USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

# ForestColl: Throughput-Optimal Collective Communications on Heterogeneous Network Fabrics

Liangyu Zhao, *University of Washington;* Saeed Maleki, *Independent Researcher;* Yuanhong Wang, *Tsinghua University;* Zezhou Wang, *University of Washington;* Ziyue Yang, *Microsoft Research;* Hossein Pourreza, *Microsoft;* Arvind Krishnamurthy, *University of Washington*

https://www.usenix.org/conference/nsdi26/presentation/zhao-liangyu

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

# ForestColl: Throughput-Optimal Collective Communications on Heterogeneous Network Fabrics

Liangyu Zhao*
*University of Washington*

Saeed Maleki†
*Independent Researcher*

Yuanhong Wang‡
*Tsinghua University*

Zezhou Wang
*University of Washington*

Ziyue Yang
*Microsoft Research*

Hossein Pourreza
*Microsoft*

Arvind Krishnamurthy
*University of Washington*

## Abstract

As modern DNN models grow ever larger, collective communications between the accelerators (allreduce, etc.) emerge as a significant performance bottleneck. Designing efficient communication schedules is challenging, given today's heterogeneous and diverse network fabrics. We present ForestColl, a tool that generates throughput-optimal schedules for any network topology. ForestColl constructs broadcast/aggregation spanning trees as the communication schedule, achieving theoretical optimality. Its schedule generation runs in polynomial time and is highly scalable. ForestColl supports any network fabric, including both switching fabrics and direct accelerator connections. We evaluated ForestColl on AMD MI250 and NVIDIA DGX A100 & H100 clusters. ForestColl shows significant improvements over the vendors' own optimized communication libraries across various settings and in LLM training. ForestColl also outperforms other state-of-the-art schedule generation techniques with both more efficient generated schedules and substantially faster generation speed.

## 1 Introduction

Collective communications have become a cornerstone of distributed machine learning training [26, 65, 68]. As large language models (LLMs) scale to hundreds of billions of parameters [42, 48, 54, 70], their training demands immense volumes of collective communication traffic, creating a performance bottleneck [24, 34, 57, 58, 68, 76, 78, 85]. Operational insights from AI hyperscalers (e.g., Meta [24], Alibaba [58], AWS [23]) 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 [4, 5, 51–53].

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 homogeneously (i.e., uniformly across the fabric) is both technically challenging [6, 28, 29] and prohibitively costly [77, 78], hardware providers adopt

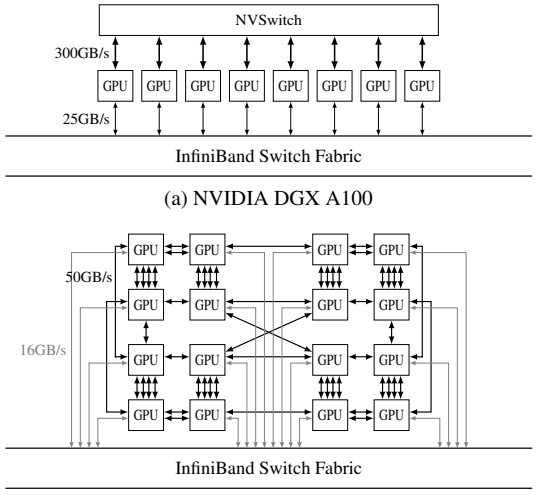
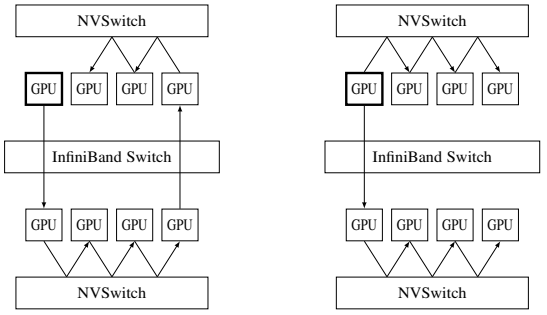



**Figure 1: Network Topologies of NVIDIA DGX A100 and AMD MI250.** The PCIe switches and IB NICs are omitted for simplicity.

*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 1 shows the network topologies of NVIDIA DGX A100 [52] 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 [44, 77].

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 [26] 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 underuti-

*The work was partially done during an internship at Microsoft Research.
†The work was done at Microsoft Research.
‡The work was done at the University of Washington.




(a) Ring (b) ForestColl

**Figure 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.

lizes the faster intra-box bandwidth (see example in Figure 2). Further, existing static algorithms (e.g., ring, recursive halving/doubling, Bruck algorithm, BlueConnect) [16,59] 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 [10], TACCL [66], TE-CCL [41], TACOS [80], BFB [82], Blink [71], MultiTree [30], TTO [36], SyCCL [11]) 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:

(i) **Optimality:** To the best of our knowledge, ForestColl is the first work capable of deriving the optimal throughput of any topology (§4 & 5.2) and generating optimal schedules for *reduce-scatter*, *allgather*, and *allreduce*.

(ii) **Generality:** ForestColl introduces a novel transformation for switch topologies that supports their flexible traffic patterns while preserving optimal throughput (§5.3). ForestColl also leverages in-network multicast/aggregation capabilities when supported by switches (§5.6).

(iii) **Scalability:** Every part of ForestColl runs in polynomial time (Appendix F), scalable to large topologies (§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.

## 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 (§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.

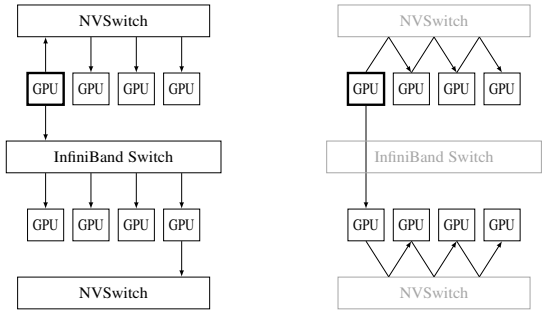
**Related Work:** Apart from the scalability consideration discussed earlier, none of the existing methods simultaneously achieves both optimality and generality. SCCL can achieve optimality for a given number of data chunks, but the optimal chunking is unknown, and it does not support switch topologies. 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 (§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 2 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.

## 3 Overview of ForestColl

We now introduce ForestColl. To ensure throughput optimality for throughput-bound LLM training [23, 24, 38, 58], 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 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 [7,8,21,64,69]. 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 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 topology and found that it is determined by the *throughput bottleneck cut*—the network cut with the highest ratio of minimal required traf-

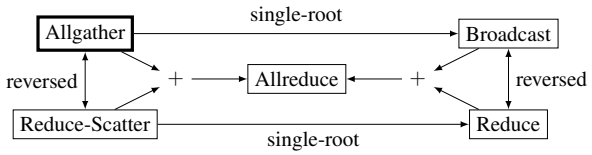



**Figure 4: Relationships between collective operations.** Reduce and reduce-scatter can be constructed by reversing the communications of broadcast and allgather [12]. Reduce and broadcast are single-root versions of reduce-scatter and allgather. Finally, allreduce can be performed via reduce-scatter plus allgather or reduce plus broadcast. While this paper focuses on allgather, the method applies to other operations.

fic to available bandwidth (§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 (§5.2). Once edge capacities are scaled accordingly, the tree packing algorithm can be applied to construct the trees (§5.4). For switches, we devised a way to transform a switch topology into a switch-free logical topology before applying tree packing (§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.

## 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 shows the relationships between operations. While we explain ForestColl in the context of allgather, it can be easily applied to other operations (§5.7).

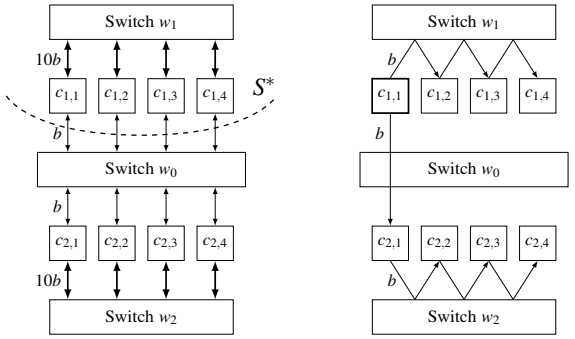



**Figure 5: Example of Spanning Out-Tree.** (a) shows a 2-box 8-compute-node switch topology along with the throughput bottleneck cut. The intra-box connections (thick lines) have 10x the bandwidth of inter-box ones (thin lines). (b) shows one example of ForestColl's spanning out-trees rooted at $c_{1,1}$.

Knowing the throughput optimality of a given network is crucial for optimizing schedules. Previous work often defines optimality as $\frac{M(N-1)}{N}\cdot\beta$ [10, 12, 82], where $\frac{M(N-1)}{N}$ is the amount of data each node must receive in allgather, and $\beta$ 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 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 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 $T_{\text{comm}}$ in topology $G$. Consider all such cuts in $G$, then $T_{\text{comm}}$ 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.

## 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 3. Figure 5(b) shows an example of the out-tree with allocated bandwidth $b$.

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

### 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* [7, 8, 21, 64, 69] to construct the optimal set of out-trees. In a switch topology, however, we retrofit the *edge splitting technique* [7, 22, 31] 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.

### 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 $T_{\text{comm}}$, 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 ($\star$).

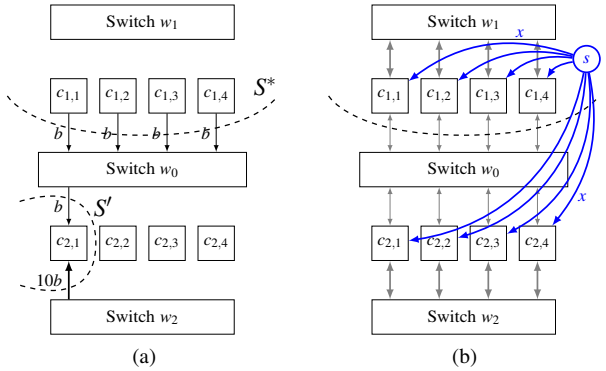



**Figure 6: The auxiliary network for optimality binary search.** (a) shows two cuts: $S^*, S'$, along with their exiting bandwidths. Note that $S'$ is $V - c_{2,1}$ instead of $\{c_{2,1}\}$. (b) shows the auxiliary network that there exists a set of spanning out-trees broadcasting $x$ amount of flow from each compute node if and only if the maxflow from $s$ to every compute node is $Nx$.

