USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

HeteCCL: Synthesizing Near-Optimal Collective Communication Schedules for Heterogeneous GPU Clusters

Chenyang Hei, Fuliang Li, and Jiayi Li, *Northeastern University;* Jiamin Cao, *Alibaba Cloud;* Chengxi Gao, *Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences;* Xiuzhu Sha, Tongrui Liu, and Dengke Zhang, *Northeastern University;* Ennan Zhai, *Alibaba Cloud;* Xingwei Wang, *Northeastern University*

https://www.usenix.org/conference/nsdi26/presentation/hei

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

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

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

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

HeteCCL: Synthesizing Near-Optimal Collective Communication Schedules for Heterogeneous GPU Clusters

Chenyang Hei[1], Fuliang Li[1]*, Jiayi Li[1], Jiamin Cao[2], Chengxi Gao[3], Xiuzhu Sha[1], Tongrui Liu[1]
Dengke Zhang[1], Ennan Zhai[2], Xingwei Wang[1]*
[1]School of Computer Science and Engineering, Northeastern University, Shenyang 110819, China
[2]Alibaba Cloud [3]Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences

Abstract

Training large language models demands massive computing and networking resources. However, existing clusters often face shortages of homogeneous resources and vendor lock-in, forcing the use of heterogeneous hardware, which makes synchronizing training across nodes highly challenging. Current solutions to cluster heterogeneity suffer from low collective communication efficiency, with suboptimal scheduling and slow algorithm synthesis. We present HeteCCL, a unified method for generating near-optimal collective communication schedules on heterogeneous clusters. HeteCCL models the cluster topology and link bandwidth in detail, quantizes data chunks at the schedule-step level, and formulates the scheduling problem as a maximum parallel transfer problem on a weighted directed graph. To accelerate synthesis, HeteCCL encodes bandwidth and routing constraints as SMT formulas and applies counterexample-guided inductive synthesis to refine constraints and prune the search space iteratively. Experiments on heterogeneous testbeds, each consisting of 32 H20 and V100 GPUs, show that HeteCCL outperforms NCCL, TACCL, and TE-CCL, achieving up to 2.8×, 4.4×, and 2.6× higher bandwidth. It also accelerates synthesis by up to 2 orders of magnitude compared to state-of-the-art efforts, and improves end-to-end training efficiency by 23%–37%.

1 Introduction

The increasing scale of large language models demands immense computational power for training [15]. As a result, datasets are distributed to multiple servers, and large models are separated into different workers, leading to various parallel training modes like data parallelism, tensor parallelism, and pipeline parallelism [21, 33, 34]. Parallelisms require collective communication to synchronize model gradients, optimizer states, and model weights. However, as the model size and cluster scale increase, the proportion of time spent on communication rises significantly, gradually becoming a bottleneck [29].

Efficient data movement is typically enabled by collective communication algorithms that are tailored to the underlying topology [8]. These algorithms can be viewed as schedules of communication operations over time. Designing efficient collective schedules is therefore critical to alleviating communication bottlenecks.

However, current collective communication libraries (CCLs) do not apply to modern clusters with diverse links. Today's production CCLs largely optimize around a small set of algorithm families (e.g., ring- and tree-based collectives) and then map these templates onto the detected topology. While such libraries are topology-aware in the sense that they recognize different interconnects and routing options, their execution structure is typically uniform across participants. In heterogeneous topologies, where link capacities differ by generation, placement, or network tier, this uniformity often makes end-to-end throughput sensitive to the slowest links and can leave faster links underutilized.

Moreover, heterogeneity is increasingly the norm in modern training clusters, which further amplifies this challenge. On the hardware side, GPU architectures and interconnect technologies advance rapidly, while service providers expand their infrastructure incrementally by purchasing successive generations of GPUs. This naturally leads to clusters composed of heterogeneous devices, differing not only in compute and memory hierarchies but also in communication capabilities. For example, deploying systems such as NVIDIA DGX B200 [12] alongside earlier platforms such as DGX H200 [10] or DGX A100 [9], while the NVLINK and network fabric simultaneously evolve across tiers and link capacities. Even within a single platform generation, systems may expose different numbers of network interfaces depending on the specific server configuration. For example, [26, 38] report that in their GPU clusters, some nodes (e.g., H800 servers) are equipped with eight 400-Gbps NICs providing one per GPU, while others (e.g., H20 servers) have four 400-Gbps NICs shared by two GPUs each. On the application side, training

*Co-corresponding authors.

workloads frequently run under these conditions, including parameter-efficient fine-tuning such as LoRA that mixes older and newer accelerators, long-running jobs where partial slowdowns or component replacements introduce transient heterogeneity, and incremental cluster expansion where newly added racks differ in GPU or network capabilities. Under these trends, heterogeneous clusters are becoming an important direction for large-scale systems, and the challenge of efficiently handling diverse interconnect links becomes even more critical in such environments.

Furthermore, existing work is for homogeneous clusters. Optimizing collective communication schedules is particularly critical in heterogeneous clusters with diverse links and topologies. Manually designing such algorithms requires substantial expert knowledge [13], and the process is time-consuming, error-prone, and infeasible for large-scale systems. This has motivated research on automatic synthesis of collective communication algorithms [5, 20, 30]. However, existing approaches are primarily tailored to homogeneous clusters and fail to scale effectively in heterogeneous environments. For example, SCCL [5] aims for full optimality by exhaustively modeling all aspects of the system with fine-grained precision, which significantly enlarges the solver's search space in heterogeneous clusters. TACCL [30] relies on detailed completion-time modeling for each transfer step, which becomes intractable under diverse link characteristics. TE-CCL [20] increases partition granularity and shortens time-slot intervals to capture link-level variations in heterogeneous clusters, but this comes at the cost of drastically increased synthesis complexity. SyCCL [7] leverages cluster topology symmetry to accelerate synthesis, significantly improving the efficiency of communication schedule generation. However, its reliance on symmetry makes it ineffective in heterogeneous network environments, where topologies are highly asymmetric. As a result, SCCL supports scheduling only at the single-server level; TACCL fails to generate an AllGather operator for a 64-GPU heterogeneous topology within 24 hours; and even with scalability optimizations, TE-CCL still requires more than 9 hours—far beyond acceptable limits. Moreover, SyCCL, which relies on symmetry insights, degrades to TE-CCL in heterogeneous topologies where such symmetry no longer holds.

This paper presents HeteCCL, a high-performance and scalable collective communication schedule synthesizer for heterogeneous cluster topologies. We identify the root cause behind the trade-offs made in prior solutions: in heterogeneous clusters, the synthesized theoretical schedules often diverge significantly from actual execution, becoming a major source of performance loss. This occurs because existing synthesizers encode the completion time of each primitive transfer into MILP formulas, yet GPU link bandwidths vary across the cluster. When primitives with highly uneven durations are scheduled in the same step without backend constraints, slower transfers delay faster ones, creating pipeline stalls ("bubbles") and degrading overall communication performance. We summarize the main challenges to designing a scalable and efficient collective communication schedule synthesizer for heterogeneous clusters as follows.

Challenge 1: Ensuring that primitive durations within the same transfer step remain balanced in heterogeneous clusters. Encoding only the transfer completion time of primitives on each link cannot produce well-balanced schedules under heterogeneous conditions. Conversely, enhancing heterogeneity awareness further complicates scheduling and reduces solver performance.

Challenge 2: The absence of usable topology symmetry–based pruning in heterogeneous clusters severely limits synthesizer efficiency, highlighting the need for a fast search-space pruning method that remains effective for heterogeneous clusters.

To address these challenges, HeteCCL introduces the key concept of *chunking*. Chunking divides each physical transmission link into multiple isolated logical links, equal to the number of chunks that can be transferred in parallel within the same algorithm step, so that link capacities within each step are more uniform. This balances primitive durations across paths, aligns tail latencies of transfer steps, and transforms the overall problem from collective communication scheduling on heterogeneous GPUs and networks into a maximum parallel primitive scheduling problem on a weighted directed graph.

To address the scalability issues caused by the failure of symmetry-based pruning and to avoid inefficient solver execution, HeteCCL introduces a pruning strategy tailored for heterogeneous clusters: counterexample-guided inductive synthesis [1]. Here, a counterexample refers to a transmission step that violates the constraint model during synthesis. This step is used as a checkpoint for restarting subsequent iterations. The counterexample guides the synthesis process by encoding incremental constraints into the model, enabling batch pruning of all potential transmission plans that could lead to similar violations after this checkpoint. This approach significantly reduces the search space, improves synthesis efficiency, and shortens synthesis time.

To summarize, our main contributions are listed as follows:

- We present HeteCCL, the first collective schedule synthesizer designed for heterogeneous clusters. We reformulate the synthesis challenge as a directed, weighted multi-path graph problem, transforming intractable continuous-time routing decisions into a discrete optimization that maximizes parallel chunk movement at each step.
- We built HeteCCL as an end-to-end system that accepts a heterogeneous topology description and produces an executable schedule. Chunking standardizes transmission at step granularity so that chunk-level primitives have comparable duration, which makes per-step load balancing across slow and fast links explicit. On top of this formulation,

HeteCCL encodes routing and capacity constraints in SMT (Satisfiability Modulo Theories) and couples it with an enhanced counterexample-guided pruning strategy, making synthesis practical at scale while preserving correctness of the returned schedules.

- We extensively evaluate HeteCCL, showing significant improvements in synthesis, communication, and training efficiency. HeteCCL reduces synthesis time by up to 2 orders of magnitude compared to TE-CCL. In heterogeneous clusters, it achieves bandwidth improvements for synthesized AllGather and AllReduce algorithms, with 2.8× over NCCL, 4.4× over TACCL, and 2.6× over TE-CCL. For end-to-end training of large language models, HeteCCL improves training efficiency by 37% and 23% compared to TACCL and TE-CCL, respectively.

2 Background and Motivation

Optimizing collective communication performance in heterogeneous clusters is far from trivial. In this section, we first introduce the background of collective communication, communication schedule synthesizers, and heterogeneous clusters (§2.1). We then present experimental analysis demonstrating why state-of-the-art approaches fail to scale effectively to heterogeneous clusters (§2.2).

2.1 Background

Collective communication is a specific communication pattern among workers within a collaborative group, employed to aggregate gradients across GPUs in distributed training. Many vendors enhance cluster communication efficiency during training by providing collective communication libraries (CCLs) tailored to their respective hardware, such as NCCL [11] and RCCL [3]. These libraries implement high-performance collective communication operators (e.g., ALLREDUCE, ALLGATHER, etc.) for distributed training. Operators are implemented using collective communication algorithms. NCCL provides simple and efficient implementations, including ring- and double binary tree-based collectives.

These algorithms can be viewed as communication schedules instantiated on a given topology, and their realized performance depends on how well the schedule matches the underlying link capacities and contention structure. However, the fixed schedules provided by existing communication libraries are only effective for specific standardized topologies, but lack the consideration and targeted design for uneven network bandwidth layers and diverse link capacity, thus resulting in poor bandwidth utilization and efficiency in complex topologies due to irregular and varied connections in large-scale clusters [8].

