

MCCL: Measurement-based Collective Communication Library

MICHELE FERRERO, EURECOM, France

GABRIELE CASTELLANO, Independent Researcher, France

PAOLO MEDAGLIANI, Independent Researcher, France

SEBASTIEN MARTIN, LCOMS, Université de Lorraine, France

ROBERTO MORABITO, EURECOM, France

MASSIMO GALLO, Independent Researcher, France

Collective communication is a critical performance bottleneck in large-scale distributed training. Existing libraries and synthesis frameworks compute *open-loop* schedules once, assuming a static topology and peak link capacities. In production, however, persistent gray failures can silently degrade link throughput and latency for hours, causing pre-computed schedules to lose much of their efficiency during execution. In this paper, we present Measurement-based Collective Communication Library (MCCL), a collective communication library that closes the loop between runtime network measurement and dynamic schedule synthesis. MCCL passively measures GPU-to-GPU throughput, detects persistent degradation, re-synthesizes the affected communication-intensive schedules with warm-started synthesizers, and hot-swaps them without interrupting training. We validate the complete loop on a multi-node hardware prototype and use simulations to evaluate cluster-scale scenarios. In our experiments, MCCL incurs negligible overhead ($< 2\%$) while maintaining high measurement accuracy ($< 1.2\%$ error) and low schedule hot-swap time (≈ 12 ms). Notably, under gray failures, MCCL reduces training time by up to $4.99\times$ over static schedules on simulated training workloads for state-of-the-art LLMs such as Llama and DeepSeek.

CCS Concepts: • **Networks** → **Network measurement; Network monitoring; Traffic engineering algorithms; Data center networks**; • **Computing methodologies** → **Distributed algorithms; Distributed artificial intelligence**.

Additional Key Words and Phrases: Collective Communication, Optimization Algorithms, Network Measurements, Runtime Adaptation, Distributed LLM Training

ACM Reference Format:

Michele Ferrero, Gabriele Castellano, Paolo Medagliani, Sebastien Martin, Roberto Morabito, and Massimo Gallo. 2026. MCCL: Measurement-based Collective Communication Library. *Proc. ACM Netw.* 4, CoNEXT4, Article 55 (December 2026), 25 pages. <https://doi.org/10.1145/3856969>

1 Introduction

Large-scale distributed training has become the foundation of modern deep learning, powering state-of-the-art models in natural language processing, computer vision, and recommendation systems. As model parameters and dataset scales continue to grow, the end-to-end efficiency of

Authors' Contact Information: Michele Ferrero, EURECOM, Biot, France, ferrerom@eurecom.fr; Gabriele Castellano, Independent Researcher, Paris, France, gcastellano.phd@gmail.com; Paolo Medagliani, Independent Researcher, Paris, France, paolomeda@gmail.com; Sebastien Martin, LCOMS, Université de Lorraine, Metz, France, martin13@univ-lorraine.fr; Roberto Morabito, EURECOM, Biot, France, roberto.morabito@eurecom.fr; Massimo Gallo, Independent Researcher, Paris, France, gallomassimo3@gmail.com.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2834-5509/2026/12-ART55

<https://doi.org/10.1145/3856969>

distributed training increasingly depends on the performance of collective communications. Primitives such as ALLGATHER, ALLTOALL, ALLREDUCE, and REDUCESCATTER directly influence training time, often accounting for a substantial fraction of the overall iteration latency. Consequently, significant research has focused on optimizing these primitives, with efforts largely concentrated on communication-computation overlap, reduction algorithms, and static topology-aware scheduling.

Hardware accelerator vendors provide Collective Communication Libraries (CCLs) such as Nvidia CCL (NCCL) [26], AMD ROCm CCL (RCCL) [35], Huawei CCL (HCCL) [15], and Intel OneCCL [31], with highly optimized collective communication schedules tailored to common cluster topologies and hardware configurations. These vendor libraries increasingly leverage specialized hardware, such as in-network computing [39], to accelerate REDUCE-intensive collectives. In parallel, schedule synthesizers [3, 5, 23, 34, 36, 43, 48] have demonstrated substantial speedups on communication-intensive primitives such as ALLGATHER and ALLTOALL. However, existing synthesis tools share a critical limitation: they operate in an *open-loop*, relying on static topology assumptions and peak link capacities that rarely hold in production environments.

Datacenter networks typically exhibit complex and dynamic behavior. A critical yet pervasive challenge is the prevalence of *gray failures* [14, 17], where components remain functionally operational but experience severe performance degradation due to thermal throttling, PCIe or Network Interface Card (NIC) symbol errors, and optical link decay. Unlike *hard failures*, which typically trigger alarms, gray failures can silently reduce link performance, sometimes substantially, for several minutes or even hours [14, 46]. Consequently, optimal collective communication schedules at the beginning of training can become increasingly inefficient as system conditions evolve, significantly slowing training processes and leading to suboptimal resource utilization across the distributed infrastructure. This mismatch is particularly pronounced in iterative training because the exact same collectives repeat for thousands of iterations; any localized performance degradation becomes persistent and cumulative.

Our key observation is that the repetitive nature of distributed training turns every collective into a free measurement opportunity. Because identical GPU-to-GPU transfers repeat across thousands of iterations, their progress can be passively monitored at path granularity, without injecting any traffic. This insight enables a fundamentally different operating model for collective communication. Instead of synthesizing a schedule once under static topology assumptions and peak capacities (*open-loop*), the library can continuously *measure* the paths it uses, *decide* when the deployed schedule diverges from actual network conditions, *re-synthesize* it for observed capacities, and *deploy* the new schedule dynamically during training (*closed-loop*).

Realizing this closed-loop architecture introduces three fundamental challenges. First, the system requires path-level visibility without perturbing training. Existing monitoring approaches either rely on synthetic probes, which compete with training traffic, or on aggregate collective completion time, which can detect slowdown but cannot localize the degraded paths. Second, to avoid triggering spurious re-synthesis cycles, the system must reliably distinguish persistent degradations from short-lived fluctuations and pinpoint the affected collectives. Finally, re-synthesis time is the primary barrier to rapid runtime adaptation. Tree-packing synthesizers such as ForestColl [48] can quickly generate ALLGATHER and REDUCESCATTER schedules, yet they lack support for warm-starting; even after an isolated, single-link degradation, they discard structurally valid subtrees and recompute schedules from scratch. The barrier is even higher for ALLTOALL: existing synthesizers [23] solve heavyweight Linear Program (LP) formulations that take hours to converge, too slow for online adaptation. While recent approaches accelerate search times by exploiting network symmetry [5, 37], gray failures inherently break topological symmetry, invalidating their core performance assumptions.

In this paper, we present Measurement-based Collective Communication Library (MCCL)¹, a collective communication library that integrates a closed optimization loop directly into the communication stack. MCCL employs a controller-executor architecture: distributed executors run collectives and passively collect GPU-to-GPU telemetry on the fast path, while a centralized controller detects persistent performance drift, selectively re-synthesizes the affected collectives, and deploys the resulting schedules through a hot-swap mechanism adapted from MCCA [47]. Each new component of MCCL is designed around the requirements of the loop rather than as a standalone optimization. MCCL targets communication-intensive collectives, namely ALLGATHER and ALLTOALL. Reduction-centric collectives such as REDUCESCATTER keep their existing highly optimized implementations, avoiding interference with in-network computing. ALLREDUCE remains compatible with MCCL through its standard decomposition into REDUCESCATTER and ALLGATHER, optimizing its ALLGATHER phase while leaving the reduction phase unchanged.

In summary, this work makes the following contributions:

- We design and implement MCCL, the first collective communication library that closes the loop between path-level measurements and schedule synthesis, continuously adapting communication-intensive schedules to live network conditions without interrupting ongoing training processes (Section 3).
- To fill the gap of existing synthesizers, we present a warm-started version of ForestColl that reuses the structure of deployed ALLGATHER schedules, and a column-generation formulation for ALLTOALL whose rounding gap stays within 2.13 % across tested topologies (Section 4).
- To enable reliable path-level visibility, we design a passive telemetry framework that captures GPU-to-GPU bandwidth metrics at the thread-block level, with < 2 % overhead and < 1.2 % measurement error (Section 5).
- We validate the complete loop (telemetry, failure detection, re-optimization, and hot-swapping) on an 8-GPU, 2-node prototype, where a schedule swap completes in ≈ 12 ms without impacting training-related collective communication scheduling (Section 6).
- We evaluate MCCL at cluster-scale simulating state-of-the-art LLM trainings, demonstrating up to 4.99 $\times$ faster training under gray failures compared to static schedules (Section 6).

The remainder of this paper is organized as follows. We provide background in Section 2, Sections 3–5 detail the system architecture, optimization formulation, and measurement framework. Finally, we evaluate MCCL in Section 6, discuss related work in Section 7, and conclude in Section 8.

2 Background and motivation

2.1 Collective Communication Synthesis

The collective communication research landscape [45] is rapidly evolving, with hand-tuned vendor libraries increasingly complemented by automated schedule-synthesis frameworks. Libraries provided by hardware accelerator vendors, like NCCL [26], RCCL [35], HCCL [15], and OneCCL [31], rely on static, heuristic algorithms (e.g., Ring, Tree) mapped to specific hardware topologies. While these libraries execute reduction-heavy collectives (e.g., ALLREDUCE) with high efficiency by leveraging specialized hardware such as SHARP in-network computing [39], the benefits of such accelerators are largely confined to collectives that involve reduction operations. Communication-intensive collectives, such as ALLGATHER and ALLTOALL, remain primarily constrained by data movement and often cannot fully exploit heterogeneous or non-standard interconnects. Optimizing these collectives is therefore critical to reducing communication overhead and, ultimately, training time. To enable this flexibility, platforms such as MSCCL [25] were developed, enabling the execution of custom collective algorithms on top of standard libraries such as NCCL.

¹MCCL is pronounced “Mickel”.

Name	Synthesis	Optimization	Re-optimization
NCCL[26], RCCL[35] SCCL[3], TACCL[36], TECCL[23]	Static	N.A.	×
ForestColl[48]	Offline	Static, peak topology	Hard
SyCCL[5]	Offline	Static, peak topology	ALLGATHER no AllToAll Hard
OptCCL[37]	Offline	Static, peak topology	with asymmetry Slow with asymmetry
MCCL	Online	Dynamic, measured throughput	Possible

Table 1. Collective Communication Algorithms.

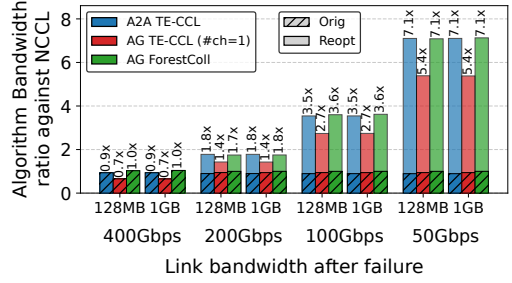


Fig. 1. CCL scheduling algorithm performances.

Building on this foundation, several collective communication synthesizers have emerged to automate schedule generation. Early approaches such as SCCL [3] use Satisfiability Modulo Theories (SMT) solvers to find Pareto-optimal algorithms, while TACCL [36] uses user-provided “communication sketches” to guide synthesis. Subsequent frameworks adopted traffic-engineering principles: TECCL [23] formulates ALLGATHER as an A^* -scaled Mixed Integer Linear Program (MILP) and AllToAll as an LP, while ForestColl [48] constructs spanning tree packs using strongly polynomial-time algorithms to generate throughput-optimal schedules. More recently, frameworks have focused on scaling these optimizations for massive topologies. Concurrent work OptCCL [37] and SyCCL [5] both leverage network and collective symmetry to decompose the scheduling problem, with OptCCL explicitly separating routing from chunk scheduling via a multi-commodity flow LP. However, a critical common denominator across these approaches (cf. Table 1) is the “static world” assumption: they optimize for a topology snapshot taken at initialization time that is assumed to remain constant throughout training.

