MCCL: Measurement-based Collective Communication Library

ANONYMOUS AUTHOR(S)

Collective communication is a critical performance bottleneck in large-scale distributed training. Existing libraries and synthesis frameworks rely on static schedules optimized for peak network conditions. In production environments, however, network performance is highly dynamic due to gray failures, congestion, and resource contention. This causes the performance of pre-computed static schedules to degrade significantly during execution. Instead of assuming static topology and peak link capacities, Measurement-based Collective Communication Library (MCCL) leverages passive GPU-to-GPU measurements to capture real-time bandwidth variations. It feeds this telemetry into a continuous optimization loop, dynamically adapting dominant data-transfer schedules without interrupting the training process. Using a multi-node hardware prototype to validate the end-to-end system and large-scale simulations to explore cluster-scale scenarios, we demonstrate that MCCL outperforms state-of-the-art static synthesis approaches and vendor-provided collectives under realistic failure scenarios. Our results show that MCCL incurs negligible ($< 2\%$) overhead while maintaining high measurement accuracy ($< 1.2\%$ error), low schedule hot-swap time (≈ 12 ms), and achieves up to $4.99\times$ training time reduction on state-of-the art LLMs workloads.

1 Introduction

Large-scale distributed training has become the foundation of modern Deep Learning (DL), powering state-of-the-art models in natural language processing, computer vision, and recommendation systems. The performance and efficiency of distributed training increasingly depend on the effectiveness of collective communications, particularly as model sizes and dataset scales continue to grow. Operations such as AllGather, AllToAll, AllReduce, and ReduceScatter tend to highly influence training time, often accounting for a substantial fraction of the overall iteration latency. As a consequence, the last few years have seen growing efforts dedicated to optimizing collective communication, focusing on the challenges of pure data transfer and reduction arithmetic.

Hardware accelerator vendors provide Collective Communication Library (CCL) such as Nvidia CCL (NCCL) [25], AMD ROCm CCL (RCCL) [32], Huawei CCL (HCCL) [14], and Intel OneCCL [28], with highly optimized collective communication implementations tailored to common cluster topologies and hardware configurations. These vendor libraries increasingly leverage specialized hardware, such as reduction engines [27] or in-network computing [34], to accelerate Reduce-intensive collectives. To complement this, schedule optimizers [3, 5, 22, 31, 33, 38, 43] have made significant progress in improving critical data-transfer intensive collective operations such as AllGather and AllToAll. However, these state-of-the-art schedule synthesizers share a fundamental limitation: they operate in an "open-loop" manner, relying on static topology assumptions and peak link capacities that rarely hold in production environments.

Datacenter networks typically exhibit complex and dynamic behavior resulting from link contention, background traffic, and transient performance anomalies. A critical challenge is the prevalence of *gray failures* [13, 16], where communication remains possible, but performance degrades due to thermal throttling, PCIe or Network Interface Cards (NIC) errors, optical degradation, etc. Unlike *hard failures*, which typically trigger alarms, gray failures can silently reduce link performance, sometimes substantially, for several minutes or even hours without triggering a hard failure [13, 41]. As a consequence, collective communication schedules that are optimal at deployment time may become increasingly inefficient as system conditions evolve, significantly slowing down training processes and leading to suboptimal resource utilization across the distributed infrastructure. This mismatch is particularly pronounced in iterative training as the same collectives repeat for thousands of iterations, thus performance degradation becomes persistent and cumulative.

While adaptive schedule re-synthesis represents an intuitive solution to such degradation, current synthesis frameworks face two fundamental limitations that prevent online adaptation. First, while recent tree-packing solvers like ForestColl [43] for AllGather, and AllReduce are fast, they lack mechanisms to reuse previous schedule structures. By optimizing from a cold start, they discard valid sub-trees when a localized failure occurs, forcing redundant exploration and wasting precious time. Second, for other operations such as AllToAll, current solvers [5, 22] rely on heavy Mixed-Integer Linear Programming (MILP) or symmetry assumptions that do not apply to clusters with gray failures. These solvers take hours to converge, making them inadequate for online adaptation.

This paper is motivated by the observation that the repetitive nature of distributed training allows for the continuous, passive instrumentation of the underlying GPU-to-GPU data paths. Rather than relying on the aggregate completion time of a collective, which can mask the specific location of a bottleneck, we leverage real-time, GPU-GPU performance telemetry to identify performance drift at the path level. To this end, we present Measurement-based Collective Communication Library (MCCL), which complements the traditional synthesis paradigm by integrating a continuous optimization loop directly into the communication stack. MCCL employs a controller-executor architecture that continuously monitors GPU-to-GPU communication performance, identifies collectives affected by runtime degradation, selectively triggers online schedule re-synthesis, and safely deploys optimized schedules during training. This design combines lightweight passive telemetry, online optimization, and safe schedule hot-swapping, enabling collective communication to track evolving network conditions throughout the training process. Crucially, MCCL targets bandwidth-heavy collective communication, such as AllGather, AllToAll. Reduction-centric collectives such as ReduceScatter continue to use existing highly optimized implementations, avoiding interference with specialized hardware accelerators or in-network reduction mechanisms. Consequently, AllReduce remains compatible with MCCL through its standard decomposition into ReduceScatter and AllGather, allowing AllGather optimization while leaving the reduction phase unchanged. In summary, this work makes the following contributions:

- We introduce MCCL, a system that continuously adapts collective communications schedules based on passive run-time measurements of GPU-to-GPU performance.
- We propose novel online optimization strategies that balance throughput optimality with rapid solvability: a decomposition-based Column Generation (CG) approach for AllToAll and a warm-start version of ForestColl for AllGather.
- We present a full-fledged MCCL prototype including a lightweight, passive measurement framework that captures high-fidelity bandwidth metrics at the thread-block level with minimal overhead ($< 2\%$), and a fast schedule swap mechanism ($\approx$12 ms).
- We demonstrate that MCCL mitigates performance degradation caused by partial failures and transient network anomalies, maintaining high performance under dynamic conditions.