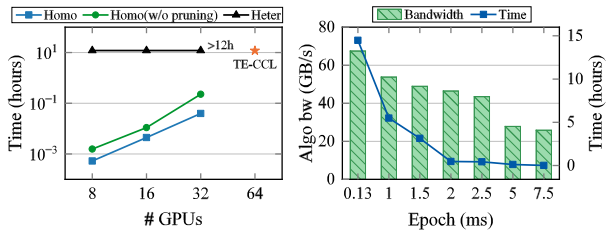
To address this problem, existing work treats the communication schedule synthesizer as an effective approach to approximate the optimal solution. These methods encode topology and transmission constraints into Mixed Integer Linear Programming (MILP), Linear Programming (LP), or Satisfiability Modulo Theory (SMT) [5, 30]. The synthesizer then derives both the data routing (i.e., transfer paths) and the schedule from the initial state, where GPUs hold their local chunks, to the target state, where all required chunks are available for computation. By exploring the solution space as fully as possible, the synthesizer aims to approach the optimal schedule.

Due to the rapid advancement of GPUs and their associated transmission systems, data centers may incorporate GPUs of different generations and configurations to ensure robustness and meet increasing training demands. For instance, a data center might deploy the latest NVIDIA DGX B200 [12] along with earlier models like the DGX H200 [10] or even the DGX A100 [9], depending on procurement batches. Besides, networking devices also vary in link capacities with different bandwidth (from 100Gbps to 400Gbps or beyond), which further complicates the GPU networking topologies [33]. Therefore, efficiently utilizing these heterogeneous devices is essential, and effective collective communication is crucial for efficient training.

2.2 Motivation

State-of-the-art communication schedule synthesizers, such as TACCL [30] and TE-CCL [20], model the scheduling problem as an MILP problem. They encode topology, link bandwidth and latency, path selection, and transfer ordering as constraints, and finally map the schedule into algorithmic steps. However, these approaches fail to generate efficient schedules for heterogeneous clusters. Their encoding minimizes overall completion time by assigning each link's transfer duration to the schedule, which is highly inefficient in heterogeneous settings. Because link bandwidths vary widely, even chunks of the same size can have very different transfer times. When such transfers are scheduled in the same step, the disparity in durations becomes severe. Without explicitly addressing this issue, the synthesized "optimal" algorithms often perform poorly in practice.

Although TACCL and TE-CCL claim to support heterogeneous modeling, their efforts remain limited to narrow cases. For example, TACCL employs "ordering" and TE-CCL introduces an adjustable "epoch" parameter, but these only capture bandwidth differences between intra-server (NVLink) and inter-server (RoCE) links. Both systems still assume identical servers across the cluster, a constraint evident in their released artifacts, which provide only a single-server sketch replicated across nodes. In addition, they rely on symmetry-based pruning to accelerate synthesis, as illustrated in Figure 14 in the Appendix. These design choices show that their methods are essentially tailored for homogeneous clusters. Extending them to heterogeneous clusters, where servers may differ in generation or link capability, requires trade-offs or compromises that introduce prohibitive overhead, making these approaches ineffective or inefficient. In the following, we analyze and



(a) Pruning and heterogeneous (b) Impact of epoch

Figure 1: Impact of symmetry failure and Epoch parameter on synthesis efficiency.

experimentally demonstrate the limitations of their heterogeneity handling.

Building fine-grained models for heterogeneous clusters leads to severe search-space expansion. The complexity of data routing grows exponentially with the number of nodes and heterogeneous links, as routing decisions must jointly consider optimal paths and interactions among multiple data blocks. Dependencies and timing constraints between operations further increase the difficulty of finding feasible solutions. Under these conditions, additional sequential scheduling decisions (as in TACCL) or finer-grained time-slot capture (as in TE-CCL) make it infeasible to obtain solutions within a reasonable time. Figure 1 illustrates this problem: in a heterogeneous setting with only 8 GPUs, TACCL fails to synthesize an optimal AllGather schedule within 12 hours (Figure 1(a)). TE-CCL shows a similar trend (Figure 1(b)): achieving smaller performance loss requires smaller epoch values to capture finer time intervals, but synthesis time grows sharply, exceeding 14.5 hours for AllGather on 32 GPUs. Moreover, using larger epochs reduces synthesis time but causes over 60% performance loss due to coarse-grained scheduling.

The failure of symmetry insights results in poor scalability. To accelerate synthesis, existing approaches typically prune the search space based on symmetry in cluster topology and GPU communication schedules. The core idea is that in homogeneous clusters, near-optimal schedules exhibit symmetry across GPUs: they share similar step counts, instructions per step, transfer patterns, and overall structure. Such symmetry ensures efficiency. Current synthesizers leverage this property by generating a schedule for a sub-domain and extending it to the entire topology through symmetric transformations such as shifting, rotation, or mirroring. However, this strategy fails in heterogeneous clusters (Figure 11 in Appendix C), where no such symmetry exists. Instead, the entire topology must be encoded with global constraints rather than replicated sub-constraints, which greatly enlarges the search space while eliminating effective pruning, leading to poor scalability in synthesis.

Both the failure of sequential scheduling and the breakdown of symmetry insights highlight a common issue: existing methods cannot strike an effective balance between efficiency and scalability in heterogeneous settings, and often perform poorly on both.

Main Idea. To address these challenges faced by existing works, we introduce HeteCCL, an efficient method for synthesizing optimal collective communication algorithms tailored for heterogeneous clusters. HeteCCL aims to optimize heterogeneous collective communication to minimize overall completion time. While GPUs of different generations differ in compute capability and memory capacity, we do not address workload-level computational heterogeneity. Allocating workloads proportionally to device performance to maximize overall compute utilization is a separate optimization problem [17, 28]. Our focus here is solely on the communication subsystem, and we further discuss the orthogonality between computation and communication in §7. Accordingly, our modeling considers only the bandwidth and latency differences across GPU generations, as well as bandwidth gains from newer NVLink, NIC, and switch technologies.

However, within the specific context of collective primitives, a distinct form of computational heterogeneity must be considered: it manifests primarily in the execution of reduction kernels. We quantified this impact through microbenchmarks across GPU generations, as shown in Figure 18 in the Appendix. The results demonstrate a distinct shift in performance bottlenecks. For large tensor transfers, transmission latency (milliseconds) dominates the execution time, rendering computation (microseconds) relatively minor. However, in the latency-sensitive regime involving small message sizes, the transmission cost decreases significantly. Consequently, the computational overhead of reduction kernels becomes a non-negligible fraction of the total step latency. Neglecting this factor could lead to suboptimal schedules that accumulate excessive computational delays. Accordingly, our modeling considers not only the bandwidth and latency differences across GPU generations (including gains from newer NVLink, NIC, and switch technologies), but also explicitly integrates reduction computational latency into our composite cost model to ensure fidelity across the entire spectrum of message sizes. This formulation is detailed in the following section.

3 Insight and Design Overview

In §3.1, we present our core insight, which enables the synthesis of heterogeneous schedules that are both efficient and scalable. §3.2 provides an overview of the system.

3.1 Insight

From the perspective of collective communication, a heterogeneous cluster topology can be equivalently viewed as a set of discrete links with complex and imbalanced bandwidth. A complete communication schedule can be decomposed into multiple algorithmic steps.

Observation 1: Imbalanced primitive durations slow down execution. We provide an example to illustrate how imbalanced primitive durations within the same step can delay

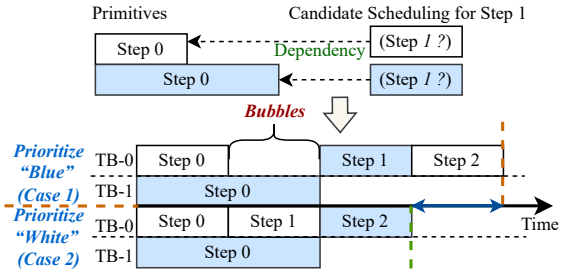


Figure 2: Pipeline stall induced by primitive execution heterogeneity on heterogeneous links.

schedule completion, thereby introducing pipeline bubbles. Some basic concepts are clarified as follows:

(i) Dependency. Primitives may have data dependencies, meaning operations on the same data block must be executed in order, and primitives within the same thread block (TB) must follow the step sequence.

(ii) Connection-oriented TB allocation. Each TB is responsible for executing data transfer instructions on a single link throughout its lifetime.

(iii) Schedule plan. We refer to the explicit schedule produced by the synthesizer as the schedule plan. It specifies both chunk routing and TB scheduling. Since TB resources are allocated in a connection-oriented manner, a TB can transmit only one chunk via one primitive within a step.

As illustrated in Figure 2, we observe two potential execution scenarios when primitives of varying durations contend for resources.

- **Case 1 (Head-of-Line blocking):** If a dependent primitive (waiting for a long-tail operation) is scheduled ahead of an independent one, it blocks the execution pipeline. Due to the serial nature of primitive execution within a Thread Block, the independent "White" primitive is forced to wait behind the stalled "Blue" primitive, creating significant idle time (bubbles) despite resources being available.
- **Case 2 (Ideal Reordering):** The shorter, independent primitive would be scheduled first, effectively overlapping to hide latency.

Achieving Case 2 consistently is infeasible with current approaches because existing synthesizers do not encode sufficient fine-grained ordering constraints to guarantee it. They typically model the aggregate completion time of a step but leave the micro-execution order of primitives within that step unconstrained. Consequently, the backend executes the generated schedule linearly without awareness of the duration disparities. This lack of explicit structural enforcement in the synthesized plan frequently results in Case 1, where unchecked long-tail primitives block critical paths and induce global pipeline bubbles.

This issue is common in heterogeneous clusters. If the synthesizer does nothing, the schedule may suffer from performance loss during execution, even when the algorithm achieves high theoretical bandwidth. On the other hand, handling this issue frequently can increase the complexity of synthesis and reduce scalability. Thus, we conclude **our first key insight** is as follows.

> **Insight 1:** For a schedule to perform well on a heterogeneous topology, *each step should ensure that the execution time of its communication primitives remains balanced.*

In other words, no step should suffer from excessive tail latency, which at the global level manifests as pipeline bubbles. In §4, we explain how we leverage a chunking technique to turn this insight into practice.

Observation 2: Counterexample-guided pruning works in concert with Insight 1. Since Insight 1 ensures that primitives within the same step have comparable durations, we can ignore irregular delays caused by complex dependencies. This makes pruning with counterexamples highly structured: once a counterexample arises, all subsequent schedules derived from it remain invalid. Consequently, this ensures that the pruning process is sound, preserving the reachability of the global optimum within the search space. More importantly, counterexamples enable pruning at scale. A step that triggers a counterexample can be regarded as the root of an entire subtree of scheduling plans, effectively pruning at a factorial magnitude (greater than n!, where n is the depth of the subtree).

> **Insight 2:** It is highly effective to leverage *counterexamples* to guide pruning in the synthesis process, serving as a substitute for the failed "symmetry" assumption in heterogeneous settings.