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 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.

[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.

**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 $\frac{1}{x^*}$

**Auxiliary Network:** To address the above issues, we construct an auxiliary network as in Figure 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 $B^+(S^*)$, 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 E.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 $\frac{1}{x^*} = \frac{p}{q}$. 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 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 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 E.1.

### 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 [66] and TACOS [80] 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 [7, 22, 31], we adapt edge splitting to handle switch topologies in the context of collective communications. Starting with the scaled topology as in Figure 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 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 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.

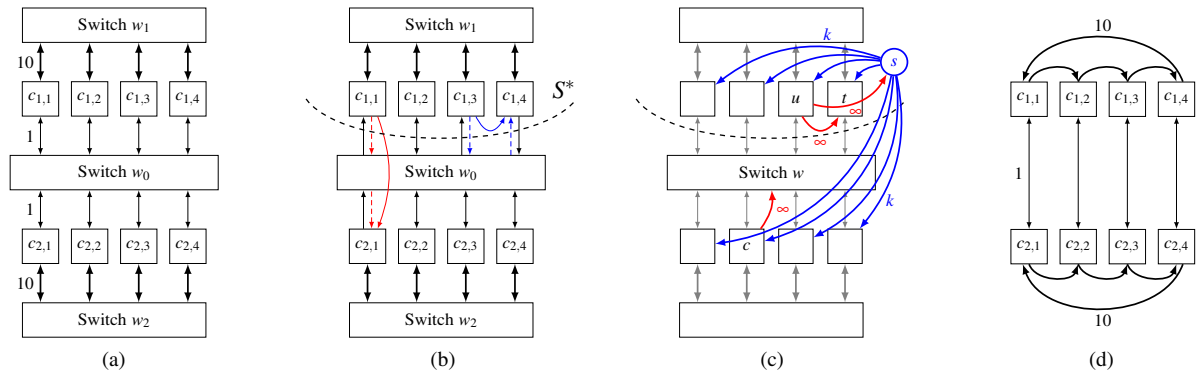



**Figure 7: Figures explaining the switch node removal process.** (a) is the starting topology after optimality binary search scales edge capacities from $\{b, 10b\}$ to $\{1, 10\}$. (b) contains examples of replacing the ingress and egress capacities of a switch node with a direct capacity bypassing the switch. (c) shows an example of the auxiliary network ForestColl uses to compute the $\gamma$ in Algorithm 2. The $\infty$ 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.

However, arbitrarily choosing an ingress link may lead to performance sacrifice. In the two examples of Figure 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 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 $\gamma$ 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 7(c) shows the auxiliary network we use to compute $\gamma$. 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 $\gamma$ in Theorem 6 in Appendix E.2.

Algorithm 2 shows the pseudocode of the switch node removal process. For each switch node $w$ and each egress edge $f=(w,t)$, we pair it with each ingress edge $e=(u,w)$ and calculate $\gamma$, 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 E.2 provides more details of the algorithm.

### 5.4 Spanning Tree Construction

Given the switch-free logical topology like Figure 7(d), ForestColl applies the spanning out-tree packing algorithm [8, 64]. 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 $\mu$ of the edge that can be added to the out-tree while maintaining the feasibility of constructing the remaining trees. If $\mu$ is less than the tree's capacity $m$, the algorithm splits the tree into two: one with capacity $m-\mu$ and another with capacity $\mu$, adding the edge to

**Algorithm 2:** Switch Node Removal

```
Input: A directed graph G = (V_s ∪ V_c, E) and k.
Output: A directed graph H = (V_c, E').
begin
  foreach switch node w ∈ V_s do
    foreach egress edge f = (w,t) ∈ E do
      foreach ingress edge e = (u,w) ∈ E do
        Compute γ, the maximum capacity we can safely
          replace f,e by (u,t), as in Theorem 6.
        if γ = 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) ∉ 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
```

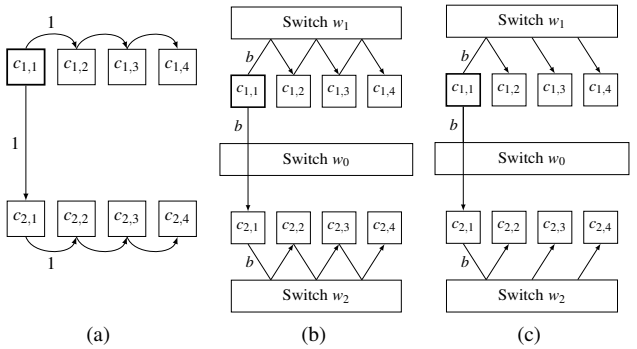



**Figure 8: The constructed spanning out-tree.** (a) shows one example of the spanning out-trees generated by applying the spanning tree packing algorithm to Figure 7(d). (b) shows the corresponding tree after mapping (a) back to the original topology. (c) shows the post-processed tree utilizing the in-network multicast/aggregation capabilities of switches $w_1, w_2$.

**Table 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.

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

the latter. Appendix E.3 describes the details of the algorithm.

Figure 8(a) shows one example of the spanning out-trees constructed by applying the algorithm to Figure 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 8(b).

### 5.5 Fixed-$k$ Schedule Generation

In §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 §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 E.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 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.

### 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, 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 8 gives an example. In Figure 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 ($c_{2,2}, c_{2,3} \to w_2$). As the data being sent is the same throughout the tree, such traffic can be deleted, resulting in the tree in Figure 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.

### 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 G. 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.

## 6 Evaluation

We present a comprehensive evaluation of ForestColl. §6.1 describes our implementations of ForestColl's schedules. §6.2 compares the performance of various schedules on both AMD and NVIDIA hardware. §6.3 evaluates ForestColl against NCCL [50] on a large-scale GPU cluster. §6.4 presents results from LLM training with PyTorch FSDP. Finally, §6.5 compares methods for large-scale schedule generation.

### 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 [47] 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++ [67] 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.

### 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 XMLs, as both are tree-flow schedules. Finally, on AMD hardware, we compared against RCCL [62] (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 (§6.5).

#### 6.2.1 AMD MI250 Experiments

**Testbed Setup:** The AMD MI250's complex topology presents significant challenges for schedule generation. Figure 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 9(a). The 8+8 setting can result from hybrid training parallelism or bin-packing jobs in a cloud environment. Figure 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 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 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.

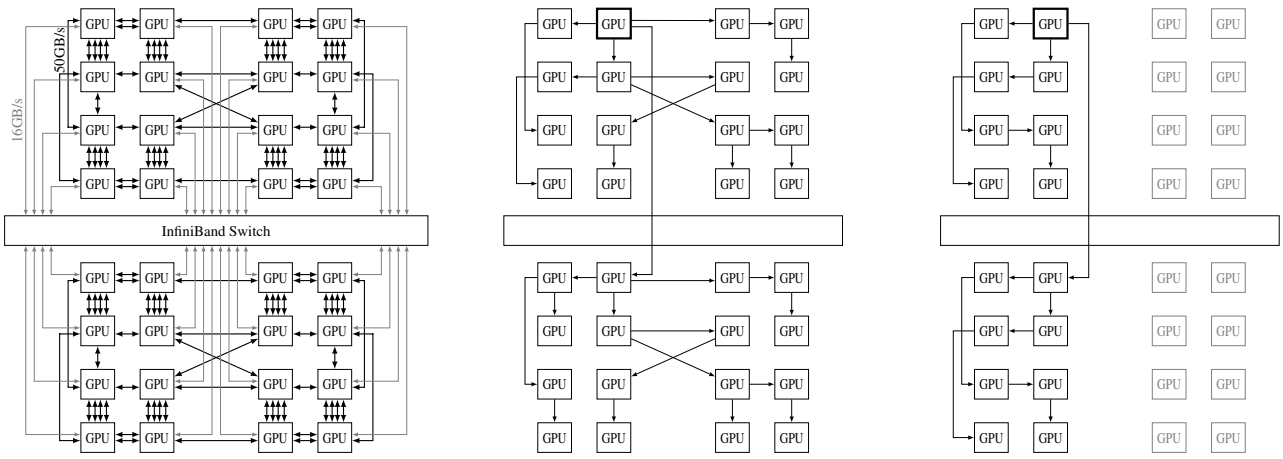
**8+8 Results:** In 8+8 setting, ForestColl also outperforms the baselines. The comparison between ForestColl and TACCL remains similar, with ForestColl being 43% faster at 1GB and 32% faster on average from 1MB to 1GB. Against Blink+Switch, ForestColl is 36% faster both at 1GB and on average. RCCL's 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.

#### 6.2.2 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 11 presents our experiment results. ForestColl leads in all three collectives by a considerable margin

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




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

**Figure 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.

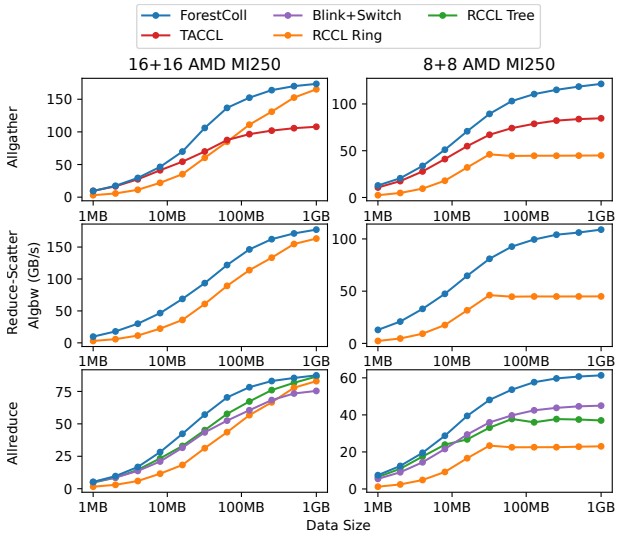



**Figure 10: Comparing collective communication performance of TACCL, Blink+Switch, RCCL, and ForestColl in 16+16 and 8+8 settings on 2-box AMD MI250.** The columns and rows correspond to different settings and collectives, respectively. "Blink+Switch" represents Blink augmented with our switch removal technique, enabling it to support switches.

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.