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 landscape [40] has shifted from vendor-optimized libraries to automated synthesis frameworks. Traditional libraries provided by hardware accelerator vendors—like NCCL [25], RCCL [32], HCCL [14], and OneCCL [28], rely on rigid, 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 NVLink

Name	Synthesis	Optimization	Re-optimization
NCCL[25], RCCL[32]	Static	N.A.	×
SCCL[3]	Offline	Static, peak topology	Hard
TACCL[33]	Offline	Static, peak topology	Hard
TECCL[22]	Offline	Static, peak topology	Hard
SyCCL[5]	Offline	Static, peak topology	Hard with asymmetry
ForestColl[43]	Offline	Static, peak topology	AllGather no AllToAll
MCCL	Online	Dynamic, measured throughput	Possible

Table 1. Collective Communication Algorithms.

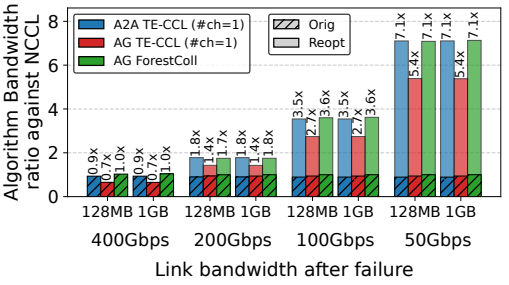


Fig. 1. CCL scheduling algorithm performances.

reduction engines [27] and SHARP in-network computing [34], the benefits of such accelerators are largely confined to collectives that involve reduction operations. Pure communication collectives, such as AllGather and AllToAll, remain primarily constrained by data movement and often cannot fully exploit heterogeneous or non-standard interconnects. Optimizing these bandwidth-bound collectives is therefore critical to reducing communication overhead and, ultimately, training time. To enable this flexibility, platforms such as MSCCL [24] were developed to execute custom, topology-aware collective algorithms on top of standard libraries such as NCCL. Building on top of this, several collective communication algorithm synthesizers have been developed: SCCL [3], which synthesizes Pareto-optimal (latency vs. bandwidth) algorithms using Satisfiability Modulo Theories (SMT) solvers; TACCL [33], which utilizes user-provided "communication sketches" to provide topological information to guide the automatic synthesis of collective algorithms; TECCL [22], which adopts a traffic-engineering-based approach using a Mixed Integer Linear Problem (MILP) formulation and A^* technique for scaling; SyCCL [5], which leverages collective and topology symmetries to decompose the scheduling problem into smaller sub-problems for scalable and efficient synthesis; ForestColl [43], which generates throughput-optimal collective schedules by constructing spanning tree packs using scalable, strongly polynomial-time algorithms. A common denominator across these approaches (cf. Table 1) is the "static world" assumption: they optimize for a topology snapshot that is assumed to remain constant throughout training.

2.2 Challenges: Scale and Reliability

Training foundational models like LLama3 [1], Gemini 2.5 [11] 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, etc.), which might be considered rare exceptions in isolation, become a near certainty and pose a critical operational challenge [30]. For instance, estimates suggest that even highly reliable NICs with a mean time to failure of 5 years would likely cause the first job failure in less than half an hour within a 100k GPU cluster [9]. 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) [13, 16], where components remain functional but operate at degraded performance levels thanks to specific mechanisms [2, 29, 37]. Notably, 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 [41]), while hardware-related faults can last significantly longer (e.g., FailSlow [13]). Because these degraded states persist for several training iterations, they turn into 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 illustrate the gains of dynamic re-synthesis, we simulated AllGather and AllToAll, the two most critical data-intensive collectives, using an extended version of SimAI [35, 39] that supports custom CCL execution.[1] We simulated a 2-host (16 GPUs) rail-optimized topology (cf. Appendix A, Figure 12 for reference), using NCCL, TECCL (1 chunk), and ForestColl as representatives of CCL's state-of-the-art. As shown in Figure 1, under healthy conditions—i.e., all links connecting the nodes to the switch have 400 Gbps capacity—TECCL and ForestColl match NCCL Ring's Algorithm Bandwidth [26], which is already throughput optimal. However, when simulating a gray failure by degrading the link capacity of GPU 0 by 50–75-88%, 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, they fail to account for the end-to-end complexity of the communication path. Indeed, traditional network-level measurements [8, 23, 41] overlook critical intra-node factors, such as PCIe bottlenecks and protocol overheads. Consequently, MCCL must explicitly instrument the actual GPU-to-GPU data paths. While recent frameworks like MCCS [42] attempt runtime adaptability, their per-job adaptation is limited to re-ordering ranks within a fixed ring algorithm, lacking the fidelity to detect specific path degradations or to synthesize new schedules online.

Even if detection is solved, existing CCL fail to meet the requirements for online adaptation due to fundamental limitations. While solvers like ForestColl [43] are computationally fast enough to re-run online (seconds), they operate statelessly. When a localized gray failure occurs, they cannot 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. Consequently, finding an optimal, asymmetry-aware AllToAll schedule for a large cluster remains computationally prohibitive for online adaptation.

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, passive network measurements, and schedule hot-swapping.

[1] The simulator repository is omitted for anonymity and will be released upon acceptance.

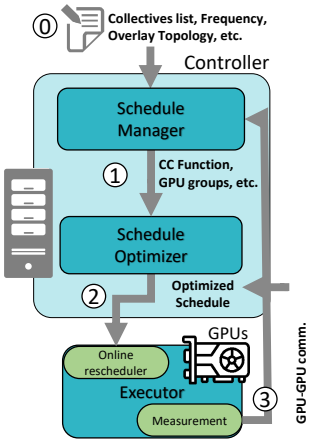


Fig. 2. MCCL System architecture.

Algorithm 1. Re-optimization selection algorithm.