3.2 Overview of HeteCCL

In HeteCCL, we develop an efficient synthesis approach to generate near-optimal collective schedules for heterogeneous clusters. As illustrated in Figure 3, the workflow begins with a domain-specific language (DSL) designed to systematically capture the heterogeneity of computational and communication resources. This DSL formalizes hardware profiles into a unified input format, enabling HeteCCL to account for variability across discrete data chunks and diverse connection types (e.g., NVLink vs. PCIe). For complete syntax definitions, please refer to Appendix A. The interpreter converts this DSL input into an abstract syntax tree (AST), which is then quantized into a directed graph representing heterogeneous multi-path transmissions using a chunking method (§4).

HeteCCL employs an SMT-based program synthesis approach to encode data block routing in the heterogeneous topology as SMT formulas, formally modeling the constraints

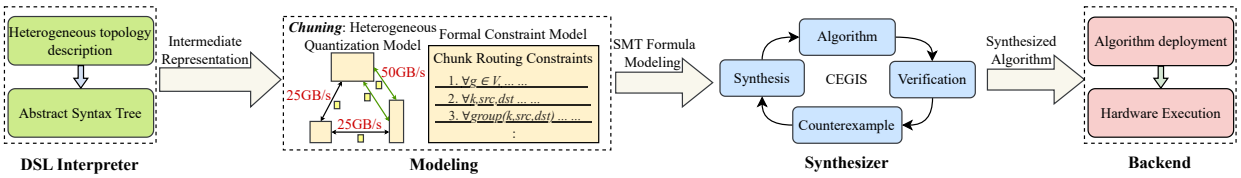


Figure 3: Overview of HeteCCL. HeteCCL inputs the heterogeneous topology description, converts it to an intermediate representation, and builds a quantified link model with formal constraints. The synthesizer uses CEGIS to generate the optimal collective communication algorithm, which is then deployed by the backend.

for the solver (§5). The constructed topology model and constraints are fed into the synthesizer, where high-level specifications and constraint models reduce the search space and accelerate solving.

To further improve synthesis efficiency, a counterexample-guided inductive synthesis (CEGIS) algorithm [2, 27] iteratively adds constraints to the SMT model based on counterexamples encountered during synthesis, efficiently finding near-optimal collective communication solutions. Finally, HeteCCL deploys the synthesized schedules to GPU hardware through the collective communication backend.

4 HeteCCL Chunking

In this section, we establish the formal modeling foundation of HeteCCL. We first detail how we quantify diverse link capabilities and formalize the cluster topology as a directed graph (§4.1). Based on this formulation, we define the synthesis problem and optimization goal (§4.2), and finally provide a theoretical analysis of the discretization trade-offs (§K).

4.1 Heterogeneity Modeling

HeteCCL introduces an innovative quantification method called *chunking*, which abstracts diverse connection types (e.g., NVLink, PCIe, RDMA) into a unified point-to-point (p2p) model. Unlike simple bandwidth slicing, chunking quantifies the effective transmission capability of each link by incorporating the $\alpha-\beta$ cost model [30] (accounting for both latency and bandwidth). Specifically, a link's capability is mapped to the number of logical chunks that can be transmitted in parallel within a normalized time step, where each chunk is quantified to a baseline size. Due to the connection-oriented TB allocation and the fact that each TB operates on only one primitive at a time (which in turn handles only one chunk), the parallel transmission of chunks at any given moment can be equivalently represented as multiple transmission paths, which reflects the link's transmission capability via chunking. This approach effectively transforms the composite transmission cost into discrete parallel paths, ensuring that primitives across heterogeneous links exhibit balanced durations.

Our micro-benchmark results (Figure 4) confirm that in the bandwidth-saturated regime (e.g., buffer sizes $\geq$ 128MB), concurrent data transfer via parallel thread blocks (TBs) achieves an effectively linear partitioning of physical link bandwidth. Specifically, scaling from 4 to 8 TBs maintains constant aggregate throughput, implying that the bandwidth allocated to each TB scales inversely with the thread count. While hardware arbitration may introduce micro-scale variations, our empirical data shows that at the granularity of collective chunks, these variations are negligible, validating the linear cost model used in our synthesis.

Additionally, HeteCCL decomposes complex cluster topologies into multi-path directed graphs that can be chunked into weighted parallel paths, transforming the parallel path problem in heterogeneous GPU communication graphs.

In the heterogeneous communication graph, we explicitly quantify the transmission potential of each edge using a cost-aware normalization strategy. To ensure execution synchronization across diverse links, we establish a *reference time step*, denoted as τ_{ref}, defined as the duration required to transmit a single chunk over the bottleneck link type (e.g., PCIe) derived from our cost model (formalized in §4.2). Accordingly, for any specific physical link (u,v) with n lanes, its *logical transmission capability*, $C(u,v)$, is calculated by normalizing its specific transmission cost $\tau_{u,v}$ against this baseline:

$$C(u,v) = \left\lfloor \frac{\tau_{ref}}{\tau_{u,v}} \right\rfloor \times n \tag{1}$$

This metric $C(u,v)$ strictly defines the number of effective parallel sub-paths available between nodes u and v. By projecting physical heterogeneity onto this cost-normalized dimension, HeteCCL ensures that transmitting $C(u,v)$ chunks on a fast link consumes approximately the same duration as transmitting a single chunk on the reference link. This transformation effectively converts the complex latency-bandwidth asymmetry into a standardized multi-graph structure, where every logical edge exhibits a uniform unit weight of one step.

4.2 Problem and Goal

Fundamentally, this quantification strategy ensures that the data chunk scale aligns strictly with the logical step scale in the transmission schedule. By maximizing parallelism, the constraint at each step is transformed from a complex transmission cost aggregation into a simple count of available parallel sub-paths. Crucially, this approach fundamentally transforms the problem space: it converts the scheduling formulation from a continuous-time domain, where the search space grows exponentially with diverse link latencies, into a

discrete packing problem with polynomial complexity. Consequently, the optimization objective shifts to maximizing the aggregate number of concurrent chunk transmissions at each global step. This formulation allows the synthesizer to bypass the tractability issues of fine-grained solvers and focus purely on global routing efficiency.

To instantiate this discrete modeling approach, we formalize the system connectivity as a labeled, multi-path directed graph $G = (V, E)$, which serves as the fundamental input for our synthesis engine. Here, V denotes the set of GPU nodes and E the set of interconnecting edges. Each edge $(u, v) \in E$ is weighted by the logical transmission capability $C(u, v)$, which serves as the capacity constraint for parallel chunk transmissions.

To mathematically capture architecture-specific contention (e.g., within PCIe complexes or network switches), we extend this basic graph definition with a set of *shared resource groups*, denoted as $\mathcal{S}_{shared}$. Unlike static P2P links, these groups model aggregate capacity limits. Formally, each group $g \in \mathcal{S}_{shared}$ is defined as a tuple (E_g, C_g), where $E_g \subseteq E$ encapsulates the subset of edges contending for a common hardware resource, and C_g denotes the resource's aggregate group capability. This formulation enables HeteCCL to enforce constraints on the cumulative flow: at any step, the sum of active chunks across all edges in E_g must not exceed C_g.

Cost Model. Following the formulation in prior synthesis works [30], we adopt an extended $\alpha - \beta$ model to evaluate the communication overhead across heterogeneous links (including both intra-node NVLink/PCIe and inter-node Ethernet/InfiniBand). The total cost is expressed as: $\tau_{total} = N_{step} \cdot \alpha + M_{trans} \cdot \beta + M_{comp} \cdot \gamma$, where N_{step} denotes the number of communication rounds (steps), and α represents the base latency (including hardware startup and link propagation delay). M is the total data volume transmitted, and β is the reciprocal of the link bandwidth, representing the transmission cost per unit of data. To account for computation in reduction primitives (e.g., AllReduce), we introduce M_{comp} as the volume of data processed, and γ as the reciprocal of the compute throughput (e.g., GPU reduction kernel throughput).

This model captures link heterogeneity at a fine granularity. Specifically, for a primitive transmitting and reducing a data chunk of size S_c, the latency cost is formalized as: $\tau_c = \alpha + S_c \cdot \beta + \mathbb{R}_{reduce} \cdot (S_c \cdot \gamma)$, where $\mathbb{R}_{\text{reduce}}$ is an indicator function that is 1 if the primitive involves a reduction operation and 0 otherwise.

In the context of HeteCCL, the formulation above serves as the foundation for our composite cost model. A key modeling decision is the exclusion of an explicit contention penalty term. We specifically refer to resource competition among concurrent TBs for endpoint injection bandwidth. As evidenced by our micro-benchmarks in Figure 4, parallel chunk transmission with up to 8 TBs exhibits contention-free bandwidth sharing. The aggregate throughput remains stable rather than degrading. This indicates that the hardware effectively

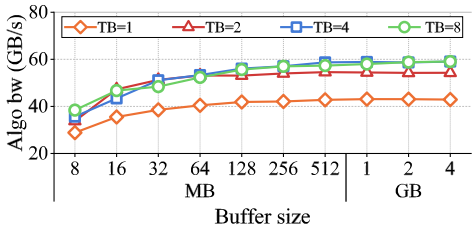


Figure 4: Impact of parallel TB execution on link communication stream.

manages concurrent flows through linear bandwidth partitioning. Consequently, the standard bandwidth cost term inherent in the model captures this sharing behavior without requiring non-linear penalty functions. Furthermore, the latency and computation terms remain independent of this transfer parallelism.

HeteCCL's chunking method synchronizes the chunk transmission with the scale of algorithmic transmission steps, therefore converting the scheduling problem into an issue of maximizing non-overlapping chunk transmissions. Each step in the algorithm incurs a cost of τ_c. After synchronizing the chunk and algorithmic steps, the total cost for the HeteCCL algorithm is defined as: $\tau_{HeteCCL} = N_{step} \cdot \max(\tau_c)$.

Therefore, the total cost of the entire communication and computation process is expressed as a polynomial, and the model is adjusted depending on the operation type (for example, non-reduction or reduction operations). For non-reduction operations, the computation cost term is excluded.

Theoretical analysis. We analyze the theoretical implications of the discrete time-step abstraction used in our chunking model. A potential concern is that normalizing heterogeneous link latencies to a coarse-grained reference step τ_{ref} (derived from the bottleneck link) enforces a *contiguity constraint* that might introduce performance overhead compared to continuous-time optimal schedules by excluding fine-grained schedules that exploit the early availability of faster links. We formally address this concern in Appendix K by demonstrating that the overhead is negligible, thereby preserving asymptotic bandwidth optimality.

5 HeteCCL Synthesizer

HeteCCL models the synthesis of near-optimal collective communication schedules as satisfiability modulo theories (SMT) formulas and employs a CEGIS algorithm [2, 27] to reduce the search space and efficiently solve the synthesis problem.

5.1 Routing Constraint Model

HeteCCL formulates the synthesis of collective algorithms as a constraints satisfaction problem over a directed graph. We encode the data routing logic and cluster constraints into first-order logical expressions, which are then solved by an SMT solver to generate a near-optimal schedule.

