceil k b_{(u,v)} / x^* \rceil}{b_{(u,v)}} \right\rfloor = \lceil k b_{(u,v)} / x^* \rceil.$$

Thus, each edge in $\vec{G}_k(\lfloor U b_e \rfloor)$ has at least k/x^* times the capacity in $\vec{G}_{x^*}$, so

$$\min_{v \in V_c} F(s,v; \vec{G}_k(\lfloor U b_e \rfloor)) \geq (k/x^*) \min_{v \in V_c} F(s,v; \vec{G}_{x^*}) \geq |V_c| k.$$

Therefore, $\frac{M}{Nk} \cdot U$ is achievable and hence $U^* \leq U$ by Theorem 34.

$$\frac{U^*}{k} \Big/ \frac{1}{x^*} \leq \frac{U}{k} \Big/ \frac{1}{x^*} = \frac{\max_{e \in E} \lceil k b_e / x^* \rceil / b_e}{k/x^*} = \max_{e \in E} \frac{\lceil k b_e / x^* \rceil}{k b_e / x^*} \leq 1 + \max_{e \in E} \frac{1}{k b_e / x^*} = 1 + \frac{x^*}{k \cdot \min_{e \in E} b_e}.$$

The theorem trivially follows. □