Require: Graph $G(V, A)$, Measures M, Schedules $\mathcal{S}$, Cut $\mathcal{K}$, Threshold Γ, Base Slack $base\_slack_a$
Definitions: C: Set of all commodities
$C(s)$: The routing of commodities under schedule s
$C_a(s)$: Set of paths crossing link a under schedule s
$x_{min}(s)$: The bottleneck data rate for schedule s

for each $s \in \mathcal{S}$ **do** ▷ — For each collectives current schedule —
  $x_{min}(s) \leftarrow \infty$ ▷ **Phase 1:** Find Bottleneck Rate for schedule s
  for each link $a \in A$ **do** ▷ Iterate all links in Topology
    $n_a \leftarrow |C_a(s)|$
    if $n_a > 0$ **then**
      $x_{curr} \leftarrow M_a / n_a$
      $x_{min}(s) \leftarrow \min(x_{min}(s), x_{curr})$ ▷ Update global bottleneck
  $Slack_{max} \leftarrow 0$ ▷ **Phase 2:** Evaluate Slack on the Cut
  for each link $a \in \mathcal{K}$ **do** ▷ Iterate only links in the Cut
    $n_a \leftarrow |C_a(s)|$
    $u_a \leftarrow n_a \times x_{min}(s)$
    $current\_slack \leftarrow (M_a - u_a)/M_a$
    $\delta \leftarrow |current\_slack - base\_slack_a|$ ▷ Compensation for approximation
    $Slack_{max} \leftarrow \max(Slack_{max}, \delta)$
  if $Slack_{max} > \Gamma$ **then** ▷ **Phase 3:** Decision
    ScheduleOptimizer(C) ▷ Re-evaluate all commodities to find a new schedule

By design, MCCL targets bandwidth-intensive, data-transfer operations (AllGather, AllToAll). However, modern distributed training paradigms, such as Fully Sharded Data Parallelism (FSDP), heavily rely on operations like AllReduce. MCCL natively supports these workloads by collectives decomposition. When a training application dispatches an AllReduce, MCCL splits it into its constituent phases. The initial arithmetic-intensive ReduceScatter phase is executed using static, precomputed schedules to fully leverage specialized hardware reduction engines [27] or in-network aggregation [34]. The subsequent bandwidth-intensive AllGather phase is routed through MCCL's Executor for dynamic optimization. This split architecture ensures maximum arithmetic performance while completely shielding the workload from data-transfer bottlenecks.

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 ⓪, a profile of the training's collective calls (e.g., list, frequency, and chunk size) and an initial overlay topology are fed to the *Schedule Manager*. Such topology models intra-node (i.e., GPUs within the same node) and inter-node communication as direct overlay links, using bandwidth and latency obtained via initial probing, as in [6, 22, 33, 43]. 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 GPU-GPU 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 health of GPU-GPU communication paths actively. 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 measurement feedback 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 on the fly. This continuous loop ensures that collective schedules evolve in lockstep with the true state of the underlying network. The remainder of this section provides a high-level overview of the components depicted in Figure 2.

3.2 Controller Components

The Controller logic is orchestrated by the *Schedule Manager* and the *Schedule Optimizer* 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 resource utilization. It inputs the application profile (step ⓪) stored in the graph $G(V, A)$, the set of collectives $\mathcal{S}$ alongside the details of their active execution schedules, and the throughput measurements (step ③), $\mathcal{M}$, smoothed over a sliding window. Measurements are stored in the Controller in a dedicated per-node memory region written via RDMA transfers.

As described in Algorithm 1, the Schedule Manager continuously iterates over the set of active collectives $\mathcal{S}$ to diagnose the health of communication paths. Instead of relying on execution time estimates, the algorithm employs a flow-based diagnostic. For each collective $s \in \mathcal{S}$, it first computes the bottleneck throughput $x_{min}(s)$ of the current schedule by iterating over all links in the topology. This value represents the maximum theoretical per-tensor throughput supported by the most congested link (defined as measured capacity M_a divided by the number of active tensors n_a on link a). The set of these bottleneck links forms the cut $\mathcal{K}$. Using this global bottleneck, the Schedule Manager evaluates the *slack*, i.e., residual non-occupied bandwidth, across $\mathcal{K}$. For each link in this set, it calculates the expected utilization $u_a = n_a \times x_{min}(s)$ and derives the current slack as the remaining capacity $(M_a - u_a)/M_a$. If the maximum observed slack exceeds a configurable threshold Γ, it implies the current schedule is under-utilizing the available bandwidth in the bottleneck. The Schedule Manager then flags the collective for re-optimization, and invokes the Schedule Optimizer with the candidate set C_{re-opt}.

Schedule Optimizer. The optimizer synchronously adapts its optimization strategy based on the collective operation. To ensure broad applicability while maintaining solvability, the optimizer focuses on the two fundamental primitives from which other collective operations are derived: AllToAll, and AllGather. For AllToAll operations in which asymmetric gray failures render traditional symmetry-based solvers ineffective, the optimizer implements a CG approach. This enables fast re-synthesis by supporting warm starts from previous solutions and allowing for time-bounded execution. The CG solver optimizes throughput to ensure robustness against network irregularities. For AllGather operations, the optimizer extends the ForestColl tree-packing approach [43]. It incorporates a guided Breadth-First Search (BFS) to support warm starts, preserving the structure of previous schedules and minimizing recomputation overhead. The solver 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, 24] backend to inject custom schedules, it consists of two elements:

Online Rescheduler. This module is responsible for the dynamic deployment of new schedules. When the Controller provides a new Optimized Schedule (step ②), the Online Rescheduler stages it. MCCL leverages a runtime "hot-swap" mechanism inspired by MCCS [42]. In essence, when the Online Rescheduler receives the new schedule, it initiates an AllGather through which GPUs involved in the schedule exchange the sequence number of their last-launched collective. All nodes then compute the maximum sequence number which serves as a barrier, indicating which collectives must be completed before using the new schedule.

