SyCCL: Exploiting Symmetry for Efficient Collective Communication Scheduling

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[†], Ennan Zhai[†]

[†]*Alibaba Cloud* [‡]*Tsinghua University*

Abstract

The performance of collective communication schedules is crucial for the efficiency of machine learning jobs and GPU cluster utilization. Existing open-source collective communication libraries (such as NCCL and RCCL) rely on fixed schedules and cannot adjust to varying topology and model requirements. State-of-the-art collective schedule synthesizers (such as TECCL and TACCL) utilize Mixed Integer Linear Program for modeling but encounter search space explosion and scalability challenges. In this paper, we propose SyCCL, a scalable collective schedule synthesizer that aims to synthesize near-optimal schedules in tens of minutes for production-scale machine-learning jobs. SyCCL leverages collective and topology symmetries to decompose the original collective communication demand into smaller sub-demands within smaller topology subsets. SyCCL proposes efficient search strategies to quickly explore potential sub-demands, synthesizes corresponding sub-schedules, and integrates these sub-schedules into complete schedules. Our 32-A100 testbed and production-scale simulation experiments show that SyCCL improves collective performance by up to 127% while reducing synthesis time by 2 to 4 orders of magnitude compared to state-of-the-art efforts.

CCS Concepts

• **Networks** → **Data path algorithms**; **Data center networks**; • **Computing methodologies** → **Machine learning**.

Keywords

Collective Communication, Deep Learning, Communication Scheduling

ACM Reference Format:

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, Ennan Zhai. 2025. SyCCL: Exploiting Symmetry for Efficient Collective Communication Scheduling. In *ACM SIGCOMM 2025 Conference (ACM SIGCOMM '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 17 pages. https://doi.org/https://doi.org/10.1145/3718958.3750499

*Both authors contributed equally to this paper.

SIGCOMM '25, September 8–11, 2025, Coimbra, Portugal

ACM ISBN 979-8-4007-1524-2/25/09.
https://doi.org/https://doi.org/10.1145/3718958.3750499

1 Introduction

Collective communication libraries (CCLs), such as NCCL [11] and RCCL [14], implement efficient inter-GPU communication in distributed machine learning (ML) training and inference. The performance of collective communication is critical, as it directly affects the efficiency of training and inference. For example, [22] reports that the time spent on collective communication accounts for >30% of the time when training GPT-22B [16] and LLaMa-7B [35] models.

The performance of collective communication heavily relies on the *communication schedule*, which defines the data transfer among GPUs to satisfy communication demands.

Various factors can impact the performance of collective communication schedules, including the collective itself (*e.g.,* communication pattern, data size, and participant GPUs) and the inter-GPU topology (*e.g.,* connectivity, bandwidth, and latency). For example, for small data sizes (*e.g.,* <10MB), network latency dominates, and thus schedules that minimize transmission hops perform better. In contrast, for large sizes (*e.g.,* >1GB), optimizing bandwidth is essential.

Current CCLs, nevertheless, employ fixed schedules (*e.g.,* ring and double binary tree [1]) for all collectives and topologies. For example, NCCL utilizes fixed ring schedules for `AllGather`, where each server connects GPUs in local chains and links these chains to form complete rings. This fixed approach can lead to underutilized bandwidth for large data sizes and high latency for small sizes.

In practice, there are numerous inter-GPU topologies with various connections, including intra-server links (*e.g.,* PCIe and NVLink [13]) and inter-server network links (*e.g.,* Ethernet and Infiniband). Additionally, ML models [19, 26, 37, 41] and parallelism configurations [20, 25, 34] can vary widely. This leads to varying collective communication calls, with different data sizes (from a few bytes to several gigabytes), communication patterns (*e.g.,* `AllReduce` and `AllGather`), and GPU participants with various connections. In one of our production clusters, the fixed ring schedule used by NCCL leads to 10.6% bandwidth wastage for large sizes and 4× latency increase for small sizes (see §2.1 for details). To optimize communication performance, synthesizing schedules tailored to topologies and collectives is critical.