Problem Formulation. The synthesis model is defined by the following tuple: $\langle G, \mathcal{T}, \mathcal{K}, \mathcal{E}, pre, post \rangle$. $G = (V, E)$ represents the topology graph with link capacity constraints. $\mathcal{T} = \{0, 1, \ldots, N_{step}\}$ denotes the set of discrete time steps. $\mathcal{K} = \{k_1, k_2, \ldots, k_n\}$ represents the set of unique data chunks to be transmitted. $\mathcal{E}$ is the set of counterexamples derived from previous verification iterations, used to prune the search space. The functions $pre(u)$ and $post(u)$ define the initial set of chunks held by node u and the target set of chunks required by node u after execution, respectively.

We define decision variables to capture the state of data movement and timing across discrete time steps $t \in \mathcal{T}$:

- $Send(k, u, v, t) \in \{0, 1\}$: A boolean variable that is true if chunk k is transmitted from node u to node v exactly at time step t.
- $Has(k, u, t) \in \{0, 1\}$: An auxiliary boolean variable that is true if node u possesses chunk k at the beginning of time step t.

The solver derives the specific algorithmic schedule by asserting constraints over these variables. An operation is scheduled when the solver evaluates $Send(k, u, v, t) = 1$, and the causal availability of data is strictly tracked sequentially via the $Has(k, u, t)$ state transitions.

Constraints. We impose valid constraints over these variables to ensure execution correctness and state consistency, with the initial state $Has(k, u, 0)$ strictly initialized according to $pre(u)$.

First, a *causality constraint* ensures that a node must possess a chunk before forwarding it, and strictly prohibits sending redundant data to a node that already possesses it. Specifically, if chunk k is sent from u to v at step t:

$$\begin{aligned} &\forall k \in \mathcal{K}, \forall (u, v) \in E, \forall t \in \mathcal{T}, \\ &Send(k, u, v, t) \implies \big(Has(k, u, t) \wedge \neg Has(k, v, t)\big) \end{aligned} \tag{2}$$

Consequently, the state evolution operates as a strict set equivalence over time. A node possesses a chunk in the next step if and only if it already had it, or it successfully received it from at least one neighbor:

$$\begin{aligned} &\forall k \in \mathcal{K}, \forall u \in V, \forall t \in \mathcal{T}, \\ &Has(k, u, t+1) \iff \left(Has(k, u, t) \bigvee_{(v,u) \in E} Send(k, v, u, t) \right) \end{aligned} \tag{3}$$

Optimization Objective. The goal of the synthesizer is to find a valid routing plan that minimizes the total execution time. We formalize this objective as minimizing the total number of steps N_{step}: Minimize(N_{step})

Although the chunking formulation transforms the scheduling problem into a domain with polynomial complexity, directly synthesizing the full schedule for large-scale heterogeneous clusters remains resource-intensive due to the massive number of concurrent constraints. To address this, HeteCCL employs a *counterexample-guided inductive synthesis* (CEGIS) approach. This method efficiently prunes the search space by iteratively refining the schedule against constraints, rather than processing all global constraints in a single pass.

5.2 Counterexample-Guided Inductive Synthesis

The synthesis process begins by generating an initial candidate schedule, $\mathcal{S}_{init}$, using a greedy heuristic that maximizes link utilization at each time step. While this greedy initialization provides a rapid starting point, it operates without full awareness of global constraints and may produce a schedule that violates logical limits.

To identify these violations, HeteCCL enters a verification phase where it validates the current candidate schedule $\mathcal{S}_{curr}$ against the logical capacity limits. Specifically, we enforce three primary validity constraints: First, the link capacity constraint ensures that the number of chunks concurrently transmitted on any link (u, v) at any single time step t does not exceed its logical capacity $C(u, v)$ (i.e., the available parallel sub-paths):

$$\forall t \in \mathcal{T}, (u, v) \in E, \quad \sum_{k \in \mathcal{K}} Send(k, u, v, t) \leq C(u, v) \tag{4}$$

Second, the shared resource constraint enforces that the aggregate traffic through any shared group g (e.g., switches) at any single time step t remains within its group limit C_g:

$$\forall t \in \mathcal{T}, g \in S_{shared}, \quad \sum_{(u,v) \in g} \sum_{k \in \mathcal{K}} Send(k, u, v, t) \leq C_g \tag{5}$$

Third, the redundancy constraint prevents data duplication by ensuring each destination node receives a specific chunk at most once over the entire execution period:

$$\forall k \in \mathcal{K}, v \in V, \quad \sum_{t \in \mathcal{T}} \sum_{u \in V} Send(k, u, v, t) \leq 1 \tag{6}$$

Any identified violations are collected into a counterexample set $\mathcal{E}$. Crucially, we identify the earliest time step where a violation occurs and treat it as a *checkpoint*, denoted as t_{check}. This checkpoint serves as a pivotal marker that divides the schedule into a valid prefix and a flawed suffix, enabling an incremental synthesis strategy.

The guided search phase leverages this checkpoint to perform incremental repair rather than re-synthesizing the entire schedule from scratch. The SMT solver generates the next candidate schedule by applying distinct constraints based on the checkpoint. For all time steps prior to the violation ($t < t_{check}$), the solver preserves the operations from the valid prefix, effectively locking the consistent history to reduce the solver's workload. At the checkpoint step ($t = t_{check}$), the solver enforces new constraints derived from $\mathcal{E}$ to prevent the specific violation from recurring. To further optimize the repair quality at this step, we employ a strategy to maximize

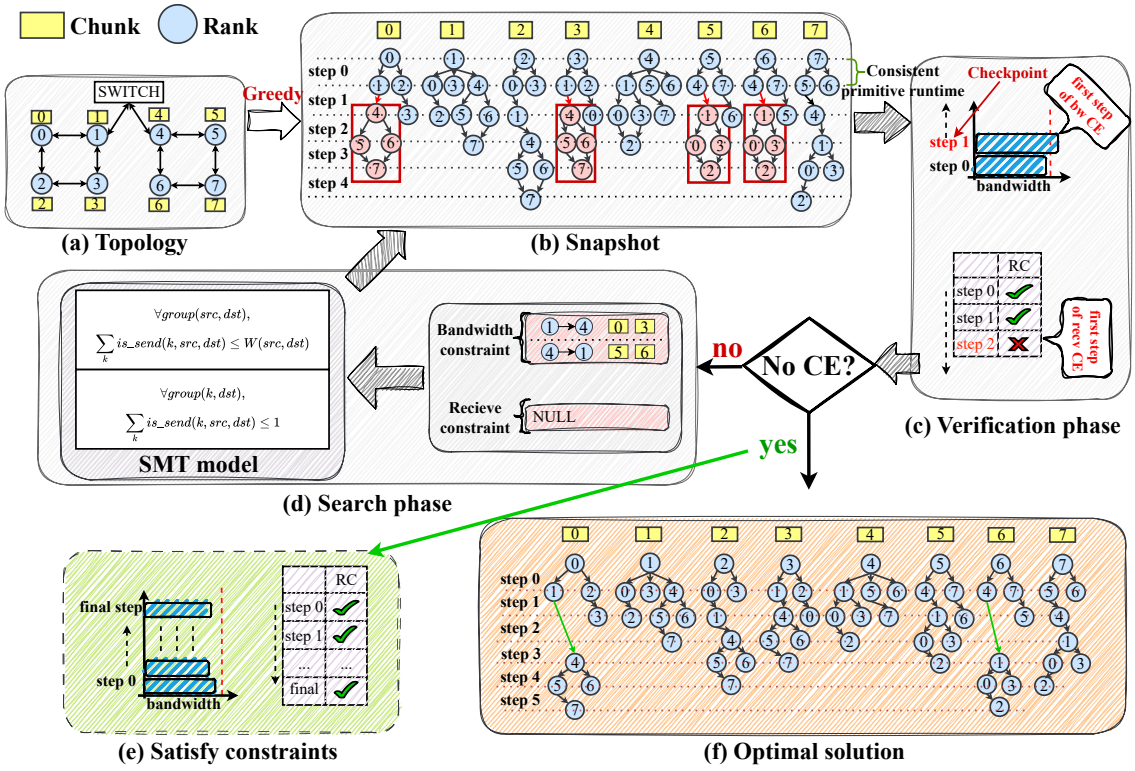


Figure 5: Workflow of counterexample-guided inductive synthesis.

link utilization, which supplements the SMT-generated operations with additional valid transmissions to ensure the link bandwidth is fully exploited. Finally, for the subsequent steps ($t > t_{check}$), the solver synthesizes a new sequence of operations to reach the target state *post*.

This cycle of verification, checkpoint identification, and incremental synthesis repeats, generating a sequence of intermediate schedule *snapshots* that progressively satisfy more constraints. In each iteration, the counterexamples act as permanent constraints that prune invalid branches of the search space. The process terminates when the verification phase returns an empty counterexample set ($\mathcal{E} = \emptyset$), at which point the current snapshot is output as the final schedule $\mathcal{S}_{fin}$.

Figure 5 illustrates a CEGIS algorithm example for generating an allgather algorithm with chunk=1 on two 4-GPU topologies. First, Figure 5(a) shows the topology, and Figure 5(b) captures the initial solution found using a greedy algorithm. Next, Figure 5(c) demonstrates the verification phase, where bandwidth and reception constraints are checked, and the first counterexample is identified (e.g., in Step 1, bandwidth congestion occurs between GPU 1 and GPU 4, and in Step 2, multiple receptions from GPU 7 to GPU 2 are recorded). Designating the earliest violation as the checkpoint (t_{check}), Step 1 is selected. Consequently, this first counterexample is output along with bandwidth constraints (chunks 0, 3 on link $1 \rightarrow 4$ and chunks 5, 6 on link $4 \rightarrow 1$). The process iterates through verification, and if a counterexample is found, it proceeds to the search phase, transforming the counterexample into constraints. These constraints are fed into the SMT solver to guide synthesis and checkpoint recovery, allowing for continued search. Once a new candidate solution is found, the process iterates again. If no counterexamples are found, the iteration stops, and the near-optimal solution is returned as the final synthesis result, as shown in Figure 5(f). Figure 5(e) illustrates the state when all constraints are satisfied. For the detailed algorithmic implementation, please refer to Algorithm 1 in Appendix H.

5.3 Schedule Lowering and Runtime Execution

The logical schedule $\mathcal{S}_{fin}$ produced by the CEGIS synthesizer defines *what* data to move and *when*, but it remains agnostic to the underlying hardware interfaces. To bridge this gap and ensure efficient execution on real-world clusters, HeteCCL implements a runtime layer that lowers the abstract schedule into hardware-specific primitives. This lowering process involves two key stages: instruction generation and backend dispatch.