Measurement. This module provides real-time telemetry that feeds the optimization loop (step ③). To ensure the feedback loop relies on measurements capturing only network communications, the telemetry collection is restricted to operations without reduction phases, thereby 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 (e.g., NVLink), it measures the cumulative per-GPU throughput, aggregating bandwidth across all concurrent channels to capture 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 RDMA[2], where they are recorded into the measurement matrix 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. State-of-the-art approaches like TECCL [22] 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 of several hours, rendering them impractical for online schedule re-computation. Conversely, solvers like ForestColl are significantly faster at generating throughput-optimal schedules, but they operate statelessly. When a localized gray failure or topology change occurs, these algorithms discard the entire communication structure and rebuild the schedule from scratch (i.e., cold start). This forces a blind, redundant exploration of previously solved sub-problems, unnecessarily wasting critical computational cycles and delaying runtime adaptation.

The core principle underlying MCCL's optimization framework is that network anomalies typically disrupt only a marginal 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 minimizes redundant exploration. As our evaluations demonstrate, utilizing historical schedules to anchor the search reduces convergence times as network size increases, rendering continuous, real-time schedule adaptation a practical reality. To achieve this, we introduce two optimization strategies tailored to the communication structures of AllGather and AllToAll operations:

Throughput-Optimal AllToAll via Column Generation (CG). We propose a throughput-optimal, decomposition-based formulation for AllToAll collectives that eliminates the solving-time overhead associated with traditional time-expanded graphs. By decoupling the problem into a Restricted Master Problem (RMP) and a Pricing sub-problem, we avoid constructing the full, exponentially large compact model while preserving LP throughput optimality. Building on this scalable foundation, we also introduce a warm-start mechanism that retains valid routing structures from previous iterations, we drastically reduce the optimization iterations required to adapt to network changes.

Throughput-Optimal AllGather via ForestColl-warm. We introduce ForestColl-warm, which advances the baseline tree-packing heuristic by incorporating an A^*-guided exploration strategy tailored for online re-computation. By leveraging the solution history as prior knowledge to direct the underlying 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.

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

4.2 AllToAll optimization

To satisfy the strict latency requirements of online re-synthesis, we model the AllToAll routing 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 tensor $c \in C$ source. 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:

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

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

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

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

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 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; these duals parameterize the pricing problem.

This formulation is the Dantzig–Wolfe reformulation of a compact, arc-based multi-commodity flow model; the compact model and its full derivation are provided in Appendix G. Directly solving this reformulation is computationally prohibitive due to its exponential number of variables. To overcome this challenge, we leverage a CG approach that generates columns on the fly. The CG algorithm alternates between solving the RMP, limiting the optimization to a subset of variables, and solving a pricing problem for each tensor. The pricing problem 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 $\sum_{a \in T_c} \beta_a^{T_c} \alpha_a < \alpha_c$.

To account for the sequential nature of tensor transfers within the CG framework, the pricing problem's shortest-path tree search is executed over a time-expanded graph $\tilde{G} = (\tilde{V}, \tilde{A})$. In this context, a *step* represents the discrete transfer of a data chunk between two 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 duplicated 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 incoming link (u, v) to a switch, we add instantaneous links (u_i, v_i) for $i \in \{0, \ldots, m_s - 1\}$; for all other physical links (u, v), we add forward-stepping links (u_i, v_{i+1}) for $i \in \{0, \ldots, m_s - 2\}$. 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 epochs 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 is chosen to guarantee maximum throughput with the lowest possible solver overhead, the formulation can optionally minimize m_s via a binary search if tighter latency bounds are desired (Appendix D). During the pricing iteration, if the cost of the optimal shortest-path tree in $\tilde{G}$ is strictly less than α_c, the corresponding column is generated and added to the Restricted Master Problem (RMP).

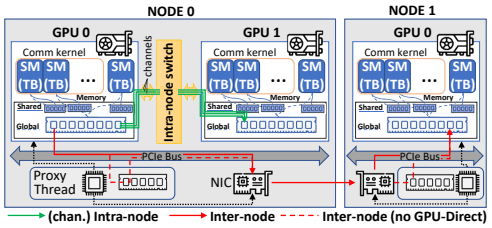


Fig. 3. GPU Communication and Memory Layout.

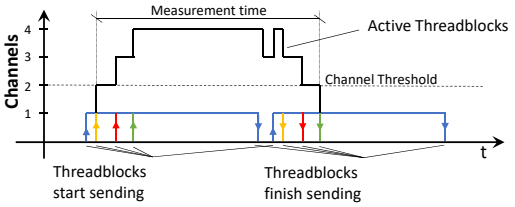


Fig. 4. Per channel (TB) measurements process.

Finally, once the CG algorithm converges, we apply a rounding phase to extract a feasible schedule that uses at most ten chunks. For this, we adopt the deep-dive approach proposed in [17] (cf. Section 6.1).

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 — The objective of this phase is to replace physical links with logical links. The procedure determines the maximum feasible flow for each link, constrained by the previously calculated maximum throughput, and assigns this capacity to the logical link in a greedy manner.
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 standard, 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 enhance the re-optimization strategy, our improved 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 minimizes 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 (cf. Section 6.1).

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 Collectives measurements

Before introducing our lightweight passive measurement architecture, we first analyze the execution model of the underlying communication library. Without loss of generality, we focus on NCCL, but note that other software stacks for GPU-GPU communication share a similar architecture.

As detailed in recent literature [15], 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:
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.
Measurement 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.

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 Measurement 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.

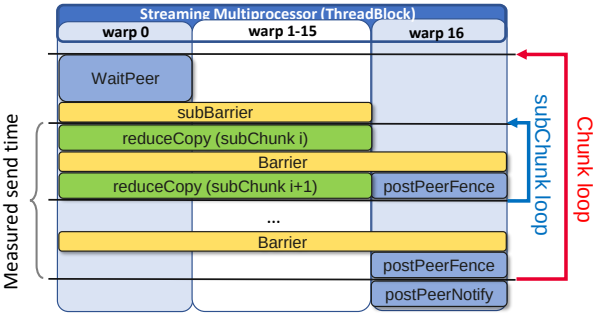


Fig. 6. NCCL communication kernel schema.