2.2 Challenges: Scale and Reliability

Training foundation models like LLaMa3 [1], Gemini 2.5 [12], DeepSeek [21] involves big clusters with over 10k GPUs/TPUs running continuously for weeks or months. Furthermore, multiple major AI labs, including OpenAI, Microsoft, xAI, and Meta, are racing to deploy individual training clusters exceeding 100k accelerators, pushing datacenters scale to unprecedented levels. At this scale, network-level component failures (e.g., links, NICs, and optical transceivers), which might be considered rare exceptions in isolation, become a near certainty and pose a critical operational challenge [33]. For instance, estimates suggest that even if each NIC-to-switch link had a mean time to failure of 5 years, the first training job failure would likely occur in less than half an hour within a 100k GPU cluster [10]. Such high hardware failure rates are particularly problematic in distributed training, where a single component failure can cause the entire training job to halt, leaving expensive accelerators idle.

While catastrophic fail-stop events are relatively easy to detect, a more challenging issue is the presence of *gray failures* (or *fail-slow* hardware) [14, 17], where components remain functional but operate at degraded performance levels due to specific recovery mechanisms [2, 32, 42]. These performance degradations are rarely brief anomalies; empirical evidence shows that congestion-induced gray failures persist for an average of 72 minutes (e.g., Greyhound [46]), while hardware-related faults can last significantly longer (e.g., FailSlow [14]). Because these degraded states persist across thousands of training iterations, they become severe long-term bottlenecks.

2.3 The case for dynamic CCL schedules

Given these reliability challenges, we argue for a shift from static planning to dynamic adaptation, as pre-optimized schedules become highly inefficient when communication performance drifts.

To motivate dynamic re-synthesis, we simulated ALLGATHER, and ALLToALL collectives under varying link-layer degradations using SimAI [40, 44], a high-fidelity simulator validated against production clusters (iteration-time and collective-communication deviations below 3.9% [44]). We extend² SimAI to support custom CCL schedules using the same validated point-to-point primitives as its native NCCL algorithms, thereby preserving the simulator’s reported accuracy, as confirmed by our custom ring schedule reproducing SimAI’s native ring completion times. We evaluate a 2-host (16 GPUs) rail-optimized topology (cf. Appendix A, Figure 12 for reference), using NCCL, TECCL, and ForestColl as representatives of state-of-the-art CCL. Synthesizers are evaluated with up to ten chunks, reporting the best-performing configuration in each case. TECCL’s ALLGATHER is limited to one chunk because of scalability issues.

As shown in Figure 1, with peak capacity i.e., all links connecting the nodes to the switch have 400 Gbps, TECCL and ForestColl match NCCL Ring’s Algorithm Bandwidth [27], which is already throughput optimal, with marginal deviations originating from chunking approximation errors. However, when simulating a gray failure by degrading the link capacity of GPU 0 by 50%, 75%, and 88%, the performance of all static schedules (Orig) collapses to the speed of the bottleneck. Strikingly, Figure 1 demonstrates that adapting the schedule via re-optimization yields up to a 7.1× improvement over these static baselines.

2.4 Limitations of Current Approaches

Realizing the theoretical gains of dynamic scheduling requires accurate, real-time detection of performance drifts. Static collective schedule synthesizers rely on active probing i.e., injecting dummy traffic, which is effective offline but slows live training by consuming valuable network resources. Furthermore, relying solely on bandwidth measurements taken from network links or switches is insufficient. While these metrics capture the state of the network fabric, they fail to account for the end-to-end complexity of the communication path. Indeed, traditional network-level measurements [9, 24, 46] overlook critical intra-node factors, such as PCIe bottlenecks and protocol overheads. Consequently, MCCL must explicitly monitor the full GPU-to-GPU data paths. While recent frameworks like MCCA [47] attempt runtime adaptability, their scope is limited to re-ordering ranks within a fixed ring algorithm; they lack the mechanisms to pinpoint path-specific degradations or the capability to synthesize new schedules online.

Moreover, accurate detection alone is not sufficient to enable dynamic schedule adaptation, as existing synthesizers fundamentally lack runtime architectural support. While synthesis tools like ForestColl [48] are faster than traditional solver-based ones, they still operate entirely statelessly. When a localized gray failure occurs, they do not preserve unaffected portions of a previously optimal schedule. Instead, they discard the entire communication structure and rebuild from scratch, forcing redundant exploration and unnecessarily delaying runtime adaptation. For complex patterns like ALLToALL, where every node sends unique data to every other node, the optimization challenge becomes intractable for time-discretized flow-based methods. While recent approaches like SyCCL [5] attempt to mitigate this scaling issue by exploiting topology symmetry, they are rendered ineffective by gray failures, which inherently break symmetry and create irregular network states. Concurrent work OptCCL [37] faces the same limitation: although its LP is polynomial, its scalability relies on structural symmetry [38, Appendix A.11]. Consequently, finding an optimal, asymmetry-aware ALLToALL schedule for a large cluster remains computationally prohibitive for online adaptation.

²<https://github.com/NetMeasurements-Team/SimAI>.

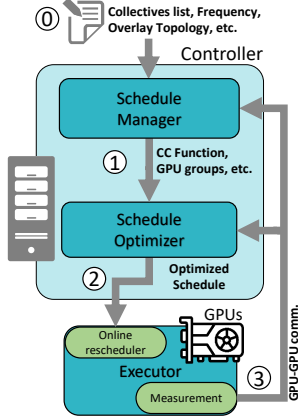


Fig. 2. MCCL System architecture.

Algorithm 1. Re-optimization selection algorithm.

Require: Graph $G(V, A)$, Measurements $\mathcal{M}$, Collectives $\mathcal{S}$ with active schedules, Bottleneck cuts $\{\mathcal{K}_s\}_{s \in \mathcal{S}}$, Threshold Γ , Base slacks $\{base_slack_{s,a}\}_{s \in \mathcal{S}, a \in \mathcal{K}_s}$

Definitions: $C(s)$: Commodities of collective s

$\mathcal{C}_a(s)$: Chunks of the active schedule for collective s crossing link a

$\mathcal{K}_s$: Links saturated by the active schedule of s (its bottleneck cut)

$x_{min}(s)$: Bottleneck data rate of the active schedule for collective s

```

1: for each  $s \in \mathcal{S}$  do
2:    $x_{min}(s) \leftarrow \infty$ 
3:   for each link  $a \in A$  do
4:      $n_a \leftarrow |\mathcal{C}_a(s)|$ 
5:     if  $n_a > 0$  then
6:        $x_{curr} \leftarrow \mathcal{M}_a / n_a$ 
7:        $x_{min}(s) \leftarrow \min(x_{min}(s), x_{curr})$ 
8:    $Slack_{max} \leftarrow 0$ 
9:   for each link  $a \in \mathcal{K}_s$  do
10:     $n_a \leftarrow |\mathcal{C}_a(s)|$ 
11:     $u_a \leftarrow n_a \times x_{min}(s)$ 
12:     $current\_slack \leftarrow (\mathcal{M}_a - u_a) / \mathcal{M}_a$ 
13:     $\delta_a \leftarrow |current\_slack - base\_slack_{s,a}|$ 
14:     $Slack_{max} \leftarrow \max(Slack_{max}, \delta_a)$ 
15:   if  $Slack_{max} > \Gamma$  then
16:     ScheduleOptimizer( $\mathcal{C}(s), \mathcal{M}$ )
```

▶ For each collective and its active schedule —
 ▶ **Phase 1:** Find Bottleneck Rate of s
 ▶ Iterate all links in Topology
 ▶ Update bottleneck of s
 ▶ **Phase 2:** Evaluate Slack on the Cut
 ▶ Iterate only links in the Cut
 ▶ Unused bandwidth on a
 ▶ Deviation from rounding slack
 ▶ **Phase 3:** Decision
 ▶ New active schedule for s

3 System Design

The key idea of MCCL is to shift from a static “optimize-once” model to a dynamic “always-optimizing” framework. To achieve this without delaying the active training job, MCCL utilizes an asynchronous controller-executor architecture, depicted in Figure 2. The **Controller** runs as a background service (e.g., on a dedicated CPU or separate node) and manages the *slow-path*: processing telemetry, diagnosing network health, and executing the schedule synthesis algorithms. The **Executor** runs on the GPUs participating in distributed training and manages the *fast-path*: executing the collectives, passively measuring network throughput, and hot-swapping schedules.

By design MCCL targets communication-intensive operations (ALLGATHER, ALLTOALL). However, distributed training paradigms such as Fully Sharded Data Parallelism (FSDP) rely on operations like ALLREDUCE. MCCL natively supports these workloads through collective decomposition. When a training job triggers an ALLREDUCE, MCCL splits it into its constituent phases. The initial arithmetic-intensive REDUCESCATTER phase is executed using static schedules to leverage specialized in-network aggregation hardware [39]. The subsequent data-intensive ALLGATHER phase is routed through MCCL’s Executor for dynamic optimization. This split preserves hardware-accelerated reduction performance while adapting the data-transfer phase to network conditions.

3.1 The Dynamic Optimization Loop

MCCL operates as a continuous feedback loop (Figure 2), with a *startup phase* for baseline optimization and a *runtime phase* for dynamic adaptation. At startup ①, a profile of the training’s collective calls (e.g., list and frequency) and an initial overlay topology are fed to the *Schedule Manager*. Such topology models communication as overlay links: each GPU connects to an intra-node switch (e.g., NVSwitch), and GPUs in different nodes are connected by direct inter-node links. Initial link capacities are obtained via probing, as for other static synthesizers [6, 23, 36, 48]. In this phase, without online measurements, the *Schedule Manager* ① acts as a static orchestrator, passing the topology to the *Schedule Optimizer* to generate a set of optimized baseline schedules. These are sent to the Executor’s *Online Rescheduler* ② to ensure that training starts with high-performance primitives. Optionally, MCCL can bypass this step and fall back to standard NCCL rings, deferring all optimization to the runtime phase. Once training begins, the system enters the *runtime phase* and transitions to its dynamic loop. While the optimized schedules are executed, the Executor’s *Measurement* module passively monitors overlay links throughput ③. This real-time data is continuously fed back to the Controller, fundamentally changing the *Schedule Manager*’s role.

At runtime, the *Schedule Manager* uses this feedback to diagnose the communication paths' health. It identifies performance drift (e.g., due to gray failures) and dynamically selects which collectives require re-optimization. The *Schedule Manager* then invokes the *Schedule Optimizer* using the measurements from the GPUs as the new input topology. The resulting re-optimized schedules are sent to the *Online Rescheduler* ②, which hot-swaps the algorithms. This continuous loop ensures that collective schedules evolve in lockstep with the underlying network's conditions.

3.2 Controller Components

The Controller operates on the “slow-path” and is composed of two modules:

Schedule Manager. This component acts as the “brain” of the MCCL framework; its primary role is to select which collective(s) to re-optimize based on network conditions. It takes three inputs: the application profile with its graph $G(V, A)$ (step ①), the set of collectives $\mathcal{S}$ with the details of their active execution schedules, and throughput measurements $\mathcal{M}$ (step ③). Executors write raw throughput samples via RDMA into a dedicated per-node memory region on the Controller (Figure 13, Appendix B for reference). The Controller applies an Exponentially Weighted Moving Average (EWMA) with smoothing factor α to these samples; the resulting smoothed throughput for each link a is then stored as $\mathcal{M}_a$. This filtering explicitly attenuates short-lived microbursts, ensuring that only persistent degradations shift $\mathcal{M}$ enough to trigger a re-synthesis cycle.