Instruction generation and dependency management. HeteCCL first translates the discrete time steps in $\mathcal{S}_{fin}$ into a sequence of executable instructions (e.g., `Send`, `Recv`, `Reduce`). A critical challenge here is preserving the causal dependencies established during synthesis without introducing excessive

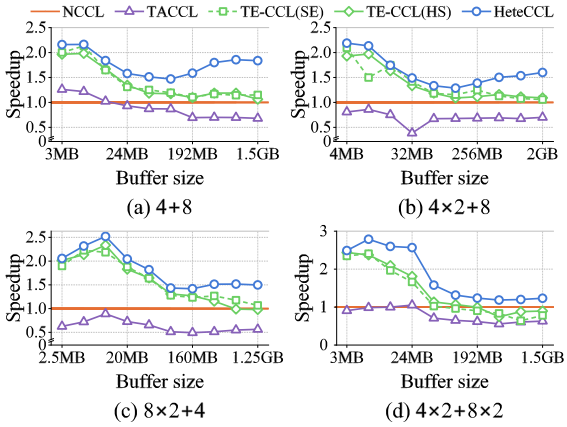


Figure 6: AllGather speedup relative to NCCL on H20 clusters. Compare HeteCCL with TACCL and two TE-CCL variants. SE (Synthesis Efficiency): TE-CCL's high-efficiency synthesis algorithm; HS (HeteCCL Step): search epochs aligned with HeteCCL steps.

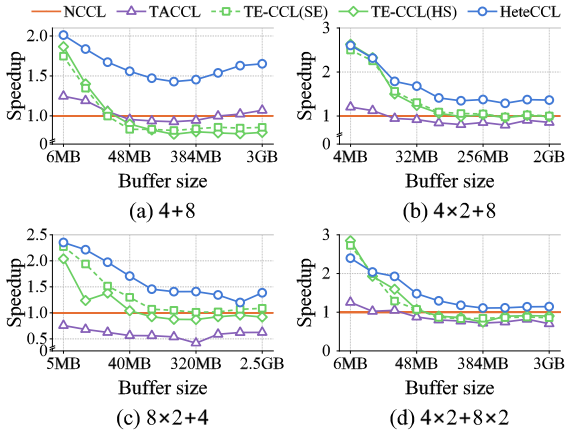


Figure 7: AllReduce speedup relative to NCCL on H20 clusters.

synchronization overhead. Rather than using global barriers that degrade performance, HeteCCL employs a fine-grained dependency mechanism. We map the synthesized steps to a directed acyclic graph (DAG) of GPU kernels. For each operation assigned to a GPU, we orchestrate its execution order in the schedule to ensure that a chunk is only processed after its requisite data dependencies are met. This design allows non-dependent chunks to be processed out-of-order, maximizing pipeline parallelism within the bounds of the synthesized logic.

Backend abstraction and execution. The logical schedule $\mathcal{S}_{fin}$ defines the timing and routing of data chunks but assumes abstract transmission capabilities. To execute this schedule on physical hardware, HeteCCL must map these logical operations to specific GPU execution units. Unlike traditional collective libraries (e.g., NCCL) that typically enforce a uniform channel topology (e.g., rings) where all channels cover all peers, HeteCCL implements a non-uniform channel mapping strategy. This approach dynamically assigns hardware resources based on the bandwidth heterogeneity.

In the runtime, a channel corresponds to a stream of operations managed by a dedicated GPU TB. To maximize link utilization, HeteCCL maps the concurrent chunks defined in the schedule directly to these parallel channels. For high-bandwidth connections (e.g., NVLink) where the synthesis model permits multiple concurrent chunks ($C(u,v) > 1$), HeteCCL instantiates multiple distinct channels to drive the link simultaneously. This multi-channel concurrency ensures that the physical link bandwidth is fully saturated by masking memory access latencies. Conversely, for low-bandwidth links (e.g., PCIe) with limited capacity, HeteCCL assigns fewer channels to prevent congestion and resource contention. This flexible mapping ensures that the execution parallelism aligns with the hardware heterogeneity captured in the chunking model, translating the theoretical multi-path advantage into realizable system throughput.

6 Evaluation

6.1 Experiment Setup

Heterogeneous cluster configuration: In this section, we conduct experiments using NVIDIA H20 96GB and V100 16GB GPU clusters. We constructed six heterogeneous environments for the experiments, named as $(4+8)$, $(4\times2+8)$, (4×3), $(4\times3+8)$, $(8\times2+4)$, and $(4\times2+8\times2)$.

Take the **Topology** $(4+8)$ as an example. This setup consists of one node with 4 GPUs and another node with 8 GPUs. The 8-GPU node has 1 SXM5 module (in the H20 server) or 2 SXM2 modules (in the V100 setup), with a total bidirectional bandwidth of 900 GBps/300 GBps. Each GPU has 9/6 NVLink ports (each at 75 GBps/25 GBps). In the H20 server, all GPUs are interconnected by 6 NVSwitches. In the V100 server, the modules are connected via a 48 GB/s UPI link, and each pair of GPUs shares a 16 GB/s PCIe switch. GPUs in the 4-GPU node are connected via NVLink. The nodes are interconnected by switches, with GPUs sharing 4 NIC bandwidths: (1) all servers with 400 Gbps in H20 setup, and (2) 8-GPU server with 100 Gbps and 4-GPU server with either 100Gbps or 64 Gbps in V100 setup[1]. The remaining five topologies are combinations of the above types.

For all schemes, we set a solving time limit of 12 hours. If the optimal solution is not found within this time, we use the last solution obtained before the deadline as the final solution, similar to TACCL [30].

Comparison baselines. We compared the bandwidth of synthesized AllReduce and AllGather algorithms on these topologies with NCCL, TE-CCL/SyCCL, and TACCL.

[1]We assume this behavior represents a broader trend, aiming to demonstrate that our approach can effectively adapt to heterogeneous topologies and successfully scale to systems with greater variability and higher economic costs.

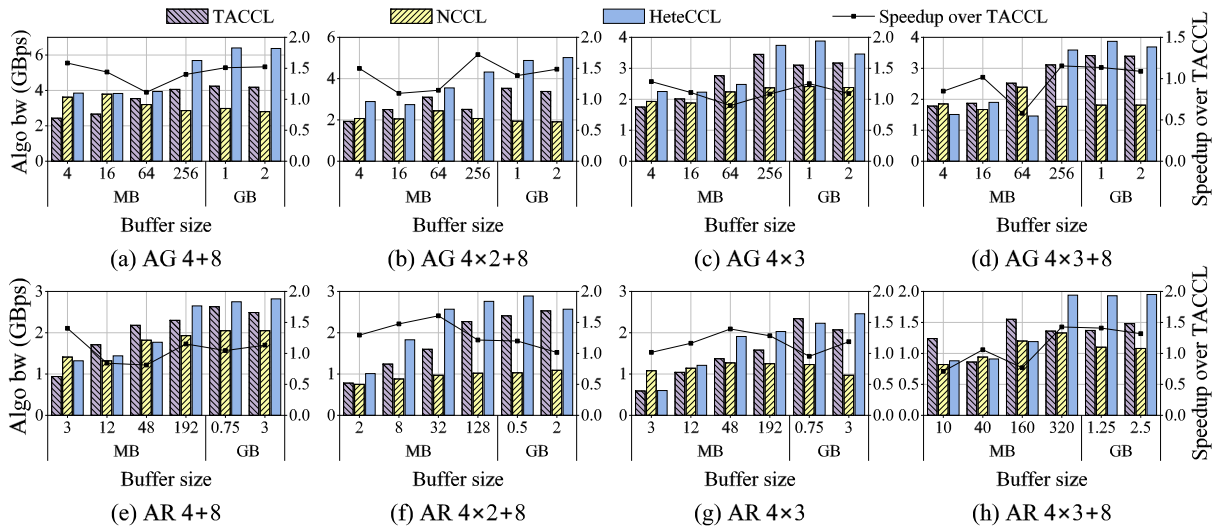


Figure 8: Bandwidth comparison of HeteCCL, TACCL, and NCCL on V100 heterogeneous clusters for different buffer sizes.

Table 1: Synthesis time of TACCL, TE-CCL, and HeteCCL.

# GPUs	Synthesis Time (s)			Speedup
	TACCL	TE-CCL	HeteCCL	
16	30	2.2	0.9	2.4×
32	Time Out	24.7	6.9	3.6×
48	Time Out	497	5.5	90.4×
64	Time Out	33990	105.3	322.8×

NCCL, a widely used collective communication library, serves as the baseline, while TE-CCL and TACCL represent state-of-the-art work in the automated synthesis of collective communication algorithms. It is worth noting that SyCCL degrades to TE-CCL in heterogeneous topologies due to the loss of symmetry. Therefore, comparing against TE-CCL is equivalent to comparing against SyCCL.

6.2 Synthesis Time

As shown in Table 1, the synthesis speed of HeteCCL scales significantly better than TE-CCL and TACCL as the size of the heterogeneous topology increases. TACCL struggles to find optimal solutions even at relatively small scales, such as 32 GPUs. TE-CCL requires more than 9 hours for 64 GPUs, far beyond a reasonable synthesis time, whereas HeteCCL consistently completes within minutes—remaining under 9 minutes for 64 GPUs, achieving a 2.4×–322.8× speedup over TE-CCL. The efficiency of HeteCCL comes from guiding the SMT solver with counterexample-guided inductive synthesis (CEGIS), which reduces the search space from all possible combinations of chunks, links, and steps to only those captured by counterexamples. This approach drastically prunes the SMT search space, enabling HeteCCL to achieve optimal bandwidth utilization in a much shorter time.

6.3 Communication Micro-Benchmark

Algorithm completion time. We evaluate HeteCCL on multiple heterogeneous topologies composed of H20 GPUs and compare its schedule completion time with that of the industry-standard NCCL and two state-of-the-art synthesizers, TE-CCL and TACCL. As shown in Figure 6 and 7, HeteCCL achieves the fastest completion time. For the AllGather algorithm, HeteCCL outperforms NCCL by 1.5×–2.5×, TE-CCL by 2×–2.6×, and TACCL by 2.2×–4.4×. For the AllReduce algorithm, HeteCCL improves over NCCL by 1.2×–2.8×, TE-CCL by 1.3×–1.5×, and over TACCL by 1.3×–3.6×.

Algorithm bandwidth. Figure 8(a) demonstrates the bandwidth performance of the HeteCCL-synthesized AllGather algorithm on a $(4+8)$ heterogeneous V100 GPU cluster, compared to TACCL and NCCL. Using a 1-chunk partitioning method, HeteCCL achieved speedups of 1.1× to 2.3× over NCCL and 1.1× to 1.6× over TACCL for buffer sizes from 4MB to 2GB, due to effective bandwidth utilization. As shown in Figure 8(b), HeteCCL outperformed NCCL by 1.4× to 2.6× and was on average 1.4x faster than TACCL, demonstrating efficient use of inter-node bandwidth. Figure 8(c) shows HeteCCL's performance on a (4×3) cluster, where it achieved 17% to 62% better performance than NCCL and 4% to 28% improvements over TACCL for buffer sizes from 4MB to 2GB, underscoring HeteCCL's generality across different topologies. Finally, Figure 8(d) shows that HeteCCL outperforms NCCL by 2.1× and TACCL by 1.2× on the $(4\times 3+8)$ topology.