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 Measurement Cache.[3]

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 i.e., receive. 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` (cf. Figure 6), which signals completion. This separation allows us to reuse the dedicated signaling warp (16) as a concurrent progress bookkeeper, feeding the MCCL measurement data structures. By executing `postPeerFence` in parallel with the `reduceCopy`[4] loop (warps 1–15), we enforce system-level memory fences on sub-block transfers without blocking the primary data movers. This effectively masks synchronization and atomic operations. Our evaluation shows that the overhead remains around 2% when the TB allocation is sufficient to saturate the available bandwidth, which is required to achieve throughput-optimal schedules.

5.2 Inter-node measurements

For communication across servers, 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 serialized execution significantly simplifies the measurement logic and effectively eliminates the need for atomic operations. Despite these architectural differences, we reuse the same *Measurement Accumulator* and *Measurement 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.

[3]Note 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.

[4]In NCCL kernel, the `reduceCopy` symbol is immutable; for non-reduction operations targeted by MCCL, it does not incur arithmetic overhead.

The measurement logic is embedded within the Proxy's Finite State Machine (FSM), which manages the life-cycle of network chunks through three primary stages: *Posted* (a transfer slot is available), *Transmitted* (the network operation, e.g., `isend`, has been triggered), and *Done* (transmission is confirmed complete). To calculate effective bandwidth, we isolate the network injection and transit time by measuring the interval between the *Transmitted* (T) and *Done* (D) 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), the PCIe bus, or the network link.

6 Experimental Results

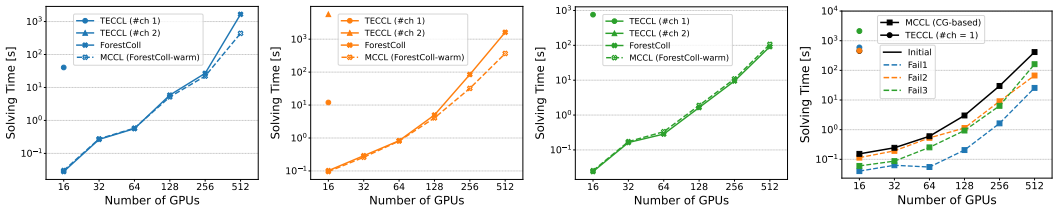
We validate MCCL at three different levels. First, we evaluate our optimization algorithm against state-of-the-art competitors in terms of solving and re-solving time (cold vs. warm start), and schedule performance when optimizing a single collective in offline settings. Second, we test a prototype implementation of MCCL to evaluate the overhead and accuracy of the measurement module, as well as the overhead of the online rescheduling mechanism. Finally, we run large-scale simulations of real-life training workloads and 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 [36], and our C++ re-implementation of the released ForestColl [10]. To evaluate schedule performance, we run simulations on a modified SimAI [35, 39]. We consider HPN rail optimized topologies [30], 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 µs 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×H×N` a topology of `S` segments, `H` hosts per segment, and `N` GPUs per host.

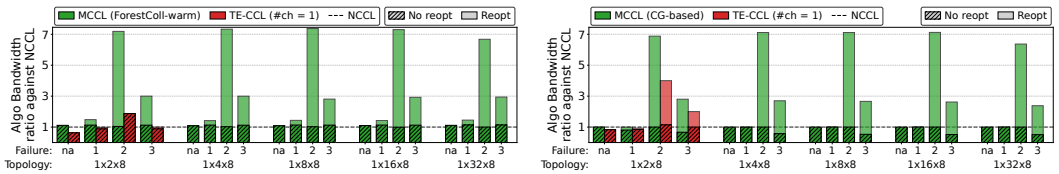
Failures. We define three different failures to evaluate our approach under diverse conditions. *Failure 1* (Fail1) affects the NVSwitch of the first host; we simulate that 13 of the 18 lanes connecting GPU 0 fail, and the intra-node bandwidth drops from 2880 Gbps to 800 Gbps. In *Failure 2* (Fail2), we reduce the capacity of the link connecting GPU 0 to its ToR switch from 400 Gbps to 50 Gbps. *Failure 3* (Fail3) simulates a fault of GPU 0, which is replaced by a backup GPU located in a separate rack with a dedicated 150 Gbps bandwidth and 1 µs additional latency from ToR switches.

Solving Time. We evaluate the scalability of MCCL's solver against the state-of-the-art static approaches TECCL [22] and vanilla ForestColl [43] by 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, MILP-based baselines like TECCL prove computationally intractable even for small topologies (up to $\mathbf{10^3}$s for a 16-GPUs collective with 1 chunk). In contrast, ForestColl and our CG-based algorithm scale significantly better, successfully computing schedules for collectives of up to 512-GPUs. Regarding AllGather collectives, we note that our version of ForestColl enhanced with warm-start is up to **2.58**× and **2.80**× faster than the base algorithm for Failure 1 and 2, for which the underlying graph structure is highly reusable. Warm-start yields instead no noticeable improvement for Failure 3, as this failure is significantly more disruptive than the first two and a great part of the graph need to be recomputed from scratch. Finally, Figure 7d



(a) AllGather, Fail 1. (b) AllGather, Fail 2. (c) AllGather, Fail 3. (d) AllToAll, Fails 1-3.

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



(a) AllGather performance against NCCL. (b) AllToAll performance against NCCL.

Fig. 8. Schedule performance for AllToAll (a), and AllGather (b).

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**× faster) and provides solutions within tens of seconds for 512-GPUs collectives, making continuous adaptation feasible even for AllToAll collectives.