Algorithm 1 details how the Schedule Manager selects the collectives to re-optimize. In a schedule, tensors are split into chunks, each routed along a path (ALLTOALL) or a tree (ALLGATHER); n_a denotes the number of chunks crossing link a . At synthesis time, the Schedule Optimizer also returns the schedule's *bottleneck cut* $\mathcal{K}_s$, i.e., the links the schedule saturates, and, for each link $a \in \mathcal{K}_s$, its *base_slack* $_{s,a}$, i.e., the fraction of bandwidth left unused on a because the solution is rounded into chunks. At runtime, the Schedule Manager checks whether, for each schedule s , $\mathcal{K}_s$ is still saturated under the current measurements $\mathcal{M}$. This takes three phases. *Phase 1* computes the bottleneck rate $x_{min}(s) = \min_{a: n_a > 0} \mathcal{M}_a / n_a$, which represents the maximum per-chunk throughput allowed by the most congested link. *Phase 2* computes, for each $a \in \mathcal{K}_s$, the current slack $(\mathcal{M}_a - n_a x_{min}(s)) / \mathcal{M}_a$ and its deviation δ_a from *base_slack* $_{s,a}$. *Phase 3* flags s if $\max_{a \in \mathcal{K}_s} \delta_a$ exceeds a threshold Γ , and invokes the Schedule Optimizer with the commodities $\mathcal{C}(s)$ and the current $\mathcal{M}$. Both Γ and the EWMA α are tuned to the cluster profile: a small α filters out longer bursts at the cost of slower detection, while a large Γ ignores milder degradations. Both reduce the number of re-optimizations and thus the risk of *schedule thrashing*, i.e., swapping schedules when degradations clear faster than a synthesis cycle. If a degradation clears during re-synthesis, the Schedule Manager aborts it and keeps the current schedule; if it clears right after, the installed schedule may be suboptimal until the next re-optimization. In practice, thrashing is unlikely, as gray failures typically persist for minutes to hours (Section 2.2), far longer than a re-synthesis cycle. Additional thrashing prevention mechanisms and Γ and α tuning from the observed failure dynamics are left for future work.

Schedule Optimizer. The optimizer serves as a unified wrapper that delegates collective optimization to specialized synthesizers. To ensure broad applicability while maintaining tractability, it targets communication-intensive primitives from which other collectives are derived: ALLTOALL and ALLGATHER. For ALLTOALL, where asymmetric gray failures render traditional symmetry-based synthesizers ineffective, the optimizer invokes a Column Generation (CG)-based synthesizer. This engine enables fast re-synthesis via warm starts from previous solutions, supports time-bounded execution, and optimizes throughput against network irregularities. For ALLGATHER, it interfaces with an extended ForestColl tree-packing synthesizer [48], which integrates an A^* -guided search to warm-start re-synthesis, preserve previous schedule structures, and minimize recomputation overhead. The underlying synthesis algorithms are detailed in Section 4.

3.3 Executor Components

The Executor handles “fast-path” operations on the GPUs. Built on top of the MSCCL [6, 25] backend to inject custom schedules, it consists of two elements:

Online Rescheduler. This module is responsible for the dynamic deployment of new schedules leveraging a runtime “hot-swap” mechanism adapted from M CCS [47]. When the Controller provides a new Optimized Schedule (step ②), the Online Rescheduler stages it. At that point, GPUs belonging to the target communicator execute an out-of-band ALLGATHER to exchange two state variables: the version number of their most recently staged schedule, and the sequence number of their last-launched collective (or collective group). Here, the minimum version number identifies the newest schedule staged across all GPUs, while the maximum sequence number defines the collective selected for the transition. Collectives with previous sequence numbers complete on the old schedule; subsequent ones complete using the re-optimized schedule. Because each swap is scoped to a single communicator, collectives on other communicators are unaffected.

Measurement. This module provides real-time telemetry feeding the optimization loop (step ③). To ensure that measurements capture only actual network communications, telemetry collection is restricted to operations without reduction phases, isolating link throughput from the computational phases of reduction arithmetic. To capture the distinct performance characteristics of different interconnects, the measurement module employs a hybrid metric strategy. For intra-node communication, it measures the cumulative per-GPU throughput, aggregating bandwidth across all concurrent transfers thus capturing the total device injection capacity. For inter-node communication, it captures point-to-point throughput between GPU pairs. These metrics are collected passively during training and asynchronously flushed to the Controller with RDMA³, which records them in a raw measurement matrix later used to derive $\mathcal{M}$ (Appendix B, Figure 13 for reference). The design of this passive measurement system is detailed in Section 5.

4 Collectives Optimization

4.1 Overview

Optimizing collective communication centers on scheduling data-block (i.e., tensor) transfers to minimize completion time. For MCCL, this requires balancing theoretical throughput optimality with the computational scalability required for online deployment. Existing synthesis tools fail to reconcile these conflicting goals because they cannot efficiently manage the optimization state during dynamic network performance drifts. Approaches like TECCL [23] optimize throughput-latency trade-offs but rely on computationally heavy formulations, such as time-expanded graphs or rigid MILP. These methods scale poorly with cluster size and may require substantial solving times, rendering them impractical for online schedule re-computation. Conversely, synthesizers like ForestColl [48] are significantly faster at generating throughput-optimal schedules, but they operate statelessly. When a localized topology change occurs, these algorithms discard the entire communication structure and rebuild the schedule from scratch (i.e., cold start) wasting critical computational cycles and delaying runtime adaptation.

The core principle underlying MCCL’s optimization framework is that network anomalies typically disrupt only a localized fraction of the topology. Rather than re-evaluating the entire search space, MCCL utilizes the previously optimal solution as a structural heuristic to constrain the new search. The shift from a naive cold-start to an efficient warm-start paradigm drastically reduces redundant exploration. As our evaluations demonstrate, using historical schedules to anchor the search reduces convergence times, rendering real-time schedule adaptation practical.

³Telemetry offload leverages compute-bound phases or auxiliary NICs not used for model synchronization during training.

To achieve this, the Schedule Optimizer integrates two specialized synthesis engines tailored to the communication structures of ALLGATHER and ALLTOALL:

ALLTOALL via Column Generation (CG). We propose an LP-throughput-optimal, decomposition-based formulation for ALLTOALL collectives that eliminates solving time overhead associated with traditional time-expanded graphs. By decoupling the problem into a Restricted Master Problem (RMP) and a pricing subproblem, we avoid enumerating the exponentially many tree variables while preserving LP-optimality. Building on this scalable foundation, we also introduce a warm-start mechanism that retains valid routing structures from previous iterations, drastically reducing the time required to adapt to network changes.

ALLGATHER via ForestColl-warm. We introduce ForestColl-warm, which extends the throughput-optimal tree-packing algorithm by incorporating an A^* -guided exploration strategy tailored for online re-computation. By leveraging the solution history to direct the underlying Breadth-First Search (BFS), this mechanism strategically reorders the sequence of candidate links to align with the previous optimal arborescence. This guided approach aggressively prunes the exploration space, minimizing redundant computational overhead and enabling rapid adaptation to network changes. Additionally, in Appendix C we propose a ForestColl-like latency-sensitive formulation which reduces maximum tree depths with the objective of reducing schedules' latency.

4.2 AllToAll optimization

To satisfy the strict time requirements of online re-synthesis, we model the ALLTOALL scheduling problem as a throughput maximization problem, which we solve using a Dantzig–Wolfe reformulation [7] with Column Generation (CG). Let $\mathcal{T}_c$ be the set of all shortest-path trees rooted at the source of tensor $c \in C$. We define z as the throughput, and x_{T_c} as the volume of traffic sent along the shortest-path tree $T_c \in \mathcal{T}_c$ for a given $c \in C$. The optimization problem is formulated as follows:

$$\max \quad z \tag{1}$$

$$\alpha_c : \sum_{T_c \in \mathcal{T}_c} x_{T_c} \geq z \quad \forall c \in C, \tag{2}$$

$$\alpha_a : \sum_{c \in C} \sum_{T_c \in \mathcal{T}_c} \sum_{a \in T_c} \beta_a^{T_c} x_{T_c} \leq c_a \quad \forall a \in A, \tag{3}$$

$$x_{T_c} \geq 0 \quad \forall c \in C, \forall T_c \in \mathcal{T}_c. \tag{4}$$

where $\beta_a^{T_c}$ is the number of shortest paths within the tree T_c that use link a . The objective (1) maximizes the collective throughput z . Constraints (2) are throughput constraints: they tie each tensor's total routed flow to z , so that maximizing z drives up the minimum per-tensor throughput (a max–min objective). Constraints (3) are link-capacity constraints, ensuring the aggregate load on each link a stays within its bandwidth c_a . Finally, constraints (4) enforce non-negativity of the flow variables. We denote the dual variables associated with (2) and (3) as α_c and α_a , respectively.

This formulation is the Dantzig–Wolfe reformulation of a compact, arc-based multi-commodity flow model; we defer the compact model and its full derivation to Appendix G. Directly solving the compact model is computationally prohibitive at scale. The CG approach overcomes this challenge by generating columns on the fly. In particular, the algorithm alternates between solving the RMP, limiting the optimization to a subset of variables, and solving a pricing subproblem for each tensor. The pricing subproblem is parameterized by the dual variables α_c and α_a and identifies whether any variable not yet generated can improve the objective value; specifically, it searches for a shortest-path tree T_c for each $c \in C$ such that $\sum_{a \in T_c} \beta_a^{T_c} \alpha_a < \alpha_c$. Intuitively, α_a duals act as congestion prices, so the left-hand side measures the congestion that T_c would experience, while α_c is the marginal value of increasing the throughput of tensor c .

To account for the sequential nature of tensor transfers within the CG framework, the pricing subproblem's shortest-path tree search is executed over a time-expanded graph $\tilde{G} = (\tilde{V}, \tilde{A})$, where *steps* model the order of discrete chunk transfers between GPUs. To govern this expansion, we bound the maximum number of steps by a parameter m_s . The set of vertices $\tilde{V}$ consists of the original nodes V replicated m_s times, where the i -th copy of a node $v \in V$ is denoted $v_i \in \tilde{V}$. The set of arcs $\tilde{A}$ comprises two types of links to model intra-switch and inter-device propagation: for each link (u, v) towards a switch node v , we add instantaneous links (u_i, v_i) for $i \in \{0, \dots, m_s - 1\}$; for all other links (u, v) where v is not a switch node, we add forward-stepping links (u_i, v_{i+1}) for $i \in \{0, \dots, m_s - 2\}$. Since $\tilde{G}$ is acyclic by construction, the pricing reduces to a shortest-path tree search that we solve with a DAG-tailored algorithm processing $\tilde{V}$ in topological order (i.e., step), which naturally favors shallow trees and runs in $O(|\tilde{V}| + |\tilde{A}|)$ time. This extended topology structurally limits any discovered path to at most m_s steps and natively encourages the shortest-path algorithm to select the routes requiring the fewest steps among equal-cost alternatives. To prioritize the aggressive computational speeds required for online re-computation and warm-starting, we statically fix $m_s = |V| - 1$. While this constant bound guarantees maximum throughput with minimal solver overhead, the formulation can optionally minimize m_s via a binary search if tighter latency bounds are desired. We defer the description of this extended formulation to Appendix D.

Once the CG converges to the fractional LP-optimal solution, we apply deep-dive rounding [18] to extract a schedule with at most ten chunks. This heuristic incurs a marginal gap to the MILP optimum, yielding a near-optimal executable schedule (evaluated in Section 6.1, Appendix F).

4.3 AllGather optimization

In this section, we present our ForestColl extension, whose baseline algorithm comprises three phases that must be executed sequentially:

Phase 1: Compute maximum throughput — This phase calculates the maximum flow for each tensor from a super-source S to the source node of each tensor c within the reversed graph. The overall maximum throughput is determined by the minimum of these computed flows.

Phase 2: Switch removal — This phase abstracts switches by replacing physical paths with direct logical links. The procedure greedily assigns capacity to each logical link (Fig. 14, Appendix C). For each node, the total capacity assigned to incoming and outgoing logical links must not exceed the underlying physical link capacity and must satisfy the calculated maximum throughput.

Phase 3: Tree generation — For every tensor in the logical topology, this procedure allocates link capacity based on the maximum throughput. The allocation uses a Breadth-First Search (BFS) strategy initiated at the tensor's source node, which guarantees an arborescence structure.

History-Guided Tree Generation (ForestColl-warm). To guarantee the rapid schedule re-synthesis required for online adaptation, we enhance the tree generation phase by making it history-aware. The original ForestColl algorithm employs a stateless BFS to construct the arborescence. This approach is computationally wasteful during a localized network anomaly, as it discards the entire communication structure and rebuilds from a cold start.