As shown in Figure 8(e)-(h), the HeteCCL-synthesized AllReduce algorithm, which uses a single-chunk reduction per GPU, outperforms NCCL and TACCL by 1.4× for buffer sizes ranging from 3MB to 3GB in the $(4+8)$ heterogeneous topology. Similarly, on the other three heterogeneous clusters, HeteCCL outperforms NCCL by 1.8× to 2.8× and TACCL by up to 1.6×.

Our implementation, along with evaluations on homogeneous clusters, A100-based heterogeneous clusters, and extreme heterogeneity scenarios such as failures and slowdowns,

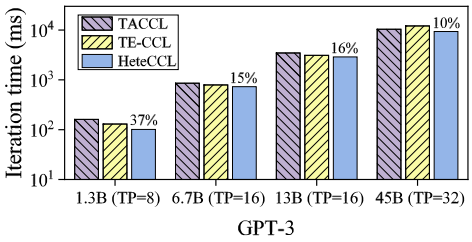


Figure 9: End-to-End training iteration time.

is provided in Appendices C-G.

6.4 End-to-End Training

We evaluate HeteCCL in distributed training of GPT-3 [4] models of varying scales, integrated into a Megatron variant [16, 31] with MSCCL [23] support. For fairness, we compare communication schedules synthesized by HeteCCL, TACCL, and TE-CCL under identical settings. Models with fewer than 45B parameters are deployed across two servers (16 GPUs), while larger models are deployed on four servers (32 GPUs). Tensor parallelism is used as the parallelization strategy. As shown in Figure 9, HeteCCL achieves consistent end-to-end training speedups over TACCL and TE-CCL, covering model sizes from 1.3B to 45B parameters. Compared to TACCL, HeteCCL reduces iteration time by 10%–37%, and compared to TE-CCL, achieves a speedup of 7%–23%.

7 Discussions and Limitations

Scalability via hierarchical composition. While our evaluation demonstrates direct synthesis for cluster units (e.g., up to 64 GPUs), applying SMT solving directly to datacenter-scale clusters (thousands of GPUs) is computationally intractable. However, industrial large-scale clusters [26] are typically constructed from identical, repeated building blocks (e.g., racks or pods). HeteCCL addresses scalability by exploiting this structural regularity through *hierarchical composition*. Similar to SyCCL [7] and Canvas [14], HeteCCL supports scalability by using this structural regularity through *hierarchical composition*. In practice, we first synthesize a highly optimized fine-grained schedule for a single pod (intra-pod). We then synthesize a coarse-grained inter-pod schedule by treating each pod as a single logical node with aggregated bandwidth. The runtime backend composes these two schedules to form the global execution plan. This approach decouples the synthesis complexity from the total GPU count, making the solver time dependent only on the complexity of the unique building blocks rather than the total cluster size.

Positioning as a communication kernel optimizer. HeteCCL operates as a specialized *communication kernel optimizer*, analogous to how cuBLAS [24] optimizes computational kernels (e.g., GEMM) for specific hardware. Our scope is distinct from but complementary to high-level training schedulers that manage computation-communication overlap [39]. By minimizing the absolute duration (makespan) of the collective communication kernel through optimal bandwidth utilization, HeteCCL reduces the exposure of communication events. This, in turn, provides a wider window for overlap schedulers to hide latency behind computation. Furthermore, regarding multi-tenancy, HeteCCL functions as a post-allocation optimizer: it assumes a specific set of resources has been assigned by the cluster job scheduler [6] and focuses solely on maximizing the utilization of those allocated heterogeneous resources, regardless of global cluster contention.

8 Related Work

Collective communication schedule optimization. To improve the efficiency and flexibility of collective communication, prior work [19, 23] has explored both customizable DSLs and automated synthesis methods. MSCCLang [13] enables developers to design custom collective algorithms with compiler optimizations that increase pipeline density and instruction fusion. In parallel, state-of-the-art synthesizers [5, 7, 20, 23, 30, 35, 37] aim to automatically generate optimal algorithms for diverse topologies, reducing manual effort and configuration time. However, these approaches rely heavily on expert knowledge or incur prohibitive synthesis costs, and thus often fail to produce high-performance schedules for heterogeneous clusters.

Computational Heterogeneity. Prior work on heterogeneous distributed training has primarily optimized model partitioning and scheduling [22, 32, 36, 39]. These systems adopt techniques such as program synthesis [36], MILP-based formulations [32], and heterogeneity-aware search to balance workloads [39], improve placement [18], and co-optimize memory [25] and parallelism [22]. While effective for computational heterogeneity, they remain orthogonal to our focus on communication efficiency.

9 Conclusion

In this paper, we introduced HeteCCL, a novel approach for synthesizing optimal collective communication algorithms tailored for heterogeneous GPU clusters. HeteCCL introduces a chunking method to balance primitive durations within the same scheduling step across heterogeneous links, thereby reducing pipeline stalls in the overall schedule. It further employs a counterexample-guided inductive synthesis approach to accelerate synthesis, compensating for the failure of symmetry-based pruning in heterogeneous clusters. Our evaluations demonstrate HeteCCL's superiority in reducing synthesis time, and improving bandwidth efficiency, and training efficiency over NCCL, TE-CCL, and TACCL.

Acknowledgments

We thank our shepherd, Junxue Zhang, and the anonymous reviewers for their insightful comments. This work is supported by the National Natural Science Foundation of China under Grant Nos. 62432003, 62572105 and U22B2005; the LiaoNing Revitalization Talents Program under Grant No. XLYC2403086.

References

[1] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In *International Conference on Computer Aided Verification*, pages 270–288. Springer, 2018.

[2] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In *Proc. of ACM CAV*, pages 270–288. Springer, 2018.

[3] Inc Advanced Micro Devices. ROCm Communication Collectives Library (RCCL). https://github.com/ROCm/rccl, 2024.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proc. of NeurIPS*, volume 33, pages 1877–1901, 2020.

[5] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing Optimal Collective Algorithms. In *Proc. of ACM PPoPP*, pages 62–75, 2021.

[6] 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 *Proc. of the ACM SIGCOMM*, page 1–15, 2024.

[7] 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 *Proc. of the ACM SIGCOMM*, pages 645–662, 2025.

[8] Sanghun Cho, Hyojun Son, and John Kim. Logical/-physical topology-aware collective communication in deep learning training. In *Proc. of IEEE HPCA*, pages 56–68, 2023.

[9] NVIDIA Corporation. NVIDIA DGX A100. https://images.nvidia.com/aem-dam/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf, 2022.

[10] NVIDIA Corporation. NVIDIA DGX H200. https://resources.nvidia.com/en-us-dgx-systems/dgx-h200-datasheet, 2023.

[11] NVIDIA Corporation. NVIDIA Collective Communications Library (NCCL). https://github.com/NVIDIA/nccl, 2024.

[12] NVIDIA Corporation. NVIDIA DGX B200. https://www.nvidia.com/en-us/data-center/dgx-b200/, 2024.

[13] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. MSCCLang: Microsoft Collective Communication Language. In *Proc. of ACM ASPLOS*, pages 502–514, 2023.

[14] Chenyang Hei, Yi Zhao, Fuliang Li, Chengxi Gao, Tongrui Liu, Xiuzhu Sha, and Xingwei Wang. Canvas: Scalable collective communication scheduling for large-scale gpu clusters. In *2025 IEEE 33rd International Conference on Network Protocols (ICNP)*, pages 1–11, 2025.

[15] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei Zhang. Characterization of Large Language Model Development in the Datacenter. In *Proc. of USENIX NSDI*, pages 709–729, 2024.

[16] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing Activation Recomputation in Large Transformer Models. *Proc. of MLsys*, pages 1–17, 2023.

[17] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. Colossal-AI: A Unified Deep Learning System For Large-Scale Parallel Training. In *Proc. of ACM ICPP*, pages 766–775, 2023.

[18] Guodong Liu, Youshan Miao, Zhiqi Lin, Xiaoxiang Shi, Saeed Maleki, Fan Yang, Yungang Bao, and Sa Wang. Aceso: Efficient parallel dnn training through iterative bottleneck alleviation. In *Proc. of ACM ASPLOS*, pages 163–181, 2024.

[19] 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*, pages 55–70, 2025.

[20] 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 *Proc. of ACM SIGCOMM*, pages 16–37, 2024.

[21] Kshiteej Mahajan, Ching-Hsiang Chu, Srinivas Sridharan, and Aditya Akella. Better Together: Jointly Optimizing ML Collective Scheduling and Execution Planning using SYNDICATE. In *Proc. of USENIX NSDI*, pages 809–824, 2023.

[22] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. Helix: Serving large language models over heterogeneous gpus and network via max-flow. In *Proc. of ACM ASPLOS*, pages 586–602, 2025.

[23] Microsoft. MSCCL. https://github.com/microsoft/msccl, 2022.

[24] NVIDIA Corporation. cublas library user guide. https://docs.nvidia.com/cuda/cublas/, 2024.

[25] Suchita Pati, Shaizeen Aga, Mahzabeen Islam, Nuwan Jayasena, and Matthew D Sinclair. T3: Transparent tracking & triggering for fine-grained overlap of compute & collectives. In *Proc. of ACM ASPLOS*, pages 1146–1164, 2024.

[26] 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 *Proc. of the ACM SIGCOMM*, page 691–706, 2024.

[27] Yiming Qiu, Ryan Beckett, and Ang Chen. FlexPlan: Synthesizing Runtime Programmable Switch Updates. In *Proc. of USENIX NSDI*, pages 613–628, 2023.

[28] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *Proc. of ACM SC*, pages 1–16, 2020.

[29] 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 *Proc. of ACM ISCA*, pages 581–596, 2022.

[30] 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 *Proc. of USENIX NSDI*, pages 593–612, 2023.

[31] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[32] Taegeon Um, Byungsoo Oh, Minyoung Kang, Woo-Yeon Lee, Goeun Kim, Dongseob Kim, Youngtaek Kim, Mohd Muzzammil, and Myeongjae Jeon. Metis: Fast automatic distributed training on heterogeneous GPUs. In *Proc. of USENIX ATC*, pages 563–578, 2024.

[33] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. Topoopt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *Proc. of Usenix NSDI*, pages 739–767, 2023.

[34] Zheng Wang, Anna Cai, Xinfeng Xie, Zaifeng Pan, Yue Guan, Weiwei Chu, Jie Wang, Shikai Li, Jianyu Huang, Chris Cai, Yuchen Hao, and Yufei Ding. Wlb-llm: Workload-balanced 4d parallelism for large language model training. In *Proc. of USENIX OSDI*, 2025.

[35] William Won, Midhilesh Elavazhagan, Sudarshan Srinivasan, Swati Gupta, and Tushar Krishna. Tacos: Topology-aware collective algorithm synthesizer for distributed machine learning. In *Proc. of IEEE/ACM MICRO*, pages 856–870. IEEE, 2024.

[36] Shiwei Zhang, Lansong Diao, Chuan Wu, Zongyan Cao, Siyu Wang, and Wei Lin. Hap: Spmd dnn training on heterogeneous gpu clusters with automated program synthesis. In *Proc. of ACM Eurosys*, pages 524–541, 2024.