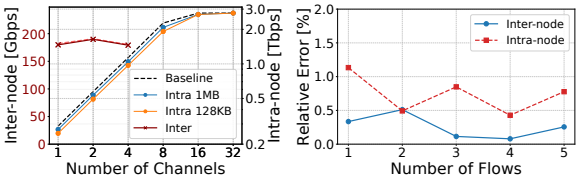
Schedule Performance. To quantify the throughput gains offered by MCCL's dynamic algorithms, we conducted offline simulations using an extended SimAI [35] version[5] that supports custom CCL execution, with the NS-3 network backend, on the rail-optimized topology described above. Using a 128 MB tensor size per source, we measured the *Algorithm Bandwidth* [26] speedup relative to the NCCL baseline. As detailed in Section 4.2, a rounding phase is required for the AllToAll collective; in our experiments, this rounding introduces a maximum error of 2.13% with respect to the optimal. Detailed results on the rounding error are reported in Appendix F.

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 in Figure 8 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 (≈**1.0**×). In contrast, MCCL detects and adapts effectively to varying link capacities. For AllGather (cf. Figure 8a), MCCL achieves speedups of up to **7.33**× over the baseline in the presence of gray failures. For AllToAll (cf. Figure 8b), MCCL significantly outperforms static solvers with speedups of up to **7.12**×. It is important to note that MCCL optimizes for computational efficiency. As detailed in Section 3, the system employs a cut-based filtering mechanism. 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. 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 top of MSCCL [24]. The prototype was deployed on a small cluster comprising two hosts, each equipped with four Nvidia H200 GPUs connected in a rail-optimized topology with Nvidia ConnectX-7 NICs (200 Gbps) using GPUDirect (details in Table 3).

[5]The simulator repository is omitted for anonymity and will be released upon acceptance.



(a) Throughput overhead. (b) Measurement error.

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.

6.2.1 Measurement Module. We first evaluate the feasibility and efficiency of the MCCL measurement module. Our evaluation focuses on two sets of key metrics: (1) the *throughput* and *memory overhead*; and (2) the *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 ≈**13**% for low channel counts (1–2) and small chunks (128KB) due to granular synchronization i.e., `postPeerFence`, and atomic operations. However, this drops to ≈**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.

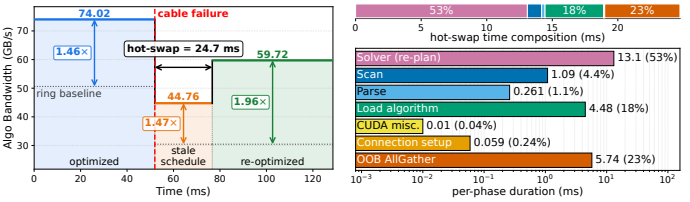
To preserve capacity for model parameters, MCCL minimizes *memory overhead* using the three-tiered hierarchy from Section 5.1. The total memory footprint comprises three distinct data structures: the Accumulator in GPU Global Memory (M_{acc}), the Cache in GPU Shared Memory (M_{cache}), and the Buffer in Host-Mapped Memory (M_{buf}). The single-entry sizes for these structures are S_{acc} = 32 B, S_{cache} = 24 B, and S_{buf} = 32 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 overhead remains negligible, particularly given the low concurrency of collectives during training.

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 [24]. 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. To synchronize the cluster state, the background thread executes an Out-of-Band (OOB) AllGather—a mechanism adapted from MCCS [42]—to retrieve the ID of the latest scheduled collective from each GPU in the group. The system then computes the maximum ID value across the cluster and instructs all GPUs seamlessly to transition to the new schedule starting from such collective.

Servers	2 hosts × 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 ×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.



(a) Algo BW before/after failure. (b) Breakdown of each phase.

Fig. 10. Prototype evaluation of the hot-swap mechanism in MCCL.

Figure 10a illustrates the impact of hot swapping on collective performance during a failure on the inter-node link (rate halves to 100 Gbps). The failure temporarily reduces AllGather Algorithm bandwidth [26] from **74.0 GB/s** to **44.8 GB/s**. Once the new schedule is activated, bandwidth recovers to **59.7 GB/s**. The swap itself completes in just **24.7 ms** and no disruption is observed when switching to the new schedule. Figure 10b breaks down the hot-swap latency. More than half of the time is spent in the solver (**13.1 ms**), while the actual swap time takes **11.6 ms** (mainly OOB synchronization and algorithm loading, **5.74 ms** and **4.48 ms**, respectively). As the number of nodes scales up, the re-optimization time increasingly dominates the total swap phase (cf. Section 6.1), making the actual swap time negligible. Crucially, MCCL executes hot-swap operations on dedicated threads, remaining off the critical execution path. The only synchronous step is the OOB AllGather, which briefly pauses the CPU from issuing new collectives. This delay becomes visible only if the swap occurs when the GPU execution queue is empty. In practice, large-scale training workloads heavily pipeline collective communication and computation, allowing even this brief CPU-side pause to overlap with ongoing GPU execution.

6.3 End-to-End Validation

To validate the end-to-end benefits of our approach, we run large-scale experiments in SimAI, simulating the training of two LLMs (see below). We use an HPN topology composed of multiple 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. As a baseline, we compare against plain NCCL.[6]

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, 20]. 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 2048–32758 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 ≈**34%** of Llama's training time, and for ≈**65%** of DeepSeek's.[7][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 in both workloads, with peaks of **2.85**× and **4.99**× under failure 2 for Llama and DeepSeek, respectively. Finally, MCCL scales effectively with increasing GPU counts, yielding similar relative gains across different cluster sizes.

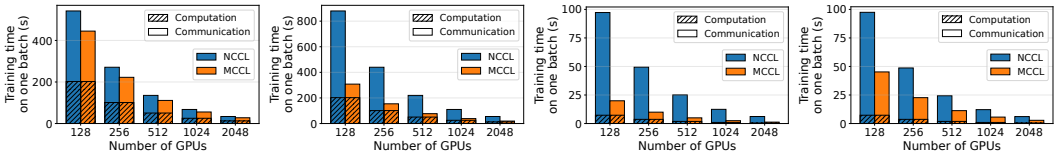
[6] We exclude TECCL due to unpractical solving time, and ForestColl as it cannot optimize AllToAll collectives, while it delivers same schedule quality as MCCL for AllGather collectives.