In addition to comparing NCCL and ForestColl, we implemented the NCCL ring as MSCCL XML and tested its performance. In Figure 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.

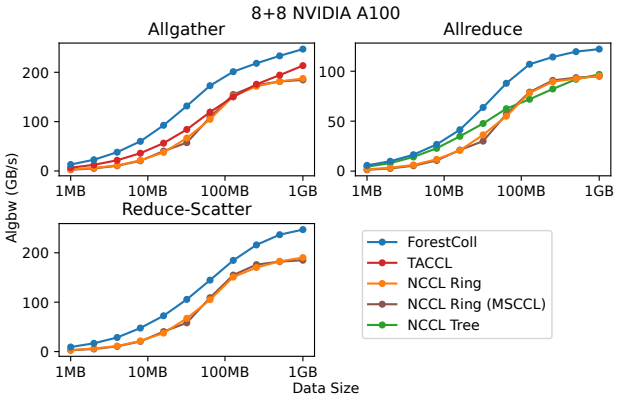
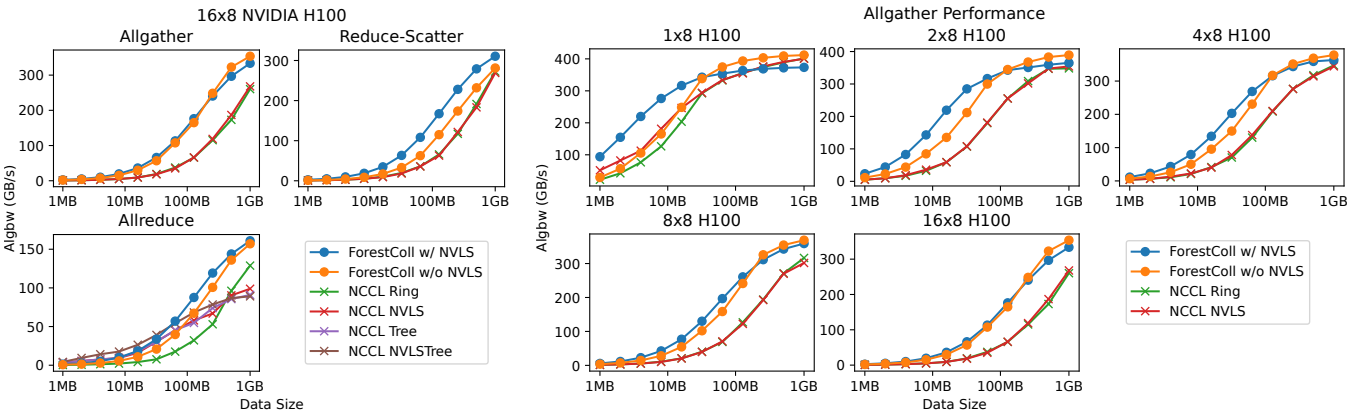



**Figure 11: Comparing collective communication performance of TACCL, NCCL, and ForestColl on 2-box NVIDIA DGX A100.** The "NCCL Ring (MSCCL)" is NCCL ring implemented in MSCCL XMLs to confirm no inherent performance difference between NCCL and MSCCL.

### 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 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 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




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

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

**Figure 12: Comparing NCCL and ForestColl collective communication performance on NVIDIA DGX H100.**

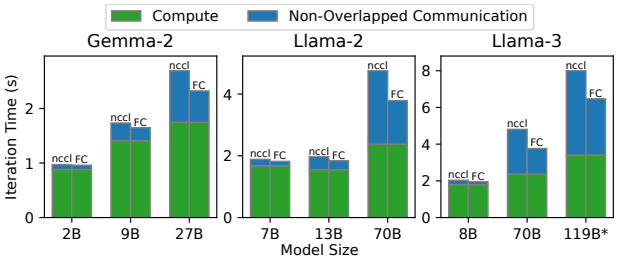



**Figure 13: Comparing NCCL and ForestColl in Fully Sharded Data Parallel (FSDP) training.** The training is on 2x DGX A100 with 16 GPUs using PyTorch FSDP [83]. The compute times are measured by training with communications skipped. Given the limited scale of our testbed, context lengths are set to 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 [79] and use FlashAttention [18, 19] with BFloat16.

low-latency schedules and ForestColl depending on the input data size, allowing the two to complement each other.

**1-16x8 Results:** Figure 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.

### 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 [61, 83] with open-source LLMs: Gemma-2 [25] from Google and Llama-2 [45] & 3 [42] from Meta. FSDP is widely used for training large models that far exceed the memory capacity of a single GPU [2, 42, 45]. 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 §6.2.2. Figure 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 [18, 19] 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, forcing sequential execution.

[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.

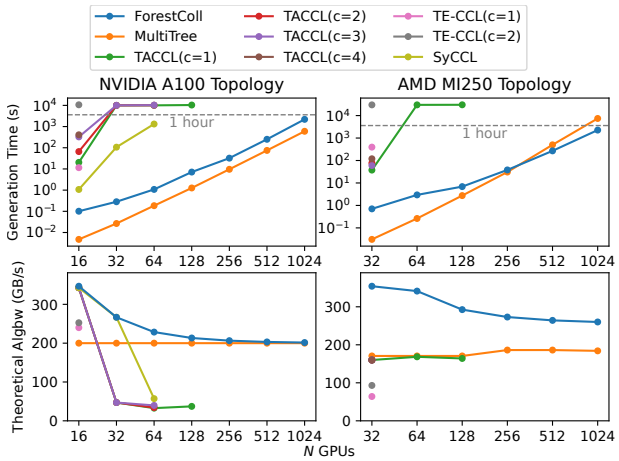



**Figure 14: Large-scale schedule generation comparison between MultiTree, TACCL, TE-CCL, SyCCL, and ForestColl on NVIDIA A100 and AMD MI250 topologies.** The top row compares the time spent on generation, and the bottom row compares the theoretical algorithmic bandwidth of the generated schedules. TACCL and TE-CCL are run with varying numbers of chunks. The time limit is set to $10^4$s for A100 and $3\times10^4$s for MI250.

### 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 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 C 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 $10^4$s time limit for A100 topologies and $3\times10^4$s (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 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 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.

## 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.

## 8 Acknowledgments

We thank the anonymous reviewers and our shepherd for their insightful feedback and guidance. We thank our collaborators at Microsoft Research, including Madan Musuvathi, Aashaka Shah, and Changho Hwang, for their support in the project. This work was also supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

## References


[1] AGRAWAL, A., AGA, S., PATI, S., AND ISLAM, M. Optimizing ml concurrent computation and communication with gpu dma engines, 2024.

[2] AI2. Olmo: Accelerating the science of language models, 2024.

[3] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication* (New York, NY, USA, 2008), SIGCOMM '08, Association for Computing Machinery, p. 63–74.

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

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

[6] BAI, W., ABDEEN, S. S., AGRAWAL, A., ATTRE, K. K., BAHL, P., BHAGAT, A., BHASKARA, G., BROKHMAN, T., CAO, L., CHEEMA, A., CHOW, R., COHEN, J., ELHADDAD, M., ETTE, V., FIGLIN, I., FIRESTONE, D., GEORGE, M., GERMAN, I., GHAI, L., GREEN, E., GREENBERG, A., GUPTA, M., HAAGENS, R., HENDEL, M., HOWLADER, R., JOHN, N., JOHNSTONE, J., JOLLY, T., KRAMER, G., KRUSE, D., KUMAR, A., LAN, E., LEE, I., LEVY, A., LIPSHTEYN, M., LIU, X., LIU, C., LU, G., LU, Y., LU, X., MAKHERVAKS, V., MALASHANKA, U., MALTZ, D. A., MARINOS, I., MEHTA, R., MURTHI, S., NAMDHARI, A., OGUS, A., PADHYE, J., PANDYA, M., PHILLIPS, D., POWER, A., PURI, S., RAINDEL, S., RHEE, J., RUSSO, A., SAH, M., SHERIFF, A., SPARACINO, C., SRIVASTAVA, A., SUN, W., SWANSON, N., TIAN, F., TOMCZYK, L., VADLAMURI, V., WOLMAN, A., XIE, Y., YOM, J., YUAN, L., ZHANG, Y., AND ZILL, B. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (Boston, MA, Apr. 2023), USENIX Association, pp. 49–67.

[7] BANG-JENSEN, J., FRANK, A., AND JACKSON, B. Preserving and increasing local edge-connectivity in mixed graphs. *SIAM Journal on Discrete Mathematics 8*, 2 (1995), 155–178.

[8] BÉRCZI, K., AND FRANK, A. Packing arborescences (combinatorial optimization and discrete algorithms). *RIMS Kokyuroku Bessatsu B23* (2010), 1–31.

[9] BERMOND, J.-C., AND FRAIGNIAUD, P. Broadcasting and NP-completeness. In *Graph Theory Notes of New York* (1992), vol. XXII, pp. 8–14.

[10] CAI, Z., LIU, Z., MALEKI, S., MUSUVATHI, M., MYTKOWICZ, T., NELSON, J., AND SAARIKIVI, O. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2021), PPoPP '21, Association for Computing Machinery, p. 62–75.

[11] CAO, J., SHI, S., GAO, J., LIU, W., YANG, Y., XU, Y., ZHENG, Z., GUAN, Y., QIAN, K., LIU, Y., XU, M., WANG, T., WANG, N., DONG, J., FU, B., CAI, D., AND ZHAI, E. Syccl: Exploiting symmetry for efficient collective communication scheduling. In *Proceedings of the ACM SIGCOMM 2025 Conference* (New York, NY, USA, 2025), SIGCOMM '25, Association for Computing Machinery, p. 645–662.

[12] CHAN, E., HEIMLICH, M., PURKAYASTHA, A., AND VAN DE GEIJN, R. Collective communication: Theory, practice, and experience: Research articles. *Concurr. Comput. : Pract. Exper. 19*, 13 (sep 2007), 1749–1783.

[13] CHANG, L.-W., BAO, W., HOU, Q., JIANG, C., ZHENG, N., ZHONG, Y., ZHANG, X., SONG, Z., YAO, C., JIANG, Z., LIN, H., JIN, X., AND LIU, X. Flux: Fast software-based communication overlap on gpus through kernel fusion, 2024.