[37] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Jason Fantl, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. Efficient direct-connect topologies for collective communications. In *Proc. of USENIX NSDI*, pages 705–737, 2025.

[38] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xuanzhe Liu, Xin Jin, and Xin Liu. Megascale-infer: Efficient mixture-of-experts model serving with disaggregated expert parallelism. In *Proc. of the ACM SIGCOMM*, page 592–608, 2025.

[39] Zhanda Zhu, Christina Giannoula, Muralidhar Andoorveedu, Qidong Su, Karttikeya Mangalam, Bojian Zheng, and Gennady Pekhimenko. Mist: Efficient distributed training of large language models via memory-parallelism co-optimization. In *Proc. of ACM Eurosys*, pages 1298–1316, 2025.

Appendix

A DSL Specification for Cluster Topology

To support the modeling formalized in §4.1, we implement a lightweight DSL that captures the hardware heterogeneity. As inputs to the HeteCCL synthesizer, this DSL quantizes transmissions into chunks of size S_{chunk} and specifies the attributes for the topology graph $G = (V, E)$.

The DSL schema includes the following core components:

- **Cluster Scale:** The total number of nodes $N_{cluster}$ and the GPU count N_i for each machine i.
- **Communication Profiles:** A set of bandwidth values $\mathcal{B}$ and latency costs $\mathcal{L}$. Crucially, the DSL differentiates between intra-node bandwidths ($\mathcal{B}^{\text{intra}}$) and inter-node bandwidths ($\mathcal{B}^{\text{inter}}$) to accurately reflect the physical hierarchy.
- **Computational Profiles:** The set $\mathcal{P}_{comp}$ characterizes the computation capabilities, allowing the cost model to estimate execution time for compute-bound operations.

By parsing these DSL parameters, the synthesizer constructs the logical capacity constraints $C(u, v)$ used in the verification phase. Figure 10 shows a DSL example for a cluster topology with two heterogeneous devices. It starts with the number of nodes using the `nodenum` attribute (line 1), then details each node's identifier and GPU count. Connection types (`direct` for intra-node and `switch` for inter-node) are specified in lines 2-7, along with connection parameters like bandwidth and the number of links (`bandwidth`, `numlinks`).

```
1 define nodenum = 2
2 define node1 = 2
3 define bandwidth = {direct => [((0,1)->(25,2))
      ,((1,0)->(25,2))]}
4 define node2 = 4
5 define bandwidth = {switch => [(<(0,1),(2,3)
      >->(16,1))],direct => [((0,1)->(25,2))
      ,((1,0)->(25,2)),((2,3)->(25,2)),((3,2)
      ->(25,2))]}
6 define internode = (node1,node2)
7 define interbandwidth = {switch => [(<(0,1)
      ,(0,1,2,3)>->(8,1))]}
```

Figure 10: Example program in HeteCCL's DSL.

B Global Notation Reference

This appendix consolidates the mathematical notations used across the HeteCCL framework. To ensure a rigorous formulation, we categorize these symbols into three logical phases corresponding to the system workflow:

1. **Cluster Inputs & DSL (Part I):** Symbols that quantify the raw physical attributes of the heterogeneous cluster, including node scale, hardware bandwidth profiles, and user-defined chunking granularity. These parameters serve as the initial inputs to the interpreter.
2. **Modeling & Theory (Part II):** Notations representing the graph-theoretic abstraction of the topology (G) and the derived cost model coefficients. This set defines the logical constraints and performance bounds (e.g., logical capacity $C(u, v)$) derived from the physical inputs.
3. **Synthesis (Part III):** Variables and sets used within the CEGIS algorithmic loop. These symbols formalize the state space (e.g., time steps $\mathcal{T}$, chunk sets $\mathcal{K}$) and the constraint solving process (e.g., counterexamples $\mathcal{E}$) required to synthesize the optimal schedule $\mathcal{S}_{opt}$.

Table 2 presents the detailed definitions and descriptions for each category.

Table 2: Global Notation System of HeteCCL.

Symbol	Description
	Part I: Cluster Inputs & DSL
$N_{cluster}$	Total number of nodes (Renamed from K).
N_i	Number of GPUs in the i-th machine.
S_c	User-defined chunk size (e.g., 1 MB).
$\mathcal{B}, \mathcal{L}$	Sets of raw hardware bandwidth and latency.
$\mathcal{P}_{comp}$	Computational profiles (Renamed from $\mathcal{C}$).
$\mathcal{B}_i^{\text{intra/inter}}$	Internal/External bandwidths of machine i.
	Part II: Modeling & Theory
$G = (V, E)$	Topology graph (V: GPUs, E: links).
$C(u, v)$	**Logical Capacity** of link (u, v) (Max parallel chunks).
C_g	Capacity limit for shared resource group g.
τ_{chunk}	**Unit Cost**. Time to transmit one chunk.
α, β, γ	Cost model coefficients (Lat, BW, Comp).
$M_{trans/comp}$	Effective Data Volume (Transfer/Compute).
$T_{cost/ideal}$	Predicted/Ideal execution time.
	Part III: Synthesis (CEGIS)
$\mathcal{K}, \mathcal{T}$	Sets of Chunks and Time steps.
N_{step}	Total steps (Optimization Objective).
t, t_{check}	Time step index and Checkpoint step.
$\mathcal{E}$	Set of **Counterexamples** (violations).
$pre, post$	Initial and Target chunk states.
$\mathcal{S}_{opt}$	The final optimal schedule.

C Heterogeneous Topology in the V100 Cluster

Figure 12 illustrates the $(4+8)$ heterogeneous topology in the V100 cluster.

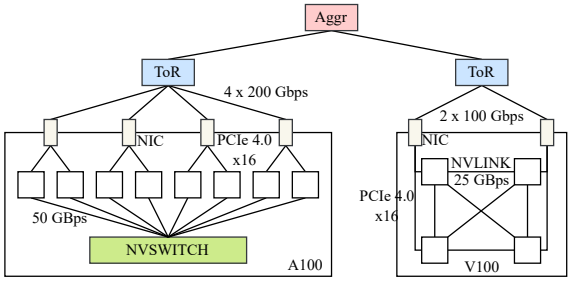


Figure 11: Heterogeneous Topology

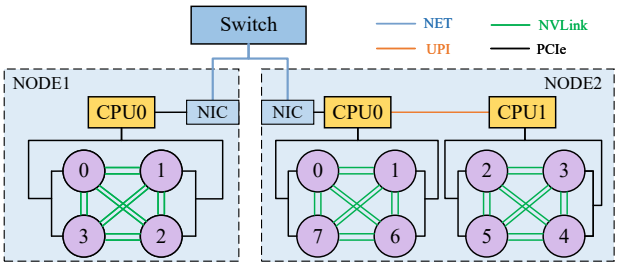
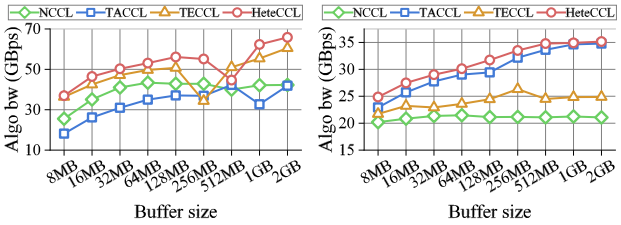


Figure 12: Detailed link diagram for the heterogeneous cluster, example: 4+8.

D Homogeneous Setup

To validate the generality of the HeteCCL synthesis method, specifically its ability to achieve broad performance improvements across various topologies, we conducted synthesis evaluations on a high-performance homogeneous A100 cluster and compared the results with existing methods. As shown in Figure 13, HeteCCL consistently demonstrates superior bandwidth performance across different scales of homogeneous cluster setups compared to other methods (including the state-of-the-art method TE-CCL).



(a) 8-GPUs (b) 16-GPUs

Figure 13: Bandwidth performance comparison of methods in a homogeneous GPU cluster setup.

E Robustness

Asymmetric heterogeneous topology in failure scenarios. To demonstrate HeteCCL's robustness and fault tolerance, we created four failure scenarios by disabling one GPU in a 4-GPU node, one GPU in an 8-GPU node, one GPU in both 4-GPU and 8-GPU nodes, and one GPU in each of the 4-GPU nodes. Even under these extreme asymmetric topologies, HeteCCL effectively synthesizes optimal collective communication algorithms. As shown in Figure 15, HeteCCL increases bandwidth by up to 6.5x over TACCL and 3.25x over NCCL.

To further demonstrate HeteCCL's robustness, we created a scenario with degraded transmission rates for server connections by reducing the link capacity of one server to 18Gbps. In this scenario, HeteCCL maintained efficient bandwidth, achieving 2.61x the bandwidth of NCCL and 1.92x that of TACCL, as shown in Figure 16.

F Implementation

The implementation of the HeteCCL synthesizer consists of three key components, comprising around 4K lines of code. First, a connection-oriented domain-specific language (DSL) is developed to define fine-grained topologies of heterogeneous clusters, offering users flexible topology representation. The DSL is embedded within a Python module, and a lightweight compiler is implemented to convert DSL inputs into an intermediate representation (abstract syntax tree). Second, a counterexample-guided inductive synthesis algorithm is integrated with the Z3 solver to efficiently process parsed DSL expressions, constructing mathematical models, sets, and topological data structures based on DSL-defined constraints to derive key results during synthesis. Lastly, we implemented a translation mechanism to read chunk operation steps from JSON files, accurately allocate tasks across ranks and threads, and generate XML files compatible with the communication backend for execution.

G Experiments on A100

To validate the scalability of our approach, we evaluated HeteCCL in a heterogeneous topology with higher communication bandwidth and GPU performance (A100-80GB-SXM4 system, GPU throughput peak of 300 GBps, NVLINK bandwidth of 600 GBps). The experimental results demonstrate that HeteCCL continues to outperform existing methods in such high-performance environments, showing a 1.78x improvement over TACCL and a 1.95x improvement over NCCL, as shown in Figure 17.