To improve the re-optimization strategy, our enhanced BFS is guided by the previously computed arborescence through a mechanism analogous to the A^* algorithm. The primary objective is to preserve its structure as much as possible. This approach aggressively reduces redundant explorations within the BFS by reordering the sequence of candidate links. Given that a gray failure typically impacts a limited number of links, there is a high probability that links from prior optimal trees remain valid in subsequent iterations, allowing the solver to rapidly converge on a new throughput-optimal schedule.

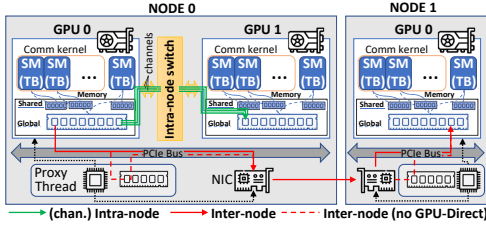


Fig. 3. GPU Communication and Memory Layout.

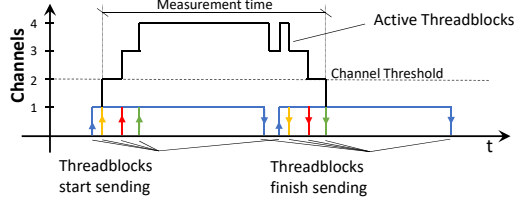


Fig. 4. Per channel (TB) measurements process.

Balancing Latency and Computation Time. While the proposed history-guided search is designed to reduce schedule re-synthesis time, ForestColl’s default greedy switch removal (Phase 2) ignores tree depth, which can lead to sparse topologies and deep communication trees, hence high latency. If rescheduling-time constraints can be relaxed and lower latency is preferred (e.g., for workloads dominated by smaller transfers), a step-aware switch-removal formulation can be used instead, which reduces maximum tree depth as a proxy for latency. We detail this extended formulation in Appendix C.

5 Measurements framework

Before detailing our lightweight passive measurement architecture, we analyze the execution model of the underlying communication library. Without loss of generality, we focus on NCCL, though other software stacks for GPU-to-GPU communication share fundamentally similar architectures.

As detailed in recent literature [16], NCCL parallelizes collective operations by splitting them into virtual pipelines known as *channels*. Each channel manages a fraction of the total payload via a producer-consumer model, transferring data from a source to a destination. At the hardware level, each channel is driven by a single Thread Block (TB) executing on a dedicated Streaming Multiprocessor (SM). For throughput-intensive protocols (e.g., Simple), NCCL partitions the payload across these channels. This enables fine-grained pipelining where computation, transmission, and synchronization occur concurrently. The underlying data movement relies on two distinct mechanisms: *intra-* and *inter-node* as depicted in Figure 3. For intra-node communication, NCCL utilizes the P2P_DIRECT transport. By mapping peer memory into a unified address space, TB issue direct Load/Store instructions to remote GPU memory, bypassing intermediate staging buffers. Inter-node communication instead relies on a Proxy-based protocol run by the Host CPU. The GPU posts work requests to a shared FIFO in host memory, which the Proxy polls to trigger network operations (e.g., RDMA or Socket transfers). With GPU-Direct, chunks are transferred immediately from the GPU to the NIC, bypassing the CPU for the payload.

This decoupled, multichannel execution model presents unique challenges for telemetry: because traffic is split across independent hardware and channels, traditional single-point monitoring is insufficient. This motivates our specialized measurement architecture, which is designed to capture effective bandwidth natively across these distributed execution paths. Currently, our measurement framework targets collective operations without a reduction phase (e.g., ALLGATHER, ALLTOALL) to ensure that the measured throughput is not affected by the computational latency.

5.1 Intra-node measurements

Measuring intra-node throughput presents two main challenges: memory contention and transient effects (naive measurements might underestimate throughput, including transient phases). To mitigate this, we introduce a conditional measurement algorithm supported by a three-tiered memory hierarchy. As illustrated in Figure 5, our measurement module uses three data structures:

Per destination Measurements Accumulators
(GPU – in Global)

Byte_sent	Start	Start_size	Active
4096 B	24 ns	1024 B	4 ch.

Per TB Measurements Cache (GPU – in Shared)

Sender	Receiver	Send Thr.	Count
0	INTRA	2880 Gbps	1
0	8	180 Gbps	3
0	16	-	3

Per TB Measurements Buffer
(pinned CPU/GPU – in Server DRAM)

Sender	Receiver	Flag	Send Thr.	Count
0	INTRA	true	2880 Gbps	1
0	8	true	180 Gbps	3
0	16	true	-	3

Fig. 5. Measurements data structures.

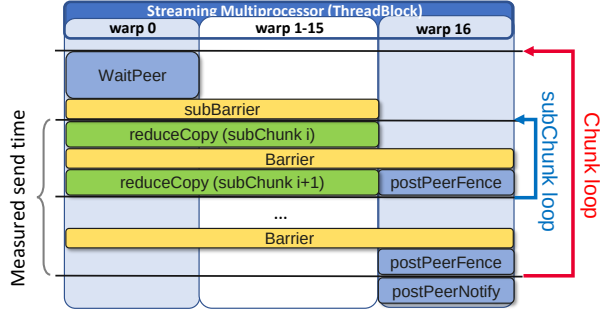


Fig. 6. NCCL communication kernel schema.

Measurements Cache – A per-TB cache located in the fast Shared Memory. It aggregates bandwidth statistics i.e., count, sum_bw, to minimize DMA transfer from the GPU to the Host.

Measurements Accumulator – A per-destination synchronization point stored in GPU global memory and accessible by all TBs. It tracks the global active channel count to detect link saturation, and bytes_sent, the cumulative amount of bytes transmitted to the destination. Additionally, it records start and start_size, the time and byte count at the beginning of the measurement window.

Measurements Buffer – A pinned memory buffer allocated in DRAM to export data from the GPU to the Host. To avoid race conditions, each TB is assigned a set of dedicated slots arranged as a circular buffer; if no slot is available, the measurement is dropped. This structure mirrors the fields of the cache but adds a flag, which signals to the Host that the data is valid for reading.

Measurement Algorithm. To ensure correct measurements, we employ a protocol that records data only when the number of active channels exceeds a specific threshold. This threshold can be established in two ways. The primary method leverages NCCL’s native topology discovery, which derives optimal allocations to maximize parallelism. Alternatively, while NCCL’s allocations are robust, an empirical lower bound can be identified through offline profiling. Note that this static threshold does not need to be adapted in case of reduced bandwidth as the collective algorithm will always use the expected number of channels needed to saturate the nominal bandwidth.

The process, depicted in Figure 4, follows this logic: as a TB begins transmission on a channel, it atomically increments the active counter in the Measurements Accumulator. When active exceeds the defined channel threshold N (cf. Figure 4), the system enters the measurement state and the current TB logs the start timestamp and current bytes_sent. As channels complete their work, they decrement the active counter. When active count drops below the threshold, the measurement window closes. The TB triggering this event calculates the bandwidth ($BW = \frac{\Delta \text{Size}}{\Delta \text{Time}}$) and stores it in the local Measurements Cache.⁴

Implementation efficiency. To isolate chunk transfer times in the NCCL communication kernel (cf. Figure 6), we exclude the initial waitPeer and subBarrier routines that await data readiness. Moreover, to ensure fine-grained data persistence without stalling primary data movers, we restructure the standard NCCL TB signaling mechanism. In the original implementation, a single monolithic postPeer routine handles both memory consistency and peer notification. We split this operation into two distinct primitives: postPeerFence, which enforces memory ordering, and postPeerNotify, which signals completion. This separation allows us to reuse the dedicated signaling warp 16 as a progress bookkeeper, feeding the MCCL measurement data structures.

⁴Note that in our prototype, a flag signals the state of the measurement accumulator i.e., measuring, below_threshold, and finalizing, to prevent a new measurement window from starting before the previous one is finalized.

By executing `postPeerFence` in parallel with the `reduceCopy`⁵ loop (warps 1–15), we enforce system-level memory fences on sub-block transfers without blocking the primary data movers. This effectively masks the latency of synchronization and atomic operations. Our evaluation shows that this instrumentation incurs a negligible overhead of $\approx 2\%$ provided the TB allocation is sufficient to saturate the available bandwidth, a prerequisite for achieving throughput-optimal schedules.

5.2 Inter-node measurements

For inter-node communication, we leverage the CPU Proxy Thread to capture performance metrics. Unlike the highly parallelized execution model of GPU TBs, the Proxy centralizes the management of network operations for a given connection. This significantly simplifies the measurement logic and effectively eliminates the need for atomic operations. Despite these architectural differences, we reuse the same *Measurements Accumulator* and *Measurements Buffer* data structures, as well as the same multi-channel measurement algorithm. The primary distinction is that, in practice, a single communication channel is often sufficient to saturate the available network bandwidth.

The measurement logic is embedded within the Proxy's Finite State Machine (FSM), which manages the life-cycle of network chunks in three stages: *Posted*—a transfer slot is available, *Transmitted*—the network operation, e.g., `isend`, has been triggered, and *Done*—transmission is confirmed as complete. To calculate the effective bandwidth, we isolate network injection and transit time by measuring the interval between *Transmitted* and *Done* states. To ensure accuracy, MCCL models this effective inter-node throughput as the minimum of the local buffer write rate and the network travel rate (i.e., local-to-remote buffer transfer), thereby accounting for potential bottlenecks across the entire data path, i.e., High Bandwidth Memory (HBM), PCIe bus, or network.

6 Experimental Results

We validate MCCL at three different levels. First, we evaluate our optimization algorithms against state-of-the-art competitors in terms of solving time and schedule performance when optimizing a single collective in offline settings. Second, we conduct a closed-loop evaluation on a prototype of MCCL to assess the overhead and accuracy of the measurement module, as well as the online rescheduling mechanism. Finally, we run large-scale simulations of real-life training workloads to quantify end-to-end advantages of MCCL online optimization in terms of training duration.

6.1 Optimization Algorithm

We evaluate solving time and schedule performance against TECCL and vanilla ForestColl, varying payload and group size. For solving-time experiments, we use a 32-core server with a commercial solver, TECCL code [41], and our C++ re-implementation of the released ForestColl [11]. We evaluate all synthesizers with up to ten chunks and report each time the best-performing configuration.⁶ To evaluate schedule performance, we run simulations on a modified version of SimAI [40, 44] that supports custom CCL execution.⁷ We consider HPN rail optimized topologies [33], featuring 8 GPUs per host, each with a dedicated NIC. Intra-node communication relies on an NVSwitch (2880 Gbps, 25 ns delay), while inter-node traffic traverses Top-of-Rack (ToR) switches (400 Gbps, 0.5 μ s delay). The topology is composed of multiple segments, each containing 8 ToR switches and up to 32 hosts (256 GPUs). Aggregation switches interconnect these segments in a single-plane configuration via 400 Gbps links. In the following, we denote by $S \times H \times N$ a topology of S segments, H hosts per segment, and N GPUs per host.

⁵In NCCL kernel, the `reduceCopy` symbol is immutable; for non-reduction operations it does not incur arithmetic overhead.

⁶Due to prohibitive solving times, we limit TECCL to a single chunk (two when specified) when solving ALLGATHER collectives; for each run we report the fastest of its Optimal and Heuristic variants.

⁷<https://github.com/NetMeasurements-Team/SimAI>.

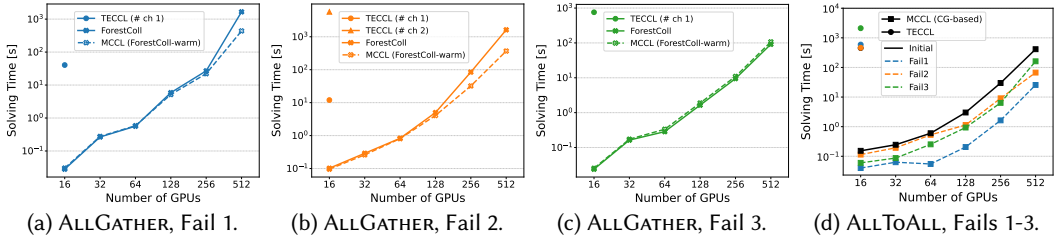


Fig. 7. Optimization solving time comparison for initial schedules and re-optimized one in case of failure.