[14] CHEN, C., LI, X., ZHU, Q., DUAN, J., SUN, P., ZHANG, X., AND YANG, C. 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* (New York, NY, USA, 2024), ASPLOS '24, Association for Computing Machinery, p. 178–191.

[15] CHEN, Z., LIU, X., LI, M., HU, Y., MEI, H., XING, H., WANG, H., SHI, W., LIU, S., AND XU, Y. Rina: Enhancing ring-allreduce with in-network aggregation in distributed model training. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP)* (2024), pp. 1–12.

[16] CHO, M., FINKLER, U., KUNG, D., AND HUNTER, H. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. In *Proceedings of Machine Learning and Systems* (2019), A. Talwalkar, V. Smith, and M. Zaharia, Eds., vol. 1, pp. 241–251.

[17] CHO, S., SON, H., AND KIM, J. Logical/physical topology-aware collective communication in deep learning training. In *2023 IEEE International Symposium*

*on High-Performance Computer Architecture (HPCA)* (2023), pp. 56–68.

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

[19] DAO, T., FU, D. Y., ERMON, S., RUDRA, A., AND RÉ, C. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)* (2022).

[20] DE SENSI, D., DI GIROLAMO, S., ASHKBOOS, S., LI, S., AND HOEFLER, T. Flare: flexible in-network allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2021), SC '21, Association for Computing Machinery.

[21] EDMONDS, J. Edge-disjoint branchings. *Combinatorial algorithms* (1973), 91–96.

[22] FRANK, A. On connectivity properties of eulerian digraphs. In *Graph Theory in Memory of G.A. Dirac*, L. D. Andersen, I. T. Jakobsen, C. Thomassen, B. Toft, and P. D. Vestergaard, Eds., vol. 41 of *Annals of Discrete Mathematics*. Elsevier, 1988, pp. 179–194.

[23] FU, X., ZHANG, Z., FAN, H., HUANG, G., EL-SHABANI, M., HUANG, R., SOLANKI, R., WU, F., DIAMANT, R., AND WANG, Y. Distributed training of large language models on aws trainium. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (New York, NY, USA, 2024), SoCC '24, Association for Computing Machinery, p. 961–976.

[24] GANGIDI, A., MIAO, R., ZHENG, S., BONDU, S. J., GOES, G., MORSY, H., PURI, R., RIFTADI, M., SHETTY, A. J., YANG, J., ZHANG, S., FERNANDEZ, M. J., GANDHAM, S., AND ZENG, H. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference* (New York, NY, USA, 2024), ACM SIGCOMM '24, Association for Computing Machinery, p. 57–70.

[25] GEMMA TEAM, GOOGLE DEEPMIND. Gemma 2: Improving open language models at a practical size, 2024.

[26] GIBIANSKY, A. Bringing hpc techniques to deep learning. *Baidu Research, Tech. Rep.* (2017). https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/.

[27] GOLDBERG, A. V., AND TARJAN, R. E. A new approach to the maximum-flow problem. *J. ACM 35*, 4 (oct 1988), 921–940.

[28] GUO, C., WU, H., DENG, Z., SONI, G., YE, J., PADHYE, J., AND LIPSHTEYN, M. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, Association for Computing Machinery, p. 202–215.

[29] HU, S., ZHU, Y., CHENG, P., GUO, C., TAN, K., PADHYE, J., AND CHEN, K. 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* (New York, NY, USA, 2016), HotNets '16, Association for Computing Machinery, p. 92–98.

[30] HUANG, J., MAJUMDER, P., KIM, S., MUZAHID, A., YUM, K. H., AND KIM, E. J. Communication algorithm-architecture co-design for distributed deep learning. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (2021), ISCA '21, IEEE Press, p. 181–194.

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

[32] JANGDA, A., HUANG, J., LIU, G., SABET, A. H. N., MALEKI, S., MIAO, Y., MUSUVATHI, M., MYTKOWICZ, T., AND SAARIKIVI, O. 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* (New York, NY, USA, 2022), ASPLOS '22, Association for Computing Machinery, p. 402–416.

[33] JIA, Z., ZAHARIA, M., AND AIKEN, A. Beyond data and model parallelism for deep neural networks. In *Proceedings of Machine Learning and Systems* (2019), A. Talwalkar, V. Smith, and M. Zaharia, Eds., vol. 1, pp. 1–13.

[34] JIANG, Z., LIN, H., ZHONG, Y., HUANG, Q., CHEN, Y., ZHANG, Z., PENG, Y., LI, X., XIE, C., NONG, S., JIA, Y., HE, S., CHEN, H., BAI, Z., HOU, Q., YAN, S., ZHOU, D., SHENG, Y., JIANG, Z., XU, H., WEI, H., ZHANG, Z., NIE, P., ZOU, L., ZHAO, S., XIANG, L., LIU, Z., LI, Z., JIA, X., YE, J., JIN, X., AND LIU, X. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)* (Santa Clara, CA, Apr. 2024), USENIX Association, pp. 745–760.

[35] LAO, C., LE, Y., MAHAJAN, K., CHEN, Y., WU, W., AKELLA, A., AND SWIFT, M. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX*

*Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association, pp. 741–761.

[36] LASKAR, S., MAJHI, P., KIM, S., MAHMUD, F., MUZAHID, A., AND KIM, E. J. Enhancing collective communication in mcm accelerators for deep learning training. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2024), pp. 1–16.

[37] LI, S., ZHAO, Y., VARMA, R., SALPEKAR, O., NOORDHUIS, P., LI, T., PASZKE, A., SMITH, J., VAUGHAN, B., DAMANIA, P., AND CHINTALA, S. PyTorch distributed: experiences on accelerating data parallel training. *Proc. VLDB Endow. 13*, 12 (Aug. 2020), 3005–3018.

[38] LI, W., LIU, X., LI, Y., JIN, Y., TIAN, H., ZHONG, Z., LIU, G., ZHANG, Y., AND CHEN, K. Understanding communication characteristics of distributed training. In *Proceedings of the 8th Asia-Pacific Workshop on Networking* (New York, NY, USA, 2024), APNet '24, Association for Computing Machinery, p. 1–8.

[39] LIN, Z., MIAO, Y., ZHANG, Q., YANG, F., ZHU, Y., LI, C., MALEKI, S., CAO, X., SHANG, N., YANG, Y., XU, W., YANG, M., ZHANG, L., AND ZHOU, L. nnScaler: Constraint-Guided parallelization plan generation for deep learning training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)* (Santa Clara, CA, July 2024), USENIX Association, pp. 347–363.

[40] LIU, S., WANG, Q., ZHANG, J., WU, W., LIN, Q., LIU, Y., XU, M., CANINI, M., CHEUNG, R. C. C., AND HE, J. 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* (New York, NY, USA, 2023), ASPLOS 2023, Association for Computing Machinery, p. 376–391.

[41] LIU, X., ARZANI, B., KAKARLA, S. K. R., ZHAO, L., LIU, V., CASTRO, M., KANDULA, S., AND MARSHALL, L. Rethinking machine learning collective communication as a multi-commodity flow problem. In *Proceedings of the ACM SIGCOMM 2024 Conference* (New York, NY, USA, 2024), ACM SIGCOMM '24, Association for Computing Machinery, p. 16–37.

[42] LLAMA TEAM, AI @ META. The llama 3 herd of models, 2024.

[43] MAHAJAN, K., CHU, C.-H., SRIDHARAN, S., AND AKELLA, A. Better together: Jointly optimizing ML collective scheduling and execution planning using SYNDICATE. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (Boston, MA, Apr. 2023), USENIX Association, pp. 809–824.

[44] MANDAKOLATHUR, K., AND JEAUGEY, S. 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/.

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

[46] MICHAIL, D., KINABLE, J., NAVEH, B., AND SICHI, J. V. Jgrapht—a java library for graph data structures and algorithms. *ACM Trans. Math. Softw. 46*, 2 (May 2020).

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

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

[49] NARAYANAN, D., SHOEYBI, M., CASPER, J., LEGRESLEY, P., PATWARY, M., KORTHIKANTI, V., VAINBRAND, D., KASHINKUNTI, P., BERNAUER, J., CATANZARO, B., PHANISHAYEE, A., AND ZAHARIA, M. 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* (New York, NY, USA, 2021), SC '21, Association for Computing Machinery.

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

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

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

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

[54] NVIDIA. Nemotron-4 340b technical report, 2024.

[55] PATI, S., AGA, S., ISLAM, M., JAYASENA, N., AND SINCLAIR, M. D. 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* (New York, NY, USA,

2024), ASPLOS '24, Association for Computing Machinery, p. 1146–1164.

[56] PENG, Y., ZHU, Y., CHEN, Y., BAO, Y., YI, B., LAN, C., WU, C., AND GUO, C. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, Association for Computing Machinery, p. 16–29.

[57] POPE, R., DOUGLAS, S., CHOWDHERY, A., DEVLIN, J., BRADBURY, J., HEEK, J., XIAO, K., AGRAWAL, S., AND DEAN, J. Efficiently scaling transformer inference. 606–624.

[58] QIAN, K., XI, Y., CAO, J., GAO, J., XU, Y., GUAN, Y., FU, B., SHI, X., ZHU, F., MIAO, R., WANG, C., WANG, P., ZHANG, P., ZENG, X., RUAN, E., YAO, Z., ZHAI, E., AND CAI, D. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference* (New York, NY, USA, 2024), ACM SIGCOMM '24, Association for Computing Machinery, p. 691–706.

[59] RABENSEIFNER, R. Optimization of collective reduction operations. In *Computational Science - ICCS 2004* (Berlin, Heidelberg, 2004), M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds., Springer Berlin Heidelberg, pp. 1–9.

[60] RAJASEKARAN, S., GHOBADI, M., AND AKELLA, A. CASSINI: Network-Aware job scheduling in machine learning clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)* (Santa Clara, CA, Apr. 2024), USENIX Association, pp. 1403–1420.

[61] RAJBHANDARI, S., RASLEY, J., RUWASE, O., AND HE, Y. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), SC '20, IEEE Press.

[62] ROCm Collective Communication Library (RCCL). ht tps://github.com/ROCm/rccl.

[63] SAPIO, A., CANINI, M., HO, C.-Y., NELSON, J., KALNIS, P., KIM, C., KRISHNAMURTHY, A., MOSHREF, M., PORTS, D., AND RICHTARIK, P. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association, pp. 785–808.