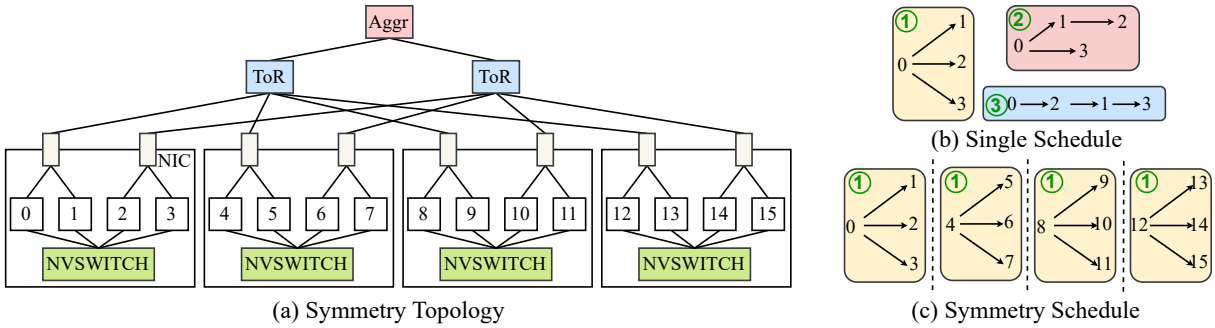


Figure 14: Symmetry-guided Pruning

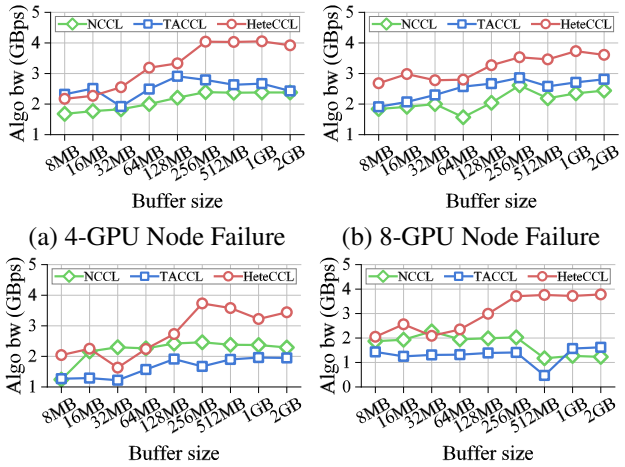


Figure 15: Bandwidth comparison of HeteCCL, TACCL, and NCCL algorithms in asymmetric heterogeneous topologies during cluster failures.

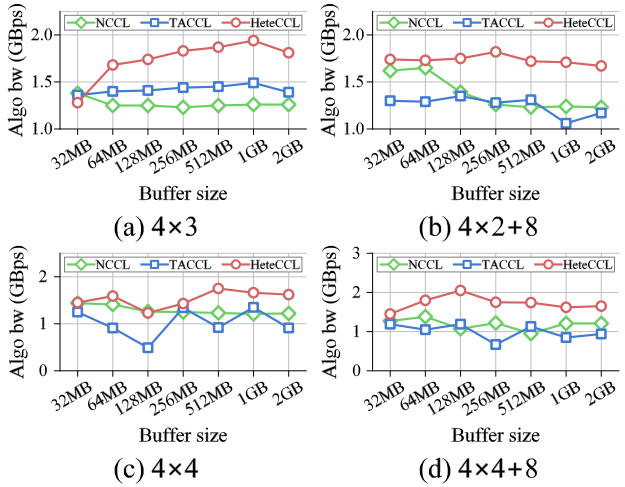


Figure 16: Bandwidth comparison of HeteCCL, TACCL, and NCCL algorithms in heterogeneous clusters with rate-limiting nodes.

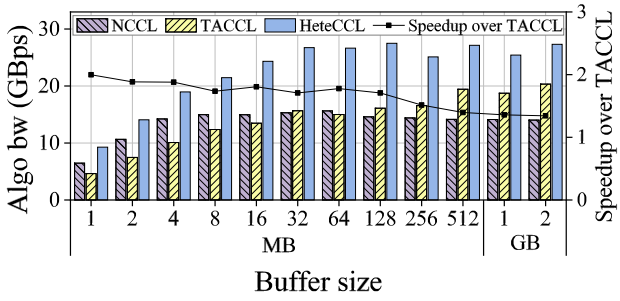


Figure 17: Algorithm communication bandwidth evaluation in higher-performance heterogeneous systems (4+8 A100).

H CEGIS Algorithm Implementation

Algorithm 1 details the implementation of our Counterexample-Guided Inductive Synthesis workflow. The process begins by initializing a schedule $\mathcal{S}_{init}$ using a GREEDYHEURISTIC, which prioritizes link saturation but may violate capacity constraints. The VERIFY function checks the candidate schedule against the physical constraints defined in §4.1 (Eqs. 7-9) and returns a set of counterexamples $\mathcal{E}$.

In the main loop, the SMTSOLVER synthesizes a valid partial schedule that satisfies the accumulated constraints in $\mathcal{E}$. If the verification phase returns a non-empty set of errors, we identify the earliest failing time step t_{check}. The REVISEFULL function then acts as a crucial optimization step: it freezes the valid history before t_{check}, imposes new constraints to block the specific violation, and greedily fills the remaining capacity at the checkpoint step to guide the solver toward high-bandwidth solutions in the next iteration. This cycle repeats until $\mathcal{E}$ is empty, ensuring the returned $\mathcal{S}_{opt}$ is both physically valid and efficient.

I Composite Algorithm Synthesis

For composite collective operations such as ALLREDUCE, HeteCCL adopts a decomposition strategy to maintain syn-

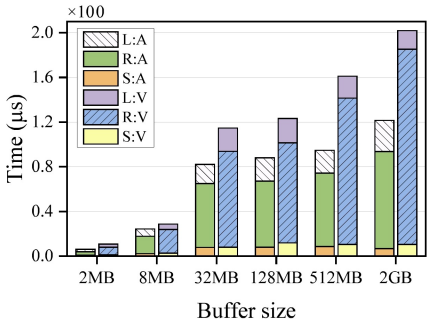


Figure 18: Execution time clocks of different operations for set-communication operators across generations of GPUs. L represents the cost of data loading, R denotes the cost of executing the REDUCE operation, S refers to the cost of data storage, V and A stand for the V100/A100 series GPUs.

Algorithm 1 Counterexample-Guided Inductive Synthesis (CEGIS)

Require: Topology graph G, Initial state pre, Target state $post$
Ensure: Optimal Schedule $\mathcal{S}_{opt}$

$\mathcal{S}_{init} \leftarrow \text{GREEDYHEURISTIC}(pre, post)$
$\mathcal{E} \leftarrow \text{VERIFY}(\mathcal{S}_{init}, G)$
$\mathcal{S}_{curr} \leftarrow \mathcal{S}_{init}$
while true do
  $\mathcal{S}_{cand} \leftarrow \text{SMTSOLVER}(G, \mathcal{S}_{curr}, \mathcal{E}, pre, post)$
  if $\text{ISSAT}(\mathcal{S}_{cand})$ **then**
    $\mathcal{E} \leftarrow \text{VERIFY}(\mathcal{S}_{cand}, G)$
    if $\mathcal{E} = \emptyset$ **then**
      $\mathcal{S}_{opt} \leftarrow \text{FINALIZE}(\mathcal{S}_{cand})$
      break
    else
      $t_{check} \leftarrow \text{GETEARLIESTSTEP}(\mathcal{E})$
      $\mathcal{S}_{curr}, \mathcal{E} \leftarrow \text{REVISEFULL}(\mathcal{S}_{cand}, t_{check}, pre, post)$
  else
    return failure
return $\mathcal{S}_{opt}$

thesis efficiency. Rather than synthesizing the entire operation as a monolithic block, we decompose ALLREDUCE into a REDUCE-SCATTER phase followed by an ALL-GATHER phase.

In the reduction phase, HeteCCL ensures load balancing by distributing reduction chunks across all available GPUs. The mapping of a specific chunk k to its reduction target GPU u is determined by a modulo distribution:

$$u = k \bmod |V| \tag{7}$$

where $|V|$ denotes the total number of GPUs in the cluster. This ensures that each chunk is fully reduced on a definitive root node.

Subsequently, the process transforms into a MULTIPLE-ROOT BROADCAST problem (equivalent to ALL-GATHER). During this phase, we impose a strict sequential constraint to prevent resource hazards. Specifically, a GPU cannot perform multiple reduction additions at the same memory location simultaneously. We encode this as a mutual exclusion constraint in the SMT model:

$$\forall group, src \in V: \quad \sum_{k \in \mathcal{K}} Send(k, src, dst) \leq 1 \tag{8}$$

This constraint ensures that for any given step, each GPU processes at most one chunk for a specific reduction group, guaranteeing correctness during the inverse execution flow. By adding these constraints to the general CEGIS workflow described in §5.2, HeteCCL synthesizes optimal schedules for composite operations that satisfy both topological and algorithmic dependencies.

J Synthesized Artifact Generation

Upon convergence, HeteCCL translates the logical schedule $\mathcal{S}_{opt}$ into executable primitives. The output is serialized into an XML-based intermediate representation that describes the precise sequence of operations for each GPU.

To map these operations to the hardware execution model, HeteCCL assigns each step of the synthesized algorithm to a GPU thread block. We utilize explicit dependency graphs to enforce the execution order, ensuring that a *Send* operation is only triggered after its prerequisite data chunks have arrived. This abstraction decouples the synthesis logic from the underlying runtime, allowing the generated algorithms to be deployed on diverse backend implementations.

K Theoretical Analysis

We analyze this trade-off by modeling the transmission of a large message M as a sequence of N algorithmic iterations (or batches), where each batch transfers a data slice of size S_{batch} (i.e., $M = N \cdot S_{batch}$).

Consider a critical path of length H hops, where l_h denotes the physical propagation latency of the h-th link. The total completion time T consists of the pipeline startup time (latency for the first batch to reach the destination) plus the steady-state serialization time for the remaining data.

In an ideal sequential model (where the full bandwidth B pushes the first chunk), the startup time is minimized. The total time T_{seq} is:

$$T_{seq} \approx \sum_{h=1}^{H} l_h + (H-1)\frac{S_{batch}}{B} + \frac{M - S_{batch}}{B} \tag{9}$$

where the last term $\frac{M-Sbatch}{B}$ represents the steady-state streaming time required for the bulk of the data to traverse the bottleneck.

HeteCCL divides the physical link bandwidth B into K logical sub-links to enable parallel chunk transmission. This

reduces the effective bandwidth for each chunk to B/K and amplifies the serialization delay at each hop during the startup phase. The completion time $T_{HeteCCL}$ is:

$$T_{HeteCCL} \approx \sum_{h=1}^{H} l_h + (H-1)\frac{S_{batch}}{B/K} + \frac{M - S_{batch}}{B} \tag{10}$$

The difference lies solely in the startup term: HeteCCL introduces a parallelization penalty of $(H-1)\frac{S_{batch}}{B}(K-1)$. Crucially, this penalty applies only to the first batch required to fill the pipeline. Once the pipeline is saturated, subsequent batches (2 to N) flow through the bottleneck link at the full aggregate bandwidth B, regardless of the chunking granularity. For LLM training, the total data volume M is massive, implying a large number of iterations N (i.e., $M \gg S_{batch}$). As $N \to \infty$, the steady-state term $\frac{M}{B}$ dominates the constant startup penalty.

$$\lim_{M \to \infty} \frac{T_{HeteCCL}}{T_{seq}} = 1 \tag{11}$$

This confirms that HeteCCL achieves asymptotic bandwidth optimality. The latency overhead is a one-time startup cost that is effectively amortized by the long-running transmission loop. Moreover, by ensuring the bottleneck link remains continuously saturated, HeteCCL synthesizes a schedule that is throughput-equivalent to any near-optimal fine-grained schedule. The missed schedules are simply alternative distributions of the same total idle time, offering no improvement in optimality. Thus, our search space reduction effectively eliminates only redundancy, not optimality.