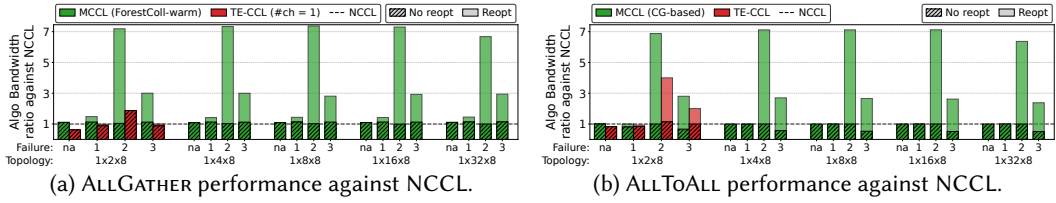


Fig. 8. Schedule performance for ALLGATHER (a), and ALLToALL (b).

Failures. We define three different failures to evaluate our approach under diverse conditions. *Failure 1* (Fail1) affects the NVSwitch of the first host; we simulate that 13 of the 18 lanes connecting GPU 0 fail, and the intra-node bandwidth drops from 2880 Gbps to 800 Gbps. In *Failure 2* (Fail2), we reduce the capacity of the link connecting GPU 0 to its ToR switch from 400 Gbps to 50 Gbps. *Failure 3* (Fail3) models a scenario where GPU 0 fails and is replaced by a backup GPU in a different rack, incurring extra 1 μ s of ToR switch latency. To accommodate the traffic distribution, where the bulk of the load remains with the original node, the backup is provisioned with a dedicated 50 Gbps bandwidth to the faulty node, while communications to the wider data center share 100 Gbps.

Solving Time. We evaluate the scalability of MCCL’s synthesizers against TECCL [23] and vanilla ForestColl [48] varying the cluster size. Figure 7 illustrates the solving times for ALLGATHER (Figures 7a–7c) and ALLToALL (Figure 7d) under the previously described failures. Across all configurations, TECCL proves computationally intractable even for small topologies (up to 10^3 s for a 16-GPU ALLToALL) as also reported in [38, 48]. In contrast, ForestColl and our CG-based algorithm scale significantly better, computing schedules for collectives of up to 512-GPUs. Regarding ALLGATHER collectives, our version of ForestColl enhanced with warm-start is up to 2.58 $\times$ and 2.80 $\times$ faster than the base algorithm for Failures 1 and 2, for which the underlying graph structure is highly reusable. Warm-start yields instead no noticeable improvement for Failure 3, which is significantly more disruptive than Failures 1 and 2 and requires recomputation of a great part of the graph. Finally, Figure 7d shows that our CG-based algorithm greatly benefits from the warm-start mechanism. It re-optimizes ALLToALL collectives in a fraction of the initial solving time (up to 18.3 $\times$ faster) and provides solutions within tens of seconds for 512-GPUs collectives. For this collective family, warm-start is up to 17.8 $\times$ faster than recomputing from scratch (result not shown).

Schedule Performance. To quantify the throughput gains offered by MCCL, we conducted offline simulations on our extended version of SimAI with the NS-3 network backend. Using a 128 MB tensor size per source, we measured the *Algorithm Bandwidth* [27] speedup relative to NCCL. As detailed in Section 4.2, a rounding phase is required for the ALLToALL collective, introducing a maximum error of 2.13 % in our experiments with respect to the optimal. TECCL time-expanded formulation enforces link capacities per epoch, making this conversion inherently harder than our column-generation rounding. This accounts for a gap with respect to MCCL even with no failures, despite using the same chunk budget. Results on MCCL rounding error are reported in Appendix F.

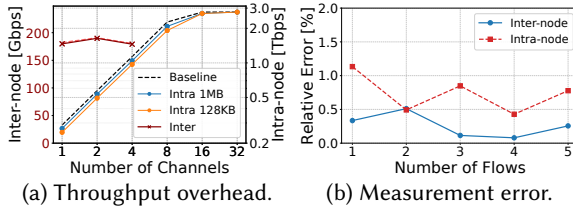


Fig. 9. Performance of the measurement module prototype.

Component	M_{acc}	M_{cache}	M_{buf}
Location	GPU Global	GPU Shared	Host DRAM
Size Formula	$S_{acc} \times (NGPU + 1)$	$S_{cache} \times TB$	$S_{buf} \times TB \times SRV$
Size NGPU=128 SRV=8, TB=1024	4KB	24KB	256KB

Table 2. MCCL Memory Overhead.

Figure 8 shows the performance of ALLToALL and ALLGATHER under the failure scenarios defined above: intra-node degradation, inter-node bottleneck, and GPU replacement. Results highlight the brittleness of static approaches, i.e., “No reopt”. When physical network conditions deviate from pre-computed topology assumptions, their performance collapses to baseline levels ($\approx 1.0\times$). In contrast, MCCL adapts effectively to varying link capacities. For ALLGATHER (cf. Figure 8a), MCCL achieves speedups of up to $7.33\times$ over the baseline in the presence of gray failures. For ALLToALL (cf. Figure 8b), MCCL significantly outperforms static synthesizers with speedups of up to $7.12\times$. It is important to note that MCCL optimizes for computational efficiency. As detailed in Section 3.2, the system employs a cut-based filtering mechanism. For this reason, when failures occur on links outside the critical cut of the active collective, or when a cut is uniformly scaled down, the existing schedule remains throughput-optimal. In these cases, MCCL correctly identifies that the slack is minimized and suppresses re-optimization, thereby conserving precious computational resources.

6.2 MCCL Prototype

We evaluate a prototype implementation of MCCL that builds on MSCCL [25]. The prototype runs on two hosts, each featuring four Nvidia H200 GPUs connected in a rail-optimized topology with Nvidia ConnectX-7 NICs (200 Gbps) using GPU-Direct (details in Table 3).

6.2.1 Measurement Module. We first evaluate the feasibility and efficiency of the MCCL measurement module. Our evaluation focuses on two sets of metrics: (1) *throughput* and *memory overhead*; and (2) *measurement accuracy*. To quantify these metrics, we design targeted benchmarks.

Overhead. We evaluate *throughput overhead* by comparing MCCL direct point-to-point throughput with and without our module, varying the reduceCopy chunk size. This setup represents a worst-case stress test: data is immediately available, rendering waitPeer latency zero and fully exposing measurement costs. Figure 9a illustrates the effective throughput for both inter-node and intra-node transfers as we vary the number of channels, demonstrating that inter-node overhead is negligible because the CPU Proxy manages telemetry without contending for GPU resources. Intra-node overhead peaks at $\approx 13\%$ for low channel counts (1–2) and small chunks (128 kB) due to granular synchronization, i.e., postPeerFence, and atomic operations. However, this drops to $\approx 2\%$ at NVLink saturation (16 channels) and to $< 1\%$ with 32 channels (default in H200), confirming the module is non-intrusive for its target operational point.

To preserve capacity for model parameters, MCCL minimizes *memory overhead* using the three-tiered hierarchy from Section 5.1. The total memory footprint comprises three distinct data structures: the Accumulator in GPU Global Memory (M_{acc}), the Cache in GPU Shared Memory (M_{cache}), and the Buffer in Host-Mapped Memory (M_{buf}). The single-entry sizes for these structures are $S_{acc} = 32\text{ B}$, $S_{cache} = 24\text{ B}$, and $S_{buf} = 32\text{ B}$, respectively. M_{acc} scales with communicating peers, while M_{cache} and M_{buf} scale with thread blocks. As shown in Table 2, even with 128 GPUs and 1024 TBs, the aggregate memory overhead remains negligible, particularly given the low concurrency of collectives during training.

Servers	2 hosts $\times$ 4 GPUs
GPUs	NVIDIA H200
Intra-node link	NVLink 4.0 (NVSwitch) NV18, ~ 900 GB/s/GPU
Inter-node NIC	ConnectX-7 1 NIC/GPU, PCIe 5.0 $\times$ 16
Inter-node link rate	200 Gbps IB
Out-of-band NIC	Mellanox ConnectX-5
Out-of-band link rate	100 Gbps Ethernet

Table 3. Prototype testbed.

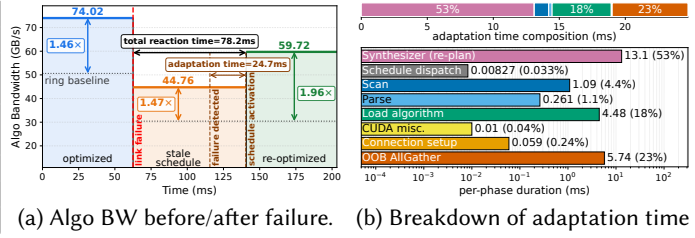


Fig. 10. Prototype evaluation of MCCL.

Accuracy. For the *accuracy* assessment, we compare the telemetry obtained from MCCL against application-level ground truth (pytorch) by varying the number of concurrent flows to emulate network contention. Figure 9b shows the relative error between the MCCL-reported throughput and the actual available bandwidth. The system demonstrates low estimation error for both inter and intra-node communication, maintaining a consistently low error rate ($< 1.2\%$). This accuracy allows the Controller to distinguish between healthy connections and gray failures, validating the module’s suitability for driving the online optimization loop.

6.2.2 Hot-Swap Mechanism. We implement the hot-swap mechanism building on top of MSCCL for on-demand connection and dynamic algorithm loading [25]. When the controller issues a new schedule version, the system delegates the transition to a dedicated background thread. This is responsible for asynchronously loading the new algorithmic routing and establishing the necessary peer-to-peer connections without interrupting ongoing communications. If any, existing connections are reused, matching NCCL’s default behavior [29].

Once setup completes, the background thread briefly pauses the issuance of new collectives on the communicator and executes an out-of-band (OOB) ALLGATHER, adapted from MCCS [47]. Each GPU reports the latest version among its successfully loaded schedules and the sequence number of its last-issued collective operation. The minimum schedule version ensures that the selected schedule is available on every GPU, while the maximum sequence number K defines the schedule transition point: all operations up to K use the old schedule, and subsequent collective operations use the new one. Since NCCL preserves operation order within a communicator and serializes their device execution [28, 30], the two schedules never execute concurrently, and no draining of in-flight operations is required. Swaps are scoped independently to each communicator and do not interfere with one another.

Figure 10a illustrates the reaction of the system during an inter-node link failure, which halves the available bandwidth between GPU ranks 0 and 4 (the first GPU of each node) to 100 Gbps. We evaluate a 16 MB per-GPU ALLGATHER, matching the per-GPU input tensor size of the dominant collective in Llama 405B. Under the stale schedule, the degradation reduces algorithm bandwidth [27] from 74.02 GB/s to 44.76 GB/s. The controller detects the failure in 53.1 ms; this window is primarily spent gathering sufficient degraded telemetry samples to prevent false positives from transient anomalies (we use a conservative smoothing factor $\alpha = 0.012$). The controller then runs the synthesizer and dispatches the new schedule to the executors, whose Online Rescheduler activates it 24.7 ms after detection. Meanwhile, collectives continue running on the stale schedule: all in-flight operations (here, a single ALLGATHER) complete on it, while the next submitted collective switches to the new schedule, starting execution 0.4 ms later for a total reaction time of 78.2 ms. Once executed, algorithm bandwidth recovers to 59.72 GB/s without interrupting training execution.