[7] In 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 solver overhead unjustified in practical settings.

[8] ReduceScatter collectives operate at the host level, as the simulator does not support in-network reduction.

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 404B and DeepSeek 671B (batch size 16K) collectives for one pipeline stage with 1024 GPUs.



(a) Llama405B, Fail 1. (b) Llama405B, Fail 2. (c) DeepSeek671B, Fail 2. (d) DeepSeek671B, Fail 3.

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

7 Related Work

A large body of work synthesizes collective schedules offline. TACCL [33], TE-CCL [22], and ForestColl [43] build topology-aware schedules from communication sketches, traffic-engineering formulations, and spanning trees, respectively, that outperform vendor-provided collectives. In contrast with MCCL, a fundamental limitation of these approaches is their rigid reliance on static, offline profiling conducted under idealized peak capacity conditions.

A second line of work adapts at runtime, but along axes orthogonal to ours. FAST [19] schedules MoE's skewed, dynamic `AllToAllV` online, rebalancing over the fast intra-server fabric so every NIC crosses the slow inter-server links with equal volume. Its adaptation tracks *workload* skew, not *network* state: modeling the cluster as two *uniform*, equal-capacity bandwidth tiers, its balancing breaks under link degradation, asymmetric capacity, or gray failures. At the *inter-job* level, Crux [4] and MCCS [42] instead arbitrate the network shared by co-located jobs. Crux prioritizes flows by "GPU intensity" (compute-to-communication ratio), routing GPU-heavy jobs to avoid contention on shared links, while MCCS, a provider-managed service, reorders ring ranks and pins flows to paths to preserve intra-jobs fairness and reduce congestion. Unlike MCCL, Crux and MCCS only perform macro-level re-optimization e.g., at job arrival/exit, or external triggers, leaving the collective schedules unchanged, making them exposed to gray failures.

A final line of work provides the execution backends. NCCL and RCCL ship primitives with static, built-in algorithms, while MSCCL [6], MSCCL++ [18], and ResCCL [21] add custom domain-specific language's 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-to-GPU telemetry directly into the optimization loop without disrupting the training workload. Our evaluation demonstrates that MCCL re-optimization algorithms can rapidly synthesize efficient schedules that adapt to detected bottlenecks, outperforming state-of-the-art baselines by up to **2.80**×. The MCCL prototype validates the passive measurement overhead (<**2**%), and accuracy (<**1.2**% error) and schedule swap-time (≈12 ms) feasibility. Finally, with end-to-end simulations demonstrate that MCCL can effectivly reduce training time up to **4.99**× in case of gray failures.

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. Syccl: Exploiting symmetry for efficient collective communication scheduling. In *Proceedings of the ACM SIGCOMM 2025 Conference*, SIGCOMM '25, page 645–662, New York, NY, USA, 2025. Association for Computing Machinery.

[6] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. Msccl: 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] 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.

[9] 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.

[10] Forestcoll github. https://github.com/liangyuRain/ForestColl, 2025.

[11] Comanici Gheorghe, Bieber Eric, Schaekermann Mike, and et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025.

[12] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981.

[13] 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.

[14] Huawei Collective Communication Library (HCCL). https://gitee.com/ascend/cann-hccl, 2026.

[15] Zhiyi Hu, Siyuan Shen, Tommaso Bonato, Sylvain Jeaugey, Cedell Alexander, Eric Spada, James Dinan, Jeff Hammond, and Torsten Hoefler. Demystifying NCCL: An In-depth Analysis of GPU Communication Protocols and Algorithms. Aug. 2025.

[16] 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.

[17] Nicolas Huin, Jérémie Leguay, Sébastien Martin, and Paolo Medagliani. Routing and slot allocation in 5g hard slicing. *Computer Communications*, 201:72–90, 2023.

[18] 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.

[19] Yiran Lei, Dongjoo Lee, Liangyu Zhao, Daniar Kurniawan, Chanmyeong Kim, Heetaek Jeong, Changsu Kim, Hyeonseong Choi, Liangcheng Yu, Arvind Krishnamurthy, Justine Sherry, and Eriko Nurvitadhi. Fast: An efficient scheduler for all-to-all gpu communication, 2026.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Microsoft collective communication library (msccl). https://github.com/microsoft/msccl, last updated 2023.

[25] Nvidia collective communication library. https://github.com/NVIDIA/nccl, 2026.

[26] Algorithm bandwidth. https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md#algorithm-bandwidth, 2026.

[27] Nvidia NVLink and NVLink Switch. https://www.nvidia.com/en-us/data-center/nvlink/.

[28] Intel OneAPI Collective Communication Library. https://github.com/uxlfoundation/oneCCL, 2026.

[29] 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.

[30] 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.

[31] 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.

[32] Amd rocm collective communication library. https://github.com/ROCm/rccl, 2026.

[33] 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.

[34] In-Network Computing with Nvidia SHARP. https://resources.nvidia.com/en-us-accelerated-networking-resource-library/network-computing-nvidia-sharp.

[35] Alibaba cloud - simai. https://github.com/aliyun/SimAI, last updated 2026.

[36] Teccl github. https://github.com/microsoft/TE-CCL, 2024.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

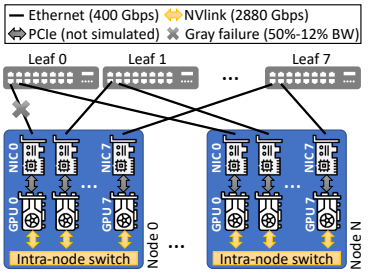


Fig. 12. Rail Optimized topology showing links between GPU nodes and leaf switches.