[64] SCHRIJVER, A. Combinatorial optimization : polyhedra and efficiency, 2003.

[65] SERGEEV, A., AND BALSO, M. D. Horovod: fast and easy distributed deep learning in tensorflow, 2018.

[66] SHAH, A., CHIDAMBARAM, V., COWAN, M., MALEKI, S., MUSUVATHI, M., MYTKOWICZ, T., NELSON, J., SAARIKIVI, O., AND SINGH, R. TACCL: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (Boston, MA, Apr. 2023), USENIX Association, pp. 593–612.

[67] SHAH, A., JANGDA, A., LI, B., ROCHA, C., HWANG, C., JOSE, J., MUSUVATHI, M., SAARIKIVI, O., CHENG, P., ZHOU, Q., DATHATHRI, R., MALEKI, S., AND YANG, Z. Msccl++: Rethinking gpu communication abstractions for cutting-edge ai applications, 2025.

[68] SHOEYBI, M., PATWARY, M., PURI, R., LEGRESLEY, P., CASPER, J., AND CATANZARO, B. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[69] TARJAN, R. E. A good algorithm for edge-disjoint branching. *Information Processing Letters 3*, 2 (1974), 51–53.

[70] THE MOSAIC RESEARCH TEAM. DBRX 132B. https://www.databricks.com/blog/introducing-dbrx-new-state-art-open-llm.

[71] WANG, G., VENKATARAMAN, S., PHANISHAYEE, A., DEVANUR, N., THELIN, J., AND STOICA, I. Blink: Fast and generic collectives for distributed ml. In *Proceedings of Machine Learning and Systems* (2020), I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, pp. 172–186.

[72] WANG, G., ZHANG, C., SHEN, Z., LI, A., AND RUWASE, O. Domino: Eliminating communication in llm training via generic tensor slicing and overlapping, 2024.

[73] WANG, H., TIAN, H., CHEN, J., WAN, X., XIA, J., ZENG, G., BAI, W., JIANG, J., WANG, Y., AND CHEN, K. Towards Domain-Specific network transport for distributed DNN training. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)* (Santa Clara, CA, Apr. 2024), USENIX Association, pp. 1421–1443.

[74] WANG, R., DONG, D., LEI, F., MA, J., WU, K., AND LU, K. Roar: A router microarchitecture for in-network allreduce. In *Proceedings of the 37th ACM International Conference on Supercomputing* (New York, NY, USA, 2023), ICS '23, Association for Computing Machinery, p. 423–436.

[75] WANG, S., WEI, J., SABNE, A., DAVIS, A., ILBEYI, B., HECHTMAN, B., CHEN, D., MURTHY, K. S., MAGGIONI, M., ZHANG, Q., KUMAR, S., GUO, T., XU, Y., AND ZHOU, Z. 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* (New York, NY, USA, 2022), ASPLOS 2023, Association for Computing Machinery, p. 93–106.

[76] WANG, W., GHOBADI, M., SHAKERI, K., ZHANG, Y., AND HASANI, N. How to build low-cost networks for large language models (without sacrificing performance)?, 2023.

[77] WANG, W., GHOBADI, M., SHAKERI, K., ZHANG, Y., AND HASANI, N. Rail-only: A low-cost high-performance network for training llms with trillion parameters. In *2024 IEEE Symposium on High-Performance Interconnects (HOTI)* (Los Alamitos, CA, USA, aug 2024), IEEE Computer Society, pp. 1–10.

[78] WANG, W., KHAZRAEE, M., ZHONG, Z., GHOBADI, M., JIA, Z., MUDIGERE, D., ZHANG, Y., AND KEWITSCH, A. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (Boston, MA, Apr. 2023), USENIX Association, pp. 739–767.

[79] WOLF, T., DEBUT, L., SANH, V., CHAUMOND, J., DELANGUE, C., MOI, A., CISTAC, P., RAULT, T., LOUF, R., FUNTOWICZ, M., DAVISON, J., SHLEIFER, S., VON PLATEN, P., MA, C., JERNITE, Y., PLU, J., XU, C., SCAO, T. L., GUGGER, S., DRAME, M., LHOEST, Q., AND RUSH, A. M. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (Online, Oct. 2020), Association for Computational Linguistics, pp. 38–45.

[80] WON, W., ELAVAZHAGAN, M., SRINIVASAN, S., GUPTA, S., AND KRISHNA, T. TACOS: Topology-Aware Collective Algorithm Synthesizer for Distributed Machine Learning. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Los Alamitos, CA, USA, Nov. 2024), IEEE Computer Society, pp. 856–870.

[81] WU, Y., XU, Y., CHEN, J., WANG, Z., ZHANG, Y., LENTZ, M., AND ZHUO, D. Mccs: A service-based approach to collective communication for multi-tenant cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference* (New York, NY, USA, 2024), ACM SIGCOMM '24, Association for Computing Machinery, p. 679–690.

[82] ZHAO, L., PAL, S., CHUGH, T., WANG, W., FANTL, J., BASU, P., KHOURY, J., AND KRISHNAMURTHY, A. Efficient Direct-Connect topologies for collective communications. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)* (Philadelphia, PA, Apr. 2025), USENIX Association, pp. 705–737.

[83] ZHAO, Y., GU, A., VARMA, R., LUO, L., HUANG, C.-C., XU, M., WRIGHT, L., SHOJANAZERI, H., OTT, M., SHLEIFER, S., DESMAISON, A., BALIOGLU, C., DAMANIA, P., NGUYEN, B., CHAUHAN, G., HAO, Y., MATHEWS, A., AND LI, S. PyTorch fsdp: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow. 16*, 12 (aug 2023), 3848–3860.

[84] ZHAO, Y., LIU, X., AND JIN, X. How useful is communication scheduling for distributed training? In *2024 International Scientific and Technical Conference Modern Computer Network Technologies (MoNeTeC)* (2024), pp. 1–13.

[85] ZHENG, L., LI, Z., ZHANG, H., ZHUANG, Y., CHEN, Z., HUANG, Y., WANG, Y., XU, Y., ZHUO, D., XING, E. P., GONZALEZ, J. E., AND STOICA, I. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 559–578.

# Appendix

In this appendix, we provide detailed mathematical analysis and proofs to supplement the main text. To summarize,

- §A provides a summary of notations used in the main text and appendix.
- §B discusses related works beyond schedule generation.
- §C describes the implementation and parallelization of schedule generation algorithm.
- §D shows a dilemma for step schedules to reach optimality.
- §E elaborates on the mathematical details of the algorithm.
- §F proves that every part of our algorithm runs in polynomial time.
- §G describes a linear program to construct optimal allreduce schedule.
- §H provides proofs of all theorems in this paper.

## A 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 §E.1.
- $G(\{Ub_e\})$: a graph obtained by multiplying each link bandwidth $b_e$ of $G$ by $U$. Defined in §E.1.
- $G^{ef}$: a graph obtained by splitting off edge $e$ and $f$ of $G$. Defined in §E.2.
- $G^*=(V_c,E^*)$: the graph after removing all switch nodes from $G$ using edge splitting technique. Defined in §E.2.
- $\widehat{D}_{(u,w),v}, \widehat{D}_{(w,t),v}$: the auxiliary networks for edge splitting (computing $\gamma$). Defined in §E.2.
- $\overline{D}$: the auxiliary network for spanning tree construction (computing $\mu$). Defined in §E.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.
- $\gamma$: the maximum capacity we can safely replace (or split off) $(u,w),(w,t)$ with $(u,t)$. Defined in Theorem 6.
- $\mu$: the maximum capacity we can add an edge into a tree. Defined in (3) of §E.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 $T_{u,i}$.
- $m(R_{u,i}), g(x,y)$: notations for spanning tree construction. Defined in Theorem 9.

**Table 2: 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.

| Method | Optimality | Generality | Scalability |
|---|---|---|---|
| SCCL [10] | Fixed-Chunk | Non-Switch | NP-hard |
| TACCL [66] | Heuristic | Yes | NP-hard |
| TE-CCL [41] | Heuristic | Yes | NP-hard |
| TACOS [80] | Greedy | Yes | Yes |
| BFB [82] | Conditional | Non-Switch | Yes |
| Blink [71] | Single-Root | Non-Switch | Yes |
| MultiTree [30] | Greedy | Yes | Yes |
| TTO [36] | Mesh-Only | Non-Switch | Yes |
| SyCCL [11] | Heuristic | Yes | NP-hard |
| ForestColl | Yes | Yes | Yes |

## B Other Related Work

**Other Optimizations of Collective Communications:** Beyond schedule generation, other works optimize collective comms in ways orthogonal to ForestColl. TopoOpt [78], Rail-only [77], and BFB [82] optimize the underlying network topology, where ForestColl's optimality for any topology can be beneficial. C-Cube [17] explores efficient comm on logical trees, such as overlapping reduction and broadcast, but does not mention tree construction. BlueConnect [16] proposes a collective algorithm for single hierarchical switching fabrics but is otherwise inapplicable. Recent works [15, 20, 35, 40, 63, 74] 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 [73], packet scheduling [56, 84], and multi-tenancy [60, 81].

**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 [49, 68] showcases how to combine data, tensor, and pipeline parallelisms to speed up LLM training. FlexFlow [33] and Alpa [85] propose automated search for hybrid parallelism strategies, with Alpa specifically aiming to minimize comm cost. nnScaler [39] 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 [37], FSDP [83], and Domino [72], exploit overlap opportunities within a single parallelism. More complicated approaches, including CoCoNet [32], Syndicate [43], and Centauri [14], optimize the scheduling of comp and comm operations to achieve overlap in hybrid parallelism. Recent works like CoCoNet [32], [75], T3 [55], and Flux [13] 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 [33, 39, 85]. 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 (§6.2.1)—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 [13, 55, 72, 75]. Besides, resource contention on GPUs (e.g., comp units, memory bandwidth) between comp and comm operations underscores that overlap is not cost-free [1, 75]. ForestColl helps reduce the non-overlapped comm cost and enhance comm efficiency in overlap (§6.4).

## C 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 [27] provided by JGraphT library [46]. Table 3 shows the breakdown of schedule generation time for 1024-GPU topologies in §6.5.