Figure 10b breaks down the adaptation time of 24.7 ms: 13.1 ms is spent in the synthesizer invoked by the Schedule Optimizer, while the hot-swap takes 11.6 ms, mainly for OOB synchronization (5.74 ms) and algorithm loading (4.48 ms). As node count grows, re-optimization increasingly

Metric	Llama 405B						DeepSeek 671B					
	AllGather		ReduceSc.		AllReduce		AllGather		ReduceSc.		AllToAll	
Group (#GPUs)	DP (64)	TP (16)	DP (64)	TP (16)	TP (16)		DP (1024)	DP_EP (16)	EP (64)	DP (1024)	DP_EP (16)	EP (64) EP (64)
Size	2.6 GB	256 MB	5.3 GB	256 MB	256 MB		1.2 GB	0.9 GB	32 KB	2.4 GB	1.9 GB	448 MB 231 MB
Frequency	16	524K	16	524K	16K		1	64	768	1	64	2304 768
Training time	0.167%	18.0%	0.555%	15.6%	7.36e-4%		0.136%	4.11%	0.141%	0.266%	8.22%	44.5% 7.66%

Table 4. Llama 405B and DeepSeek 671B (batch size 16K) collectives for one pipeline stage with 1024 GPUs.

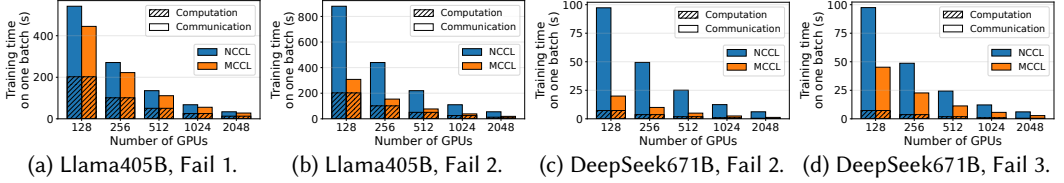


Fig. 11. Training time for Llama405B (a),(b) and DeepSeek671B(c),(d) with different failures profiles.

dominates this latency (cf. Section 6.1), making the swap overhead comparatively small. Moreover, hot-swap operations run on dedicated threads, remaining off the critical execution path; only the OOB ALLGATHER pauses the issuing of new collectives for 5.74 ms. In practice, for pipelined large-scale training workloads this brief CPU-side pause overlaps with ongoing GPU execution.

6.3 End-to-End Validation

To validate the end-to-end benefits of our approach, we run large-scale simulations in SimAI. We use an HPN topology with segments of 128 GPUs each (16 hosts) and scale the cluster from 128 to 2048 GPUs; we apply the same three failure scenarios as in Section 6.1. Because detection parameters (Γ, α) are inherently cluster-specific, calibrating them to our simulation offers little generalizable insight. We therefore decouple condition detection from schedule adaptation, evaluating MCCL's re-synthesis and failover in isolation under prompt failure notifications, in regimes where adaptation proceeds without schedule thrashing. As a baseline, we compare against plain NCCL. We exclude TECCL due to impractical runtime solving time, and ForestColl as it cannot optimize ALLTOALL collectives, while it delivers the same schedule quality as MCCL for ALLGATHER collectives.

We design two sets of workloads that mimic the training of Llama3.1 with 405B parameters and DeepSeek-v3 with 671B parameters and 256 experts [1, 21]. We configure a pipeline parallelism of 16 and only simulate one pipeline stage as traffic between stages is point-to-point. Therefore, our simulations with 128–2048 GPUs correspond to training the full model on 2048–32768 GPUs. We execute a forward and backward pass on a batch of 16K tokens. Table 4 details collective communications of both workloads, each with its share over the total training time. In our settings, collective communications account for $\approx 34\%$ of Llama's training time, and for $\approx 65\%$ of DeepSeek's.⁸

Figure 11 reports the training time for Llama and DeepSeek under the failures described above. The results of all failure profiles are reported in Appendix E. When no failures are applied, MCCL provides no advantages over NCCL (cf. Appendix E); this is because this topology has, ring-wise, a perfect ratio between inter- and intra-node bandwidth. Under failure conditions, MCCL significantly reduces training time, with peaks of $2.85\times$ and $4.99\times$ under failure 2 for Llama and DeepSeek, respectively. Finally, MCCL scales effectively with increasing GPU counts, yielding similar relative gains across different cluster sizes.

⁸In our end-to-end experiments, we do not optimize the 1024-GPU ALLGATHER in DeepSeek, as it accounts for only 0.136% of the total execution time, making the associated synthesizer overhead unjustified in practical settings. Moreover, REDUCESCATTER collectives operate at the host level, as the simulator does not support in-network reduction.

7 Related Work

Offline schedule synthesis. A large body of work synthesizes collective schedules offline. TACCL [36], TE-CCL [23], and ForestColl [48] build topology-aware schedules from communication sketches, traffic-engineering formulations, and spanning trees, respectively, outperforming vendor-provided collectives. However, they rely on static, offline profiling of the topology at peak capacity and cannot promptly react to link degradation, multi-tenant interference, or gray failures. The concurrent work OptCCL [37] synthesizes provably optimal schedules from a multi-commodity flow LP solved via Dantzig–Wolfe decomposition, with an ALLToALL formulation close to ours (Section 4.2). OptCCL scales the decomposition by exploiting topology symmetry, which gray failures inherently break, forcing a fallback to a cold start. MCCL instead exploits the locality of re-optimization, reducing re-synthesis time without relying on symmetry. The two approaches are complementary; moreover, OptCCL neither detects degradations nor redeploys schedules at runtime.

Runtime adaptation. A second line of work adapts at runtime, but along axes orthogonal to ours. FAST [20] schedules MoE’s skewed, dynamic AllToAllv online, rebalancing over the fast intra-server fabric so that every NIC sends equal volume across the slow inter-server links. Its adaptation tracks *workload* skew, not *network* state: it models the cluster as two uniform, equal-capacity bandwidth tiers, so its balancing breaks under link degradation, asymmetric capacity, or gray failures. At the inter-job level, Crux [4] and MCCC [47] arbitrate the network shared by co-located jobs: Crux prioritizes flows by GPU intensity (compute-to-communication ratio), routing GPU-heavy jobs away from contended links, while MCCC, a provider-managed service, reorders ring ranks and pins flows to paths to preserve fairness and reduce congestion. Both re-optimize only at job arrival/exit or on external triggers, leaving collective schedules unchanged and thus exposed to gray failures. The concurrent work Theseus [8] hot-swaps schedules at runtime according to user policies, but neither synthesizes schedules nor measures the network: it relies on slow MILP-based synthesizers or precomputed candidates, and on NIC statistics that miss intra-host degradations (e.g., NVLink, PCIe). MCCL instead measures each GPU-to-GPU path and re-synthesizes within tens of seconds, only when the bottleneck cut loses saturation.

Execution backends. A final line of work provides the execution backends. NCCL and RCCL ship static, built-in algorithms, while MSCCL [6], MSCCL++ [19], and ResCCL [22] add DSL-defined schedules, portable low-level primitives, and resource-efficient thread-block scheduling, respectively. They improve expressiveness, portability, and efficiency, but still execute static, offline-generated schedules and cannot adapt to link variability, congestion, or gray failures.

8 Conclusion

This paper introduced a Measurement-based Collective Communication Library (MCCL) that shifts the paradigm from “optimize-once” to “continuously-adaptive”. By leveraging a controller-executor architecture, MCCL integrates lightweight, passive, high-fidelity GPU telemetry directly into the optimization loop without disrupting the training workload. Our evaluation demonstrates that MCCL re-optimization algorithms rapidly synthesize schedules adapted to detected bottlenecks, improving solving times by up to $2.80\times$ for ALLGATHER and by up to $17.8\times$ for ALLToALL compared to static synthesizers. The MCCL prototype demonstrates the feasibility of the complete closed loop, with low measurement overhead ($< 2\%$), high accuracy ($< 1.2\%$ error), and fast schedule swaps (≈ 12 ms). Finally, large-scale simulations of LLM training workloads show that MCCL reduces training time by up to $4.99\times$ under gray failures compared to static schedules.

This work does not raise any ethical issues.

References

- [1] Grattafiori Aaron, Dubey Abhimanyu, Jauhri Abhinav, and et al. The llama 3 herd of models, 2024.
- [2] Sobhani Ashkan, Sadrhaghighi Sogand, and Chu Xingjun. Prime: Pseudo-random integrated multi-part entropy for adaptive packet spraying in ai/ml data centers, 2025.
- [3] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '21, page 62–75, New York, NY, USA, 2021. Association for Computing Machinery.
- [4] Jiamin Cao, Yu Guan, Kun Qian, Jiaqi Gao, Wencong Xiao, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. Crux: Gpu-efficient communication scheduling for deep learning training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 1–15, 2024.
- [5] Jiamin Cao, Shangfeng Shi, Jiaqi Gao, Weisen Liu, Yifan Yang, Yichi Xu, Zhilong Zheng, Yu Guan, Kun Qian, Ying Liu, Mingwei Xu, Tianshu Wang, Ning Wang, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. Sycccl: Exploiting symmetry for efficient collective communication scheduling. In *Proceedings of the ACM SIGCOMM 2025 Conference*, SIGCOMM '25, page 645–662, New York, NY, USA, 2025. Association for Computing Machinery.
- [6] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. Mscclang: Microsoft collective communication language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 502–514, New York, NY, USA, 2023. Association for Computing Machinery.
- [7] George B. Dantzig and Philip Wolfe. Decomposition principle for linear programs. *Operations Research*, 8(1):101–111, 1960.
- [8] Rui Ding, Xiangdong Lu, Jiajun Wang, Xunpeng Liu, Feiyang Wang, Xuran Hao, Houyuan Zhu, Anyi Xu, Sinuo Cao, Haifeng Sun, Qun Huang, Jiamin Cao, and Jiaqi Gao. Theseus: Runtime-adaptive gpu collective communication with hot-swappable schedules. In *Proceedings of the ACM SIGCOMM 2026 Conference*, SIGCOMM '26, page 222–241, New York, NY, USA, 2026. Association for Computing Machinery.
- [9] Gen Dong, Yu Hua, Yongle Zhang, Zhangyu Chen, and Menglei Chen. Understanding and detecting fail-slow hardware failure bugs in cloud systems. *USENIX ATC '25*, USA, 2025. USENIX Association.
- [10] Patel Dylan and Nishball Daniel. 100,000 h100 clusters: Power, network topology, ethernet vs infiniband, reliability, failures, checkpointing. <https://newsletter.semianalysis.com/p/100000-h100-clusters-power-network>, 2024.
- [11] Forestcoll github. <https://github.com/liangyuRain/ForestColl>, 2025.
- [12] Comanici Gheorghe, Bieber Eric, Schaeckermann Mike, and et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025.
- [13] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981.
- [14] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Deepthi Srinivasan, Biswaranjan Panda, Andrew Baptist, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage*, 14(3):1–26, August 2018.
- [15] Huawei Collective Communication Library (HCCL). <https://gitee.com/ascend/cann-hccl>, 2026.
- [16] Zhiyi Hu, Siyuan Shen, Tommaso Bonato, Sylvain Jeaugey, Cedell Alexander, Eric Spada, James Dinan, Jeff Hammond, and Torsten Hoefer. Demystifying NCCL: An In-depth Analysis of GPU Communication Protocols and Algorithms. Aug. 2025.
- [17] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. *HotOS '17*, page 150–155, New York, NY, USA, 2017. Association for Computing Machinery.
- [18] Nicolas Huin, Jérémie Leguay, Sébastien Martin, and Paolo Medagliani. Routing and slot allocation in 5g hard slicing. *Computer Communications*, 201:72–90, 2023.
- [19] Changho Hwang, Peng Cheng, Roshan Dathathri, Abhinav Jangda, Saeed Maleki, Madan Musuvathi, Olli Saarikivi, Aashaka Shah, Ziyue Yang, Binyang Li, Caio Rocha, Qinghua Zhou, Mahdieh Ghazimirsaeed, Sreevatsa Anantharamu, and Jithin Jose. Msccl++: Rethinking gpu communication abstractions for ai inference. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, page 1201–1215. ACM, March 2026.
- [20] Yiran Lei, Dongjoo Lee, Liangyu Zhao, Daniar Kurniawan, Chanmyeong Kim, Heetaek Jeong, Changsu Kim, Hyeon-seong Choi, Liangcheng Yu, Arvind Krishnamurthy, Justine Sherry, and Eriko Nurvitadhi. Fast: An efficient scheduler for all-to-all gpu communication, 2026.