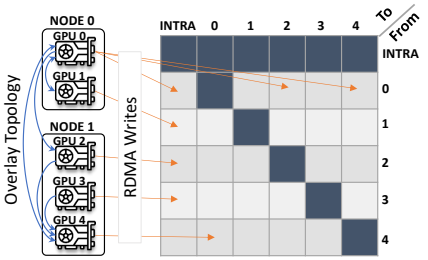


Fig. 13. Per collective type, size, and communicator group point-to-point measurements. Darker blocks indicate intra-node bandwidth, lighter blocks inter-node connections.

C Latency-Aware Switch Removal for AllGather

ForestColl switch removal [43] 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 w 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 $\sum_v y_{uv} \leq c_{uw}$,

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

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:

$$\max\; M\textstyle\sum_{a\in\bar{A}_i} y_a - \sum_{a=(u,v)\in\bar{A}_i} h_{uv} \quad (5)$$

$$h_{uv} \ge k\,y_a - w_{uv}\tfrac{k(k-1)}{2} \quad \forall a=(u,v)\in\bar{A}_i,\; k\in\{1,\dots,K\} \quad (6)$$

$$\textstyle\sum_{v:\,(u,v)\in\bar{A}_i} y_{uv} \le c_{uw} \quad \forall u\in\delta^-(w), \quad (7)$$

$$\textstyle\sum_{u:\,(u,v)\in\bar{A}_i} y_{uv} \le c_{wv} \quad \forall v\in\delta^+(w), \quad (8)$$

$$\textstyle\sum_{a\in A_{\bar{V}'}} y_a \ge MT - OF_{V'} \quad \forall V'\subseteq V, \quad (9)$$

$$y_a,\; h_{uv} \ge 0 \quad \forall a=(u,v)\in\bar{A}_i. \quad (10)$$

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 $A_{\bar{V}'} = A_{V'}\cap\bar{A}_i$ 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 [12], 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.

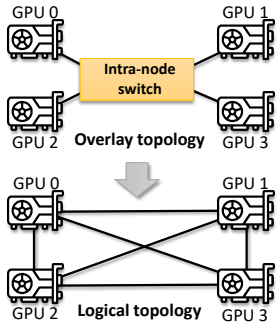


Fig. 14. Overlay to Logical topology.

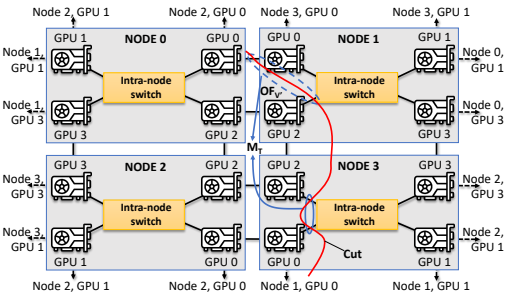


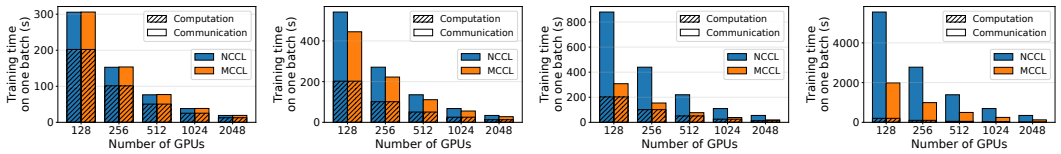
Fig. 15. Example cut induced by the max-flow in the switch removal phase, enforcing capacity constraints on the logical topology.

D Minimizing AllToAll Steps via Binary Search

The online loop fixes the step bound to $m_s = |V|-1$ for fast re-synthesis under transient 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 $m_s^* \le |V|-1$ 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.

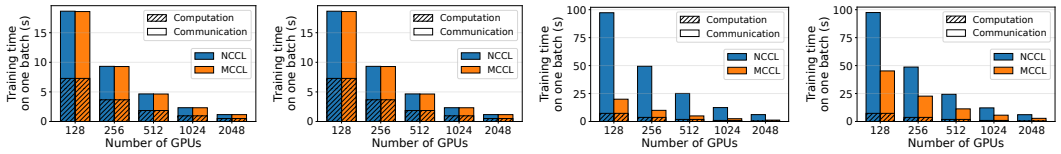
E End-to-End Validation

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).



(a) No failure (b) Fail 1 (Switch) (c) Fail 2 (Link) (d) Fail 3 (GPU)

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



(a) No failure (b) Fail 1 (Switch) (c) Fail 2 (Link) (d) Fail 3 (GPU)

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 ILP 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
ILP 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
ILP 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 ILP 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 multicommodity-flow integer program (Appendix G.2), relaxing it to a max–min concurrent-flow Linear Problem (LP) (Appendix G.3), and applying a Dantzig–Wolfe decomposition to recover the master and pricing subproblems from 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 $\big((d,t),\sigma_d\big)$ 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 $\tilde{\delta}^+(v_i)$ and $\tilde{\delta}^-(v_i)$ 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

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

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

G.2 Compact 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 $c_a\,\chi$:

$$\min \quad \chi \tag{11}$$

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

$$\textstyle\sum_{k \in \mathcal{K}} \sum_{a_i \in \tilde{A}_a} f_{a_i}^k \;\leq\; c_a\,\chi \qquad \forall a \in A, \tag{13}$$

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

where the supply/demand vector encodes the per-commodity demand $b_{v_i}^k = K(\mathbf{1}[v_i{=}\sigma_d] - \mathbf{1}[v_i{=}s_0])$. 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 $K/\chi^\star$, where $\chi^\star$ is the optimal congestion. The overall throughput is recovered by scanning the chunk budget

$$z_{\max}^{\mathrm{ILP}} \;=\; \max_{K \in \mathbb{Z}_{>0}} \frac{K}{\chi^\star(K)}. \tag{15}$$

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):

$$\max \quad z \tag{16}$$

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

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

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

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 $s = src(c)$ 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 $src(c) \to d$ route for each destination $d \in dst(c)$ 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.