**Table 3: Breakdown of schedule generation time for 1024-GPU topologies in §6.5.** Runtimes were measured on a 128-core 2.2GHz CPU.

| 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 |

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 $\gamma$ in Algorithm 2, which also requires independent 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 $\mu$ 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 $\mu$, 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 $\mu$ for $\Omega(kN^2)$ times. To address this, we take inspiration from branch prediction. Specifically, thread $j$ speculatively computes $\mu$ 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 $\mu$ 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.

## D 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 15a. 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 15b. 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

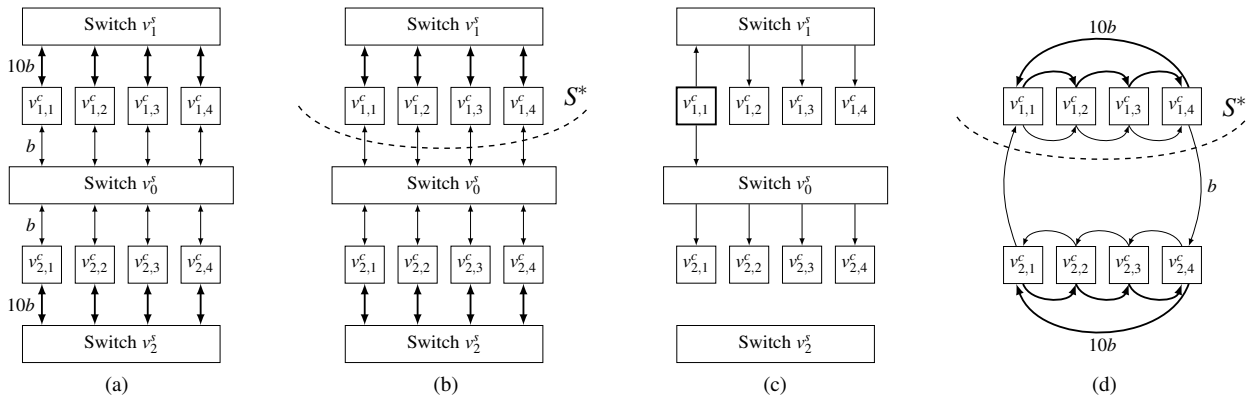



**Figure 15: An 8-compute-node switch topology in 2-box setting.** The thick links have 10x the bandwidth of the thin ones. Figure (a) shows the original switch topology. Figure (b) shows the bottleneck cut in this topology. Figure (c) shows a spanning tree rooted at $v_{1,1}^c$ with switch-node broadcast. Figure (d) shows a suboptimal way of transforming the switch topology into a direct-connect logical topology (resulting in 4x worse optimal performance).

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.

# E 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:

- §E.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 ($\star$).
- §E.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.
- §E.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 [7, 8, 21, 64, 69]. 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 [7, 22, 31]. 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 [66] and TACOS [80] 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 15a into rings, resulting in Figure 15d. 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 16b, which has the same optimality as Figure 15a.

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, $F(x,y;D) \le c(A,\bar{A};D)$ if $x \in A, y \in \bar{A}$, and there exists an $x$-$y$ cut $(A^*,\overline{A^*})$ that $F(x,y;D) = c(A^*,\overline{A^*};D)$.

### E.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 1.** $\min_{v\in V_c} F(s,v;\vec{G}_x) \ge |V_c|x$ *if and only if* $1/x \ge \max_{S\subset V, S\not\supseteq V_c} |S\cap V_c|/B^+_G(S)$.

The implication of Theorem 1 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

$$\frac{N-1}{\min_{v\in V_c} B^-_G(v)} \le \max_{S\subset V, S\not\supseteq V_c} \frac{|S\cap V_c|}{B^+_G(S)} \le N-1.$$

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 $|S\cap V_c| \le N-1$ and $B^+_G(S) \ge 1$. Starting with the initial range, one can then continuously test if $\min_{v\in V_c} F(s,v;\vec{G}_x) \ge |V_c|x$ 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 $|S^*\cap V_c| \le N-1$ and $|S^*\cap V_c|/B^+_G(S^*) \ge (N-1)/\min_{v\in V_c} B^-_G(v)$, so $B^+_G(S^*) \le \min_{v\in V_c} B^-_G(v)$. 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 $b,d \le X$ for some $X \in \mathbb{Z}_+$, then $|a/b - c/d| \ge 1/X^2$. The proposition implies that if $1/x^* = a/b$ for some $b \le \min_{v\in V_c} B^-_G(v)$, then any $c/d \ne 1/x^*$ with $d \le \min_{v\in V_c} B^-_G(v)$ satisfies $|c/d - 1/x^*| \ge 1/\min_{v\in V_c} B^-_G(v)^2$. 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 15a'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. [7]:

**Theorem 2** (Bang-Jensen et al. [7])**.** *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 2 leads to the following:

**Theorem 3.** *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

$$\begin{aligned}\min_{v\in V_c} F(s,v;\vec{G}_k(\{Ub_e\})) &= U \cdot \min_{v\in V_c} F(s,v;\vec{G}_{x^*}) \\ &\ge U\cdot|V_c|x^* \\ &= |V_c|k.\end{aligned}$$