- [21] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.
- [22] Tongrui Liu, Chenyang Hei, Fuliang Li, Chengxi Gao, Jiamin Cao, Tianshu Wang, Ennan Zhai, and Xingwei Wang. Resccl: Resource-efficient scheduling for collective communication. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 55–70, 2025.
- [23] Xuting Liu, Behnaz Arzani, Siva Kesava Reddy Kakarla, Liangyu Zhao, Vincent Liu, Miguel Castro, Srikanth Kandula, and Luke Marshall. Rethinking machine learning collective communication as a multi-commodity flow problem. In *Proceedings of the ACM SIGCOMM 2024 Conference*, page 16–37, Sydney NSW Australia, August 2024. ACM.
- [24] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. Fast in-network gray failure detection for isps. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 677–692, New York, NY, USA, 2022. Association for Computing Machinery.
- [25] Microsoft collective communication library (msccl). <https://github.com/microsoft/msccl>, last updated 2023.
- [26] Nvidia collective communication library. <https://github.com/NVIDIA/nccl>, 2026.
- [27] Algorithm bandwidth. <https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md#algorithm-bandwidth>, 2026.
- [28] NVIDIA. NCCL source code: kernel launch path (enqueue.cc), per-communicator device stream. <https://github.com/NVIDIA/nccl/blob/v2.28.9-1/src/enqueue.cc>. Accessed: 2026-09.
- [29] NVIDIA. NCCL user guide 2.28.9: Environment variables – NCCL_RUNTIME_CONNECT. https://docs.nvidia.com/deeplearning/nccl/archives/nccl_2289/user-guide/docs/env.html#nccl-runtime-connect. Accessed: 2026-09.
- [30] NVIDIA. NCCL user guide 2.28.9: Group calls – group operation ordering semantics. https://docs.nvidia.com/deeplearning/nccl/archives/nccl_2289/user-guide/docs/usage/groups.html#group-operation-ordering-semantics. Accessed: 2026-09.
- [31] Intel OneAPI Collective Communication Library. <https://github.com/uxlfoundation/oneCCL>, 2026.
- [32] Leon Poutievski, Omid Mashayekhi, Joon Ong, and et al. Jupiter evolving: transforming google’s datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 66–85, New York, NY, USA, 2022. Association for Computing Machinery.
- [33] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 691–706, New York, NY, USA, 2024. Association for Computing Machinery.
- [34] Saeed Rashidi, William Won, Sudarshan Srinivasan, Srinivas Sridharan, and Tushar Krishna. Themis: a network bandwidth-aware collective scheduling policy for distributed training of dl models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 581–596, New York, NY, USA, 2022. Association for Computing Machinery.
- [35] Amd rocm collective communication library. <https://github.com/ROCm/rccl>, 2026.
- [36] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. TACCL: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 593–612, Boston, MA, April 2023. USENIX Association.
- [37] Richard Shapley, Rachit Agarwal, and David Shmoys. Optccl: Scalable synthesis of optimal collective communication algorithms. In *Proceedings of the ACM SIGCOMM 2026 Conference*, SIGCOMM '26, page 177–189, New York, NY, USA, 2026. Association for Computing Machinery.
- [38] Richard Shapley, Rachit Agarwal, and David Shmoys. OptCCL: Scalable synthesis of optimal collective communication algorithms. Technical report, GitHub / ML-Collectives, 2026. <https://github.com/ml-collectives/optccl/blob/main/optccl-techreport.pdf>.
- [39] In-Network Computing with Nvidia SHARP. <https://resources.nvidia.com/en-us-accelerated-networking-resource-library/network-computing-nvidia-sharp>.
- [40] Alibaba cloud - simai. <https://github.com/aliyun/SimAI>, last updated 2026.
- [41] Teccl github. <https://github.com/microsoft/TE-CCL>, 2024.
- [42] Bonato Tommaso, Kabbani Abdul, Ghalayini Ahmad, Papamichael Michael, Dohadwala Mohammad, Gianinazzi Lukas, Khalilov Mikhail, Achermann Elias, De Sensi Daniele, and Hoefler Torsten. Reps: Recycled entropy packet spraying for adaptive load balancing and failure mitigation, 2025.
- [43] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. In *Proceedings of Machine Learning and Systems*, volume 2, page 172–186, 2020.
- [44] Xizheng Wang, Qingxu Li, Yichi Xu, Gang Lu, Dan Li, Li Chen, Heyang Zhou, Linkang Zheng, Sen Zhang, Yikai Zhu, Yang Liu, Pengcheng Zhang, Kun Qian, Kunling He, Jiaqi Gao, Ennan Zhai, Dennis Cai, and Binzhang Fu. SimAI:

- Unifying architecture design and performance tuning for Large-Scale large language model training with scalability and precision. pages 541–558, April 2025.
- [45] Adam Weingram, Yuke Li, Hao Qi, Darren Ng, Liuyao Dai, and Xiaoyi Lu. xccl: A survey of industry-led collective communication libraries for deep learning. *Journal of Computer Science and Technology*, 38(1):166–195, February 2023.
 - [46] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. Greyhound: hunting fail-slows in hybrid-parallel training at scale. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '25, USA, 2025. USENIX Association.
 - [47] Yongji Wu, Yechen Xu, Jingrong Chen, Zhaodong Wang, Ying Zhang, Matthew Lentz, and Danyang Zhuo. Mccs: A service-based approach to collective communication for multi-tenant cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*, page 679–690, Sydney NSW Australia, August 2024. ACM.
 - [48] Liangyu Zhao, Saeed Maleki, Yuanhong Wang, Zezhou Wang, Ziyue Yang, Hossein Pourreza, and Arvind Krishnamurthy. Forestcoll: Throughput-optimal collective communications on heterogeneous network fabrics. <https://arxiv.org/abs/2402.06787>, 2025.

A Rail-optimized Topology

Figure 12 details the rail-optimized topology used in our evaluation, specifically highlighting a single segment consisting of dual servers with an 8-GPU configuration.

B Measurement Matrix

Figure 13 visualizes the passive measurement data structure described in Section 5. Each GPU fills its corresponding row using *RDMA writes* during collective execution. The diagonal represents high-bandwidth intra-node communication (NVLink), while off-diagonal blocks represent inter-node communication, providing a real-time heatmap of cluster network health.

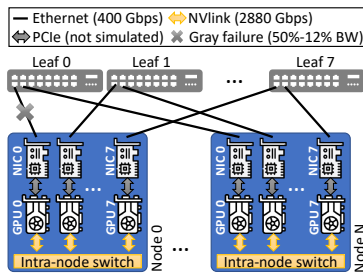


Fig. 12. Rail Optimized topology showing links between GPU nodes and leaf switches.

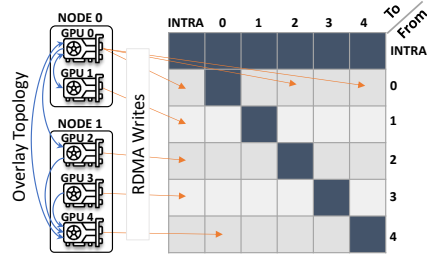


Fig. 13. Per collective type, size, and communicator group point-to-point measurements. Darker blocks indicate intra-node bandwidth, lighter blocks inter-node connections.

C Latency-Aware Switch Removal for AllGather

ForestColl switch removal [48] only optimizes throughput, ignoring topological diameter and often yielding sparse structures (e.g., rings). Thus the tree-generation phase often builds excessively deep trees. We propose a formulation that bounds tree height and reduces communication steps.

Overlay to Logical Topology — Given a node with n GPUs and one high-speed switch, and being G_i the subgraph of the i -th node's GPUs, switch, and links, switch removal replaces G_i with a logical topology $\bar{G}_i = (\bar{V}_i, \bar{A}_i)$ where vertices $\bar{V}_i$ are the node's GPUs and arcs $\bar{A}_i$ connect all pairs of GPUs linked by a communication path. Each GPU's outgoing capacity is split across its logical links.

Improved Switch Removal Formulation — We remove one switch at a time. When processing switch w , $\bar{G}_i = (\bar{V}_i, \bar{A}_i)$ is the local graph on the predecessors $\delta^-(w)$ and successors $\delta^+(w)$ of w ; its candidate *logical links* $\bar{A}_i = \{(u, v) : u \in \delta^-(w), v \in \delta^+(w), u \neq v\}$ join each predecessor to each successor. Each $a = (u, v)$ carries a capacity variable $y_a \geq 0$, and we seek values that reproduce, between predecessors and successors, the connectivity y provided. We write $N = |\bar{V}_i|$, c_a for physical capacities, and c_{uw} (resp. c_{wv}) for the ingress (resp. egress) capacity of w .

To keep trees shallow, a secondary objective spreads each node's traffic over many successors. Being $\delta^+(w)$ the set of successors of w , we add a convex, piecewise-linear penalty h_{uv} whose k -th piece has slope k and whose breakpoints are integer multiples of the width

$$w_{uv} = \frac{c_{uw}}{|\delta^+(w)| - 1};$$

thus w_{uv} only depends on u and equals the fair per-successor share of u 's ingress—we round w_{uv} down to the nearest integer to keep the constraint coefficients integral. Since the marginal penalty grows with the capacity placed on a single link, concentrating flow costs more than spreading it, biasing the logical topology toward a low-depth star. The throughput objective takes priority, so we combine the two lexicographically with a single big- M weight: any M strictly larger than the penalty's maximum value suffices. At optimality, each h_{uv} equals $\max_{1 \leq k \leq K} \{k y_a - w_{uv} \frac{k(k-1)}{2}\} \leq K y_a$, with $K = |\delta^+(w)| - 1 \leq N - 1$. Summing over $\bar{A}_i$ and using the predecessor bounds $\sum_v y_{uv} \leq c_{uw}$,

$$\sum_{a=(u,v) \in \bar{A}_i} h_{uv} \leq K \sum_{a \in \bar{A}_i} y_a \leq (N-1) \sum_{u \in \delta^-(w)} c_{uw} \leq (N-1) N c_{\max} < N^2 c_{\max},$$

where $c_{\max} = \max_a c_a$. We therefore set $M = N^2 c_{\max}$, so improving throughput by one unit always outweighs any attainable change in the penalty; the penalty only discriminates among throughput-optimal solutions. The switch removal is formulated as follows:

$$\max M \sum_{a \in \bar{A}_i} y_a - \sum_{a=(u,v) \in \bar{A}_i} h_{uv} \quad (5) \quad \sum_{u: (u,v) \in \bar{A}_i} y_{uv} \leq c_{wv} \quad \forall v \in \delta^+(w), \quad (8)$$

$$h_{uv} \geq k y_a - w_{uv} \frac{k(k-1)}{2} \quad \forall a = (u, v) \in \bar{A}_i, k \in \{1, \dots, K\} \quad (6) \quad \sum_{a \in A_{V'}} y_a \geq MT - OF_{V'} \quad \forall V' \subseteq V, \quad (9)$$

$$\sum_{v: (u,v) \in \bar{A}_i} y_{uv} \leq c_{uw} \quad \forall u \in \delta^-(w), \quad (7) \quad y_a, h_{uv} \geq 0 \quad \forall a = (u, v) \in \bar{A}_i. \quad (10)$$

In (5), $M \sum_a y_a$ maximizes the total logical capacity installed between the predecessors and successors of w (primary throughput goal) and $-\sum h_{uv}$ applies the star-inducing penalty (secondary goal). Constraints (6) define h_{uv} as the lower envelope of the convex penalty above. Constraints (7)–(8) preserve capacity: links leaving a predecessor u carry at most its ingress capacity c_{uw} , and links entering a successor v at most its egress capacity c_{wv} , so the logical topology is realizable on the fabric. Constraints (9) are the throughput (min-cut) constraints: across every cut $V' \subseteq V$, the flow must be at least the maximum throughput MT , where $OF_{V'}$ is the flow crossing outside the switch's logical links and $A_{V'} = A_{V'} \cap \bar{A}_i$ is the logical part of the cut. Finally, (10) enforces non-negativity.