By Theorem 3, 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 $Ub_e$

**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 ∈ V_s do
        foreach egress edge f = (w,t) ∈ E do
            foreach ingress edge e = (u,w) ∈ E do
                Compute γ as in (1).
                if γ > 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) ∉ E.
                    routing[(u,t)][w] ← routing[(u,t)][w] + γ
                    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
```

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

$$T_{\text{comm}} \le \max_{e\in E} \frac{M}{Nk}\cdot\frac{Ub_e}{b_e} = \frac{M}{N}\cdot\frac{U}{k} = \frac{M}{N}\cdot\frac{1}{x^*} = \frac{M}{N}\max_{S\subset V, S\supsetneq V_c}\frac{|S\cap V_c|}{B^+_G(S)}$$

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 15c. 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.

### E.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 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 $\min_{v\in V_c} F(s,v;\vec{G}_k(\{Ub_e\})) \ge |V_c|k$ is maintained in the process. Thus, by Theorem 3, the existence of spanning trees with the same optimal performance is also preserved.

We start with the following theorem from Bang-Jensen et al. [7]. The theorem was originally proven by Frank [22] and Jackson [31].

**Theorem 4** (Bang-Jensen et al. [7])**.** *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 $\lambda(x,y;D^{ef}) \ne \lambda(x,y;D)$ as long as $\min_{v\in V_c} F(s,v;\vec{D}_k^{ef}) = \min_{v\in V_c}\lambda(s,v;\vec{D}_k^{ef}) \ge |V_c|k$ holds after splitting. Theorem 4 is used to derive the following theorem:

**Theorem 5.** *Given an integer-capacity Eulerian digraph $D = (V_s \cup V_c, E)$ and $k \in \mathbb{N}$ with $\min_{v\in V_c} F(s,v;\vec{D}_k) \ge |V_c|k$, for every edge $f = (w,t)$ $(w \in V_s)$ there is an edge $e = (u,w)$ such that $\min_{v\in V_c} F(s,v;\vec{D}_k^{ef}) \ge |V_c|k$.*

Note that here, $f$ and $e$ each represent one of the multi-edges (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 $\min_{v\in V_c} F(s,v;\vec{G}_k^{ef}(\{Ub_e\})) \ge |V_c|k$. 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 3, 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 $\infty$ capacity. Similarly, we construct a flow network $\widehat{D}_{(w,t),v}$ that connects $(w,s),(u,t),(v,t)$ with $\infty$ capacity.

**Theorem 6.** *Given an integer-capacity Eulerian digraph $D = (V_s \cup V_c, E)$ and $k \in \mathbb{N}$ with $\min_{v\in V_c} F(s,v;\vec{D}_k) \ge |V_c|k$, the maximum capacity that $e = (u,w), f = (w,t)$ can be split off with the resulting graph $D^{ef}$ satisfying $\min_{v\in V_c} F(s,v;\vec{D}_k^{ef}) \ge |V_c|k$ is*

$$\gamma = \min\Big\{ c(u,w;D)\ ,\ c(w,t;D)\ ,\ \min_{v\in V_c} F(u,w;\widehat{D}_{(u,w),v}) - |V_c|k\ ,\ \min_{v\in V_c} F(w,t;\widehat{D}_{(w,t),v}) - |V_c|k \Big\}. \tag{1}$$

Based on Theorem 6, 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

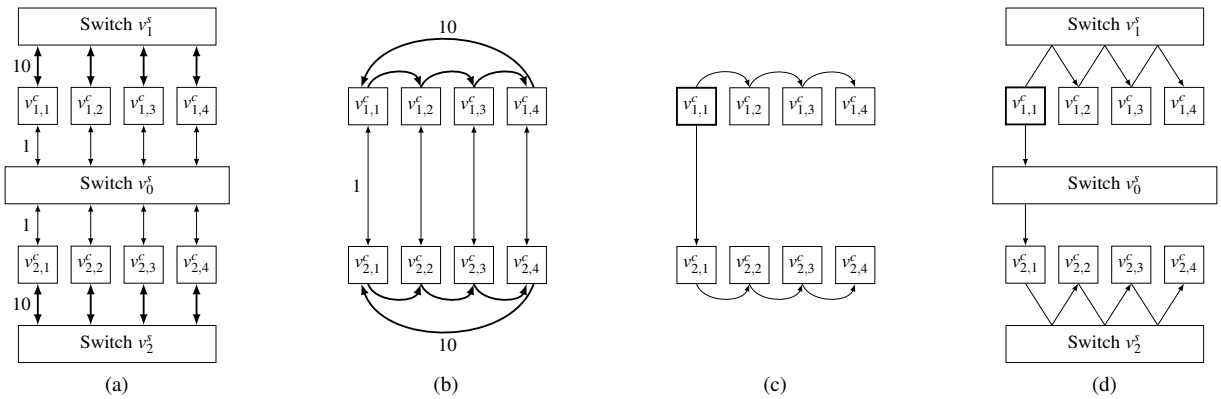



**Figure 16: Different stages of the topology in schedule construction.** Figure (a) shows the topology of $G(\{Ub_e\})$. Note that the link capacities no longer have $b$ as a multiplier. Figure (b) shows the topology $G^*$ after edge splitting removes all switch nodes. Figure (c) shows a spanning tree constructed in $G^*$. Figure (d) shows the routings in $G$ corresponding to the spanning tree.

**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 $R_{u,1} = \{u\}, \mathcal{E}(R_{u,1}) = \emptyset, m(R_{u,1}) = k, n_u = 1$ for all $u \in V_c$.
  Initialize $g(e) = c(e; D^*)$ for all $e \in E^*$.
  **while** there exists $R_{u,i} \neq V_c$ **do**
    **while** $R_{u,i} \neq V_c$ **do**
      Pick an edge $(x,y)$ in $D^*$ that $x \in R_{u,i}, y \notin R_{u,i}$.
      Compute $\mu$ as in (4).
      **if** $\mu = 0$ **then continue**
      **if** $\mu < m(R_{u,i})$ **then**
        $n_u \leftarrow n_u + 1$
        Create a new copy $R_{u,n_u} = R_{u,i}, \mathcal{E}(R_{u,n_u}) = \mathcal{E}(R_{u,i}), m(R_{u,n_u}) = m(R_{u,i}) - \mu$.
        $m(R_{u,i}) \leftarrow \mu$
      $\mathcal{E}(R_{u,i}) \leftarrow \mathcal{E}(R_{u,i}) + (x,y)$
      $R_{u,i} \leftarrow R_{u,i} + y$
      $g(x,y) \leftarrow g(x,y) - \mu$.
      Remove $(x,y)$ if $g(x,y)$ reaches 0.

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 16 gives an example of edge splitting. In Figure 16a, 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 16b has compute nodes only, and the optimal communication time is still $(M/N)(4/4b)$ if bandwidth multiplier $b$ is added.

### E.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 [69]. The theorem was originally proven by Edmonds [21].

**Theorem 7** (Tarjan [69])**.** *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 $S \neq V$,*

$$c(S, \bar{S}; D) \ge |\{i \mid R_i \subseteq S\}|. \tag{2}$$

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 8.** *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 (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 (2). Tarjan [69] 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 (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 [8] have proposed a strongly polynomial time algorithm based on Schrijver [64]. The runtime of the algorithm does not depend on $k$ at all. In particular, the following theorem has been shown:

**Theorem 9** (Bérczi & Frank [8]). *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 $\mu: 0<\mu \le \min\{g(x,y), m(R_1)\}$ copies of the $m(R_1)$ out-trees without violating (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 $\mu$, 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 [8], $\mu$ is defined as follows:

$$\mu = \min\Big\{ g(x,y)\ ,\ m(R_1)\ ,\ \min\{c(S,\bar{S};D)-p(S;D) : x\in S, y\in\bar{S}, R_1 \not\subseteq S\}\Big\} \quad (3)$$

where $p(S;D)=\sum\{m(R_i) \mid R_i \subseteq S\}$. Neither Bérczi & Frank [8] nor Schrijver [64] explicitly mentioned how to compute $\mu$ 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 $\infty$ capacity. We then show the following result:

**Theorem 10.** *For any edge $(x,y)$ in $D$ with $x \in R_1, y \notin R_1$,*

$$\mu = \min\Big\{ g(x,y)\ ,\ m(R_1)\ ,\ F(x,y;\overline{D}) - \textstyle\sum_{i\neq 1} m(R_i)\Big\}. \quad (4)$$

Thus, $\mu$ 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 $m(R_{u,i})$.

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

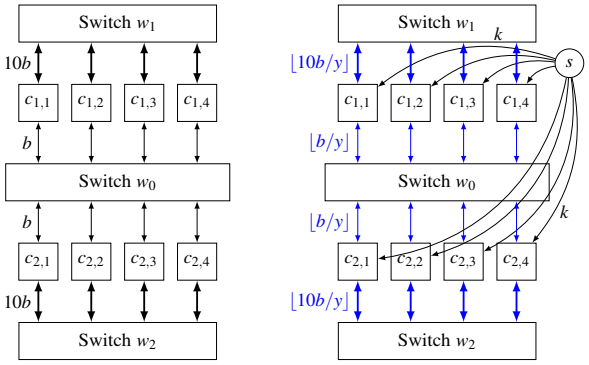



**Figure 17: The auxiliary network for fixed-$k$ binary search.** (a) shows the original topology. (b) shows the auxiliary network ForestColl uses to binary search for the optimal $y$ (the bandwidth utilized by each tree) given a fixed $k$ (the number of trees rooted at each compute node).

`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 16's example, we construct a spanning tree like 16c for each compute node. By reversing the edge splitting with `routing`, the spanning tree becomes the schedule in 16d. Note that the corresponding schedule of a spanning tree in $G^*$ is not necessarily a tree in $G$. For example, the schedule in 16d 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 [9].

### E.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 11.** *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 $G(\{\lfloor Ub_e \rfloor\}_{e\in E})$.*

To test the existence of edge disjoint $\{T_{u,i}\}_{u\in V_c, i\in[k]}$ in $G(\{\lfloor Ub_e \rfloor\}_{e\in E})$, by Theorem 3, we can simply test whether

**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 $y^*$

$\min_{v\in V_c} F(s,v;\vec{G}_k(\{\lfloor Ub_e \rfloor\})) \geq |V_c|k$ holds. The following theorem provides a method for binary search to find the lowest communication time for the given $k$.

**Theorem 12.** *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 $G(\{\lfloor Ub_e \rfloor\}_{e\in E})$ 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:

$$\frac{(N-1)k}{\min_{v\in V_c} B_G^-(v)} \leq U^* \leq (N-1)k.$$

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 E.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 17 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 13.** *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,*

$$\frac{M}{Nk}\cdot U^* \leq \frac{M}{N} \max_{S\subset V, S\not\supseteq V_c} \frac{|S\cap V_c|}{B_G^+(S)} + \frac{M}{Nk}\cdot \frac{1}{\min_{e\in E} b_e}.$$

# F 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 §C 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 §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 [27] 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 $\gamma$ in Theorem 6 takes $O(N|V|^4)$, and $\gamma$ 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, $\mu$ only needs one maxflow to be computed. The runtime is thus $O(N^4)$. Bérczi & Frank [8] proved that $\mu$ 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 $O(N|V|^2|E|(\log N + \log k + \log\max_{e\in E} b_e))$.

## G 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:

(i) Each node can be the root of a variable number of spanning trees instead of equal number in allgather.

(ii) 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 §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

$$T_{\text{comm}} = M / \sum_{v\in V_c} x_v.$$

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 $G$s, on which we can apply the spanning tree construction to derive in-trees and out-trees, respectively. The linear program formulation is as follows:

$$\begin{aligned}
\max \quad & \textstyle\sum_{v\in V_c} x_v & \\
\text{s.t.} \quad & F(s,t;\vec{G}) \geq \textstyle\sum_{v\in V_c} x_v, && \forall t \in V_c \\
& \quad \text{w.r.t. } 0 \leq f^{\text{s,t}}_{(s,v)} \leq x_v \text{ and } 0 \leq f^{\text{s,t}}_{(u,v)} \leq c^{\text{BC}}_{(u,v)} && \\
& F(t,s;\vec{G}) \geq \textstyle\sum_{v\in V_c} x_v, && \forall t \in V_c \\
& \quad \text{w.r.t. } 0 \leq f^{t,s}_{(v,s)} \leq x_v \text{ and } 0 \leq f^{t,s}_{(u,v)} \leq c^{\text{RE}}_{(u,v)} && \\
& c^{\text{RE}}_{(u,v)} + c^{\text{BC}}_{(u,v)} \leq b_{(u,v)}, && \forall (u,v) \in E_G \\
& c^{\text{RE}}_{(u,v)}, c^{\text{BC}}_{(u,v)} \geq 0, && \forall (u,v) \in E_G \\
& x_v \geq 0. && \forall v \in V_c
\end{aligned} \tag{5}$$

Note that for reduce in-trees, we constrain the maxflow from $V_c$ to $s$ rather than from $s$ to $V_c$. Accordingly, the $x_v$s are also capacities from $v \in V_c$ to $s$. The solution to LP (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 (5). We add multi-commodity flow constraints into the LP to ensure $b'_{(\alpha,\beta)}$ commodity flow from $\alpha$ to $\beta$ 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 (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.

## H Proofs

**Theorem 1.** $\min_{v\in V_c} F(s,v;\vec{G}_x) \geq |V_c|x$ *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

$$\begin{aligned}
c(A,\bar{A};\vec{G}_x) &= c(S',\bar{A};\vec{G}_x) + \sum_{u\in\bar{A}\cap V_c} c(s,u;\vec{G}_x) \\
&= B^+_G(S') + |V_c - S'|x \\
&< |S'\cap V_c|x + |V_c - S'|x \\
&= |V_c|x.
\end{aligned} \tag{6}$$

By min-cut theorem, $\min_{v\in V_c} F(s,v;\vec{G}_x) \le F(s,v';\vec{G}_x) \le c(A,\bar{A};\vec{G}_x) < |V_c|x$.

$\Leftarrow$: Suppose $1/x \ge \max_{S\subset V, S\not\supseteq V_c} |S\cap V_c|/B^+_G(S)$. 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 $1/x \ge |S'\cap V_c|/B^+_G(S')$. Thus, following (6),

$$\begin{aligned} c(A,\bar{A};\vec{G}_x) &= B^+_G(S') + |V_c - S'|x \\ &\ge |S'\cap V_c|x + |V_c - S'|x \\ &= |V_c|x. \end{aligned}$$

Because cut $(A,\bar{A})$ is arbitrary, we have $F(s,v';\vec{G}_x) \ge |V_c|x$. Because $v'$ is also arbitrary, $\min_{v\in V_c} F(s,v;\vec{G}_x) \ge |V_c|x$. □

Given two unequal fractional numbers $a/b$ and $c/d$ with $a,b,c,d\in\mathbb{Z}_+$, if denominators $b,d\le X$ for some $X\in\mathbb{Z}_+$, then $|a/b - c/d| \ge 1/X^2$.

*Proof.* Because $a/b \ne c/d$, we have $ad - bc \ne 0$. Thus,

$$\left|\frac{a}{b} - \frac{c}{d}\right| = \left|\frac{ad-bc}{bd}\right| \ge \frac{1}{bd} \ge \frac{1}{X^2}.$$

□

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 $\alpha,\beta$ 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 $\alpha$ is $p$, and the largest such $\beta$ is $\gcd(q,\{b_e\}_{e\in E})$. The proposition immediately follows. □

**Theorem 2** (Bang-Jensen et al. [7])**.** *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 $\mathcal{F}$.*

**Theorem 3.** *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 $\min_{v\in V_c} F(s,v;\vec{D}_k)\ge |V_c|k$.*

*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 $\min_{v\in V_c} F(s,v;\vec{D}_k)\ge |V_c|k$.

$\Leftarrow$: Suppose $\min_{v\in V_c} F(s,v;\vec{D}_k)\ge |V_c|k$. Because $D$ is Eulerian, we have $T' = \emptyset$ and the assumption in Theorem 2 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 4** (Bang-Jensen et al. [7])**.** *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 5.** *Given an integer-capacity Eulerian digraph $D=(V_s\cup V_c,E)$ and $k\in\mathbb{N}$ with $\min_{v\in V_c} F(s,v;\vec{D}_k)\ge |V_c|k$, for every edge $f=(w,t)$ $(w\in V_s)$ there is an edge $e=(u,w)$ such that $\min_{v\in V_c} F(s,v;\vec{D}^{ef}_k)\ge |V_c|k$.*

*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 4, given $f=(w,t)$, there exists an edge $e=(u,w)$ such that $\lambda(s,v;\vec{D}'^{ef}_k) = \lambda(s,v;\vec{D}'_k)$ 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 $v\in V_c$,

$$\begin{aligned} F(s,v;\vec{D}^{ef}_k) &= \lambda(s,v;\vec{D}^{ef}_k) = \lambda(s,v;\vec{D}'^{ef}_k) \\ &= \lambda(s,v;\vec{D}'_k) = \lambda(s,v;\vec{D}_k) = F(s,v;\vec{D}_k). \end{aligned}$$

The theorem trivially follows. □

**Theorem 6.** *Given an integer-capacity Eulerian digraph $D=(V_s\cup V_c,E)$ and $k\in\mathbb{N}$ with $\min_{v\in V_c} F(s,v;\vec{D}_k)\ge |V_c|k$, the maximum capacity that $e=(u,w), f=(w,t)$ can be split off with the resulting graph $D^{ef}$ satisfying $\min_{v\in V_c} F(s,v;\vec{D}^{ef}_k)\ge |V_c|k$ is*

$$\gamma = \min\left\{ c(u,w;D)\ ,\ c(w,t;D)\ ,\ \min_{v\in V_c} F(u,w;\widehat{D}_{(u,w),v}) - |V_c|k\ ,\ \min_{v\in V_c} F(w,t;\widehat{D}_{(w,t),v}) - |V_c|k \right\}. \quad (1)$$

*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 $\gamma$ 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,

$$\begin{aligned} F(u,w;\widehat{D}_{(u,w),v'}) &\leq c(A,\bar{A};\widehat{D}_{(u,w),v'}) \\ &= c(A,\bar{A};\vec{D}_k) = c(A,\bar{A};\vec{D}_k^{ef}) + \gamma < |V_c|k + \gamma, \end{aligned}$$

contradicting $\gamma \leq \min_{v \in V_c} F(u,w;\widehat{D}_{(u,w),v}) - |V_c|k$. For $s,w \in A \wedge v', u,t \in \bar{A}$, one can similarly show a contradiction by looking at $F(w,t;\widehat{D}_{(w,t),v'})$.

$\leq$: 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 $\gamma'$ 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 $\infty$ 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,

$$\begin{aligned} F(s,v';\vec{D}_k^{ef}) \leq c(A,\bar{A};\vec{D}_k^{ef}) &= c(A,\bar{A};\vec{D}_k) - \gamma' \\ &= c(A,\bar{A};\widehat{D}_{(u,w),v'}) - \gamma' < |V_c|k. \end{aligned}$$

One can show similar result for $\gamma' > F(w,t;\widehat{D}_{(w,t),v'}) - |V_c|k$. □

**Theorem 7** (Tarjan [69])**.** *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 $S \neq V$,*

$$c(S,\bar{S};D) \geq |\{i \mid R_i \subseteq S\}|. \tag{2}$$

**Theorem 8.** *Given an integer-capacity digraph $D=(V_c,E)$ and $k \in \mathbb{N}$, $c(S,\bar{S};D) \geq |S|k$ for all $S \subset V_c, S \neq V_c$ if and only if $\min_{v \in V_c} F(s,v;\vec{D}_k) \geq |V_c|k$.*

*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

$$\begin{aligned} c(S,\bar{S};D) = c(A,\bar{A};\vec{D}_k) - \sum_{u \in \bar{A}} c(s,u;\vec{D}_k) \\ < |V_c|k - |V_c - S|k = |S|k. \end{aligned}$$

$\Leftarrow$: 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

$$\begin{aligned} F(s,v';\vec{D}_k) \leq c(A,\bar{A};\vec{D}_k) &= c(S,\bar{S};D) + \sum_{u \in \bar{A}} c(s,u;\vec{D}_k) \\ &< |S|k + |V_c - S|k = |V_c|k. \end{aligned}$$

□

**Theorem 9** (Bérczi & Frank [8])**.** *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 10.** *For any edge $(x,y)$ in $D$ with $x \in R_1, y \notin R_1$,*

$$\mu = \min\left\{ g(x,y)\ ,\ m(R_1)\ ,\ F(x,y;\overline{D}) - \textstyle\sum_{i \neq 1} m(R_i) \right\}. \tag{4}$$

*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 (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

$$\begin{aligned} c(S,\bar{S};D) - p(S;D) &= c(S,\bar{S};D) - \textstyle\sum\{m(R_i) \mid R_i \subseteq S\} \\ &= c(S,\bar{S};D) + \textstyle\sum\{m(R_i) \mid i \neq 1, R_i \not\subseteq S\} \\ &\quad - \textstyle\sum_{i \neq 1} m(R_i) \\ &= c(A,\bar{A};\overline{D}) - \textstyle\sum_{i \neq 1} m(R_i) \\ &\geq F(x,y;\overline{D}) - \textstyle\sum_{i \neq 1} m(R_i). \end{aligned}$$

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 $\infty$ edge $(s_i,v)$ crosses $(A',\overline{A'})$. Thus, we have

$$\begin{aligned} &F(x,y;\overline{D}) - \textstyle\sum_{i \neq 1} m(R_i) \\ =&c(A',\overline{A'};\overline{D}) - \textstyle\sum_{i \neq 1} m(R_i) \\ =&c(S',\overline{S'};D) + \textstyle\sum\{m(R_i) \mid i \neq 1, R_i \not\subseteq S'\} - \textstyle\sum_{i \neq 1} m(R_i) \\ =&c(S',\overline{S'};D) - \textstyle\sum\{m(R_i) \mid i \neq 1, R_i \subseteq S'\}. \end{aligned} \tag{7}$$

Now, we consider two cases:

(a) Suppose $R_1 \not\subseteq S'$. Then, $c(S',\overline{S'};D) - p(S';D) \geq L$. By (7), we have

$$\begin{aligned} L &\geq F(x,y;\overline{D}) - \textstyle\sum_{i \neq 1} m(R_i) \\ &= c(S',\overline{S'};D) - \textstyle\sum\{m(R_i) \mid i \neq 1, R_i \subseteq S'\} \\ &= c(S',\overline{S'};D) - p(S';D) \end{aligned}$$

Thus, $L = c(S', \overline{S'}; D) - p(S'; D) = F(x, y; \overline{D}) - \sum_{i \neq 1} m(R_i)$ and (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,

$$\begin{aligned} L &\geq F(x, y; \overline{D}) - \textstyle\sum_{i \neq 1} m(R_i) \\ &= c(S', \overline{S'}; D) - \textstyle\sum\{m(R_i) \mid i \neq 1, R_i \subseteq S'\} \\ &\geq m(R_1). \end{aligned}$$

Thus, $\mu = \min\{g(x, y), m(R_1)\}$ and (4) also holds.

□

**Theorem 11.** *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

$$\begin{aligned} T_{\text{comm}} = \frac{M}{Nk} \cdot \max_{e \in E} \frac{1}{b_e} \sum_{T \in \{T_{u,i}\}} \mathbb{I}[e \in T] \\ \leq \frac{M}{Nk} \cdot \max_{e \in E} \frac{\lfloor U b_e \rfloor}{b_e} \leq \frac{M}{Nk} \cdot U. \end{aligned}$$

$\Rightarrow$: Suppose $\{T_{u,i}\}_{u \in V_c, i \in [k]}$ achieves $\frac{M}{Nk} \cdot U$ communication time, then

$$\begin{aligned} \max_{e \in E} \frac{1}{b_e} \sum_{T \in \{T_{u,i}\}} \mathbb{I}[e \in T] \leq U \\ \implies \sum_{T \in \{T_{u,i}\}} \mathbb{I}[e \in T] \leq U b_e \quad \text{for all } e \in E. \end{aligned}$$

Since $\sum_{T \in \{T_{u,i}\}} \mathbb{I}[e \in T]$ must be an integer, the edge-disjointness trivially follows. □

**Theorem 12.** *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 11, it is edge-disjoint in $G(\{\lfloor U b_e \rfloor\}_{e \in E})$ for all $U \geq U^*$. □

**Theorem 13.** *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

$$\begin{aligned} c(u, v; G(\lfloor U b_e \rfloor)) &= \left\lfloor b_{(u,v)} \cdot \max_{e \in E} \frac{\lceil k b_e / x^* \rceil}{b_e} \right\rfloor \\ &\geq \left\lfloor b_{(u,v)} \cdot \frac{\lceil k b_{(u,v)} / x^* \rceil}{b_{(u,v)}} \right\rfloor = \lceil k b_{(u,v)} / x^* \rceil. \end{aligned}$$

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 12.

$$\begin{aligned} \frac{U^*}{k} \Big/ \frac{1}{x^*} \leq \frac{U}{k} \Big/ \frac{1}{x^*} = \frac{\max_{e \in E} \lceil k b_e / x^* \rceil / b_e}{k / x^*} \\ = \max_{e \in E} \frac{\lceil k b_e / x^* \rceil}{k b_e / x^*} \leq 1 + \max_{e \in E} \frac{1}{k b_e / x^*} = 1 + \frac{x^*}{k \cdot \min_{e \in E} b_e}. \end{aligned}$$

The theorem trivially follows. □