The min-cut constraints are exponential in number, so we solve the model by cutting planes: at each iteration we compute, for every destination GPU, the maximum flow from the source under the current y ; if any is below MT we add the corresponding cut (9), otherwise the relaxation is optimal. By cutting-plane theory [13], the number of generated constraints is polynomial, $O(n^\kappa)$, in the number of variables n (κ constant).

Figure 14 shows an example intra-node overlay-to-logical mapping, and Figure 15 illustrates a cut added by the cutting-plane algorithm during switch removal, ensuring the generated trees respect the bandwidth limits identified by the max-flow formulation.

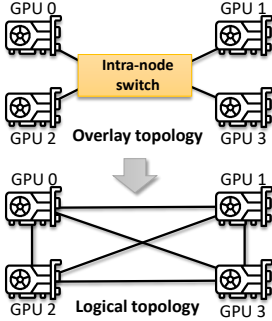


Fig. 14. Overlay to Logical topology.

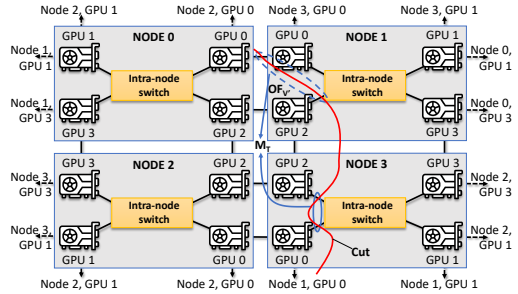


Fig. 15. Example cut induced by the max-flow in the switch removal phase, enforcing capacity constraints on the logical topology.

D Minimizing AllToAll Steps via Binary Search

The online loop fixes the step bound to $m_s = |V| - 1$ for fast re-synthesis under persistent anomalies, but when computational time allows, it's possible to minimize steps without sacrificing throughput. Let MT be the optimal throughput at $m_s = |V| - 1$. We binary-search the smallest bound $m_s^* \leq |V| - 1$ that still attains MT : from $L = 1$, $U = |V| - 1$, each iteration reduces the time-expanded graph $\tilde{G}$ to $m_{test} = \lfloor (L + U)/2 \rfloor$ steps and runs CG. If MT is reached we set $U = m_{test} - 1$ and cache the schedule; if the model is infeasible or falls short of MT we set $L = m_{test} + 1$. To accelerate convergence we warm-start via MCCL's mechanism, injecting columns from prior iterations into the RMP. The search converges logarithmically to the minimal throughput-optimal step count.

E End-to-End Validation

Figures 16 and 17 show complete end-to-end training time for Llama 405B and DeepSeek 671B, expanding on the results from Section 6.3 under all failure scenarios: no failure, Failure 1 (Switch), Failure 2 (Link degradation), and Failure 3 (GPU replacement).

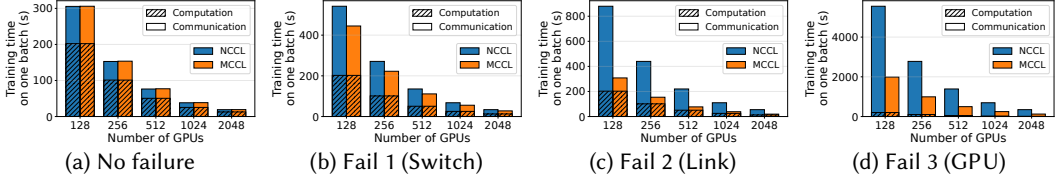


Fig. 16. **Llama 405B Training Time.** Compare NCCL and MCCL for one pipeline stage (out of 16).

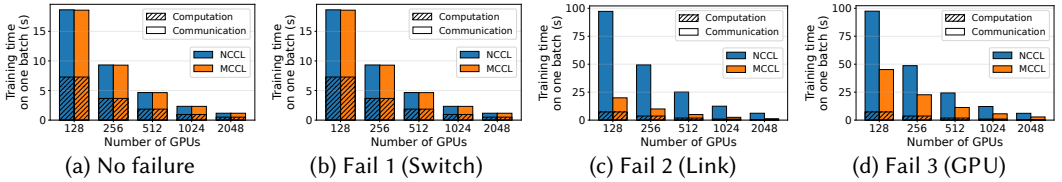


Fig. 17. **DeepSeek 671B Training Time.** Compare NCCL and MCCL for one pipeline stage (out of 16).

F Rounding

Table 5 shows results for the rounding process. The rounding error is computed between the optimal MILP solution and the rounded LP. We used 60 cores, 1TB of RAM and a time limit of 1 hour for each {topology-#chunks} combination and the #chunks is searched between 1 and 10.

Metric	1x2x8				1x4x8				1x8x8			
	Base	Fail 1	Fail 2	Fail 3	Base	Fail 1	Fail 2	Fail 3	Base	Fail 1	Fail 2	Fail 3
LP Bound	0.6250	0.6250	4.4531	1.0000	16.6250	16.6250	14.8125	4.8065	2.3750	7.1250	6.3438	2.3492
Rounded Obj	0.6250	0.6250	4.4444	1.0000	16.6250	16.6250	14.7778	4.8000	2.3750	7.1250	6.3333	2.3333
MILP Obj	0.6250	0.6250	4.4444	1.0000	16.6250	16.6250	14.7778	4.8000	2.3750	7.1250	6.3333	2.3333
Error (%)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Metric	1x16x8				1x32x8				1x64x8			
	Base	Fail 1	Fail 2	Fail 3	Base	Fail 1	Fail 2	Fail 3	Base	Fail 1	Fail 2	Fail 3
LP Bound	0.3750	3.3750	3.0000	1.2205	1.6250	1.6250	1.4531	0.5608	0.1250	0.3750	0.6719	0.1722
Rounded Obj	0.3750	3.3750	3.0000	1.1915	1.6250	1.6250	1.2500	0.5294	0.1250	0.3750	0.5625	0.1613
MILP Obj	0.3750	3.3750	3.0000	1.2174	1.6250	1.6250	-*	-*	0.1250	0.3750	-*	-*
Error (%)	0.00	0.00	0.00	2.13	0.00	0.00	-	-	0.00	0.00	-	-

Table 5. Single-Segment Experiments Error Results (*not all MILP results completed due to out-of-memory).

G Compact Model and Derivation of the Extensive ALLToALL Formulation

This appendix derives the extensive formulation (1)–(4) by stating a compact, arc-based multi-commodity flow MILP (Appendix G.2), relaxing it to a max–min concurrent-flow LP (Appendix G.3), and applying a Dantzig–Wolfe decomposition to recover the master problem and pricing subproblem of Section 4.2 (Appendix G.4). We reuse all notation defined in Section 4.2.

G.1 Capacities and commodities

We extend $\tilde{G}$ with one *super-sink* σ_d per destination GPU $d \in V$, reached through *exit* arcs $((d, t), \sigma_d)$ that absorb a chunk once it has reached its destination d at some step $t < m_s$. For $v_i \in \tilde{V}$ we write $\delta^+(v_i)$ and $\delta^-(v_i)$ for its outgoing and incoming arcs.

Commodities and chunk budget – In an ALLToALL collective, each tensor $c \in C$ has source $src(c)$ and destinations $dst(c) = V \setminus \{src(c)\}$. We route at the granularity of ordered GPU pairs, i.e., commodities

$$\mathcal{K} = \{k = (s, d) : s \in V, d \in dst(s)\},$$

where commodity $k = (s, d)$ delivers $K \in \mathbb{Z}_{>0}$ chunks from the source node s_0 to its super-sink σ_d . The commodities sharing a common source $s = \text{src}(c)$ together form the flow of tensor c ; this grouping is what the decomposition of Appendix G.4 exploits.

G.2 Compact mixed-integer model

Let $f_{a_i}^k \in \{0, \dots, K\}$ be the integer number of chunks of commodity $k \in \mathcal{K}$ routed on arc $a_i \in \tilde{A}$, and let $\chi \geq 0$ be a global *congestion* multiplier. The compact model minimises congestion subject to delivering K chunks per commodity, with every link load held below $c_a \chi$:

$$\min \quad \chi \quad (11)$$

$$\sum_{a_i \in \tilde{\delta}^-(v_i)} f_{a_i}^k - \sum_{a_i \in \tilde{\delta}^+(v_i)} f_{a_i}^k = b_{v_i}^k \quad \forall k \in \mathcal{K}, \forall v_i \in \tilde{V}, \quad (12)$$

$$\sum_{k \in \mathcal{K}} \sum_{a_i \in \tilde{A}_a} f_{a_i}^k \leq c_a \chi \quad \forall a \in A, \quad (13)$$

$$f_{a_i}^k \in \{0, \dots, K\}, \quad \chi \geq 0 \quad \forall k \in \mathcal{K}, \forall a_i \in \tilde{A}, \quad (14)$$

where the supply/demand vector encodes the per-commodity demand $b_{v_i}^k = K(1[v_i=\sigma_d] - 1[v_i=s_0])$. Constraints (12) are per-commodity flow conservation on $\tilde{G}$, and constraints (13) make the maximum normalised link load equal to χ by summing every step-indexed copy of a link against its capacity c_a ; the step budget enters implicitly through the truncation of $\tilde{A}$ at m_s . Since every commodity routes exactly K chunks, the delivered throughput at optimum is K/χ^* , where χ^* is the optimal congestion. The overall throughput is recovered by scanning the chunk budget

$$z_{\max}^{\text{MILP}} = \max_{K \in \mathbb{Z}_{>0}} \frac{K}{\chi^*(K)}. \quad (15)$$

G.3 Linear relaxation

Relaxing the integrality requirement (14) to $f_{a_i}^k \geq 0$ makes the chunk budget K redundant: the model is scale-invariant in K , so we fix $K = 1$ and let each commodity deliver a common rate $z = 1/\chi$ to its sink. Writing the relaxation directly on the flow-conservation constraints yields the max-min concurrent-flow LP, denoted (CF-LP):

$$\max \quad z \quad (16)$$

$$\sum_{a_i \in \tilde{\delta}^-(v_i)} f_{a_i}^{(s,d)} - \sum_{a_i \in \tilde{\delta}^+(v_i)} f_{a_i}^{(s,d)} = b_{v_i}^{(s,d)}(z) \quad \forall (s,d) \in \mathcal{K}, \forall v_i \in \tilde{V}, \quad (17)$$

$$\sum_{(s,d) \in \mathcal{K}} \sum_{a_i \in \tilde{A}_a} f_{a_i}^{(s,d)} \leq c_a \quad \forall a \in A, \quad (18)$$

$$f_{a_i}^{(s,d)} \geq 0 \quad \forall (s,d) \in \mathcal{K}, \forall a_i \in \tilde{A}, \quad (19)$$

where the (now rate-parameterised) demand vector is $b_{s_0}^{(s,d)} = -z$, $b_{\sigma_d}^{(s,d)} = +z$, and 0 otherwise. Constraints (17) are the per-commodity flow conservation, which tie the rate routed to each sink to the common objective z ; constraints (18) are the (coupling) link-capacity bounds; and (19) is non-negativity. Maximising z therefore maximises the minimum per-commodity throughput.

G.4 Dantzig-Wolfe decomposition recovers the master

We apply Dantzig-Wolfe decomposition to (CF-LP) tensor by tensor, grouping the commodities that share a common source $s = \text{src}(c)$ into one block per tensor c . The coupling link-capacity rows (18) are kept in the master, while the conservation rows (17) define the blocks. Every block flow is a non-negative combination of source-rooted trees $T_c \in \mathcal{T}_c$: choosing one $\text{src}(c) \rightarrow d$ route for each destination $d \in \text{dst}(c)$ yields one such tree, with $\beta_a^{T_c}$ counting the paths of T_c that load link a . Substituting the structure weights $x_{T_c} \geq 0$ for the arc flows eliminates the conservation rows and reproduces, line for line, the extensive master and its pricing subproblem; we therefore solve the Restricted Master Problem and price columns exactly as described in Section 4.2.

Received 5 June 2026; accepted 10 September 2026