Recent efforts [17, 28, 33] model the schedule synthesis as a Mixed Integer Linear Program (MILP) problem and employ solvers to produce a near-optimal schedule automatically. While these methods outperform fixed schedules, they struggle with

larger topologies. For example, TACCL [33] fails to synthesize an `AllGather` schedule on a 128-GPU topology within eight hours. To address scalability issues, TECCL [28] divides the collective into multiple independent time intervals and applies a greedy heuristic to solve the schedule for each interval. This approach accelerates the process but sacrifices accuracy (*e.g.,* 20% performance loss [28]).

This paper presents SyCCL, a scalable collective communication schedule synthesizer that can propose near-optimal schedules in minutes for production-scale ML jobs.

We identify that the culprit of the scalability issue in previous solutions is encoding the entire collective and topology into a holistic formula. While in reality, symmetries commonly exist in the collective and topology. For example, all GPUs in a collective may send or receive equal amount of data, and datacenter topologies (*e.g.,* Clos [6] and Multi-rail [2]) are also inherently symmetric. Thus, the optimal plan for one symmetric group can apply to others.

The challenge lies in effectively leveraging these symmetries. MILP solvers may not efficiently detect complex symmetric structures. Conversely, directly encoding them as MILP constraints may further complicate the problem and degrade solver performance.

To address this, SyCCL introduces a key concept called *sketch*. A sketch divides the original collective communication demand into smaller sub-demands across smaller topology subsets (§3). Each sub-demand can be satisfied using various sub-schedules. The sub-schedules for these sub-demands together form a complete schedule. This approach improves scalability by solving smaller problems individually and combining the results. In addition, by leveraging topology and collective symmetries, the sub-demands and sub-schedules across isomorphic subsets are consistent, greatly reducing the overall search space.

SyCCL's synthesis consists of two phases. First, SyCCL explores potential sketches based on the input topology and collective (§4). SyCCL designs an efficient method to search for sketches and combines the generated sketches to maximize bandwidth usage. Second, SyCCL employs an MILP modeling approach to synthesize near-optimal sub-schedules and combines them into complete schedules (§5). Additionally, SyCCL enhances accuracy through two-step synthesis, and accelerates synthesis using isomorphism and parallelism.

Our 32-A100 experiments show that SyCCL achieves up to 91% improvement in schedule performance compared to TECCL and 108% improvement compared to NCCL. Our production-scale H800 simulation experiments further show that SyCCL significantly improves schedule performance by up to 127% while decreasing synthesis time by 2 to 4 orders of magnitudes compared to TECCL. In addition, SyCCL improves end-to-end model training performance by up to 11.2% compared to NCCL (§7).

Ethics. This work does not raise any ethical issues.

2 Background and Motivation

Optimizing collective communication performance is non-trivial. In this section, we first introduce the background of collective communication (§2.1). Then we explain the search space explosion challenge in schedule synthesis (§2.2) and why the state-of-the-art efforts are not scalable (§2.3).

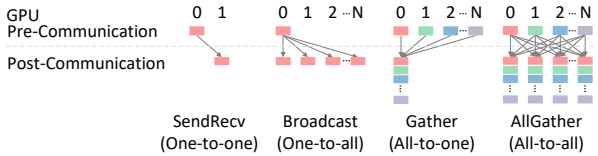


Figure 1: Collectives are categorized into four types: one-to-one, one-to-all, all-to-one, and all-to-all.

Variable	Description
$\mathcal{V}$	Set of GPUs.
C	Set of chunks.
s	Chunk size
F_s: $C \to \mathcal{V}$	Mapping each chunk $c \in C$ to the GPU it is initially on.
F_d: $C \to \mathcal{P}(\mathcal{V})$	Mapping each chunk $c \in C$ to the GPUs that demand it. $\mathcal{P}(\mathcal{V})$ denotes the power set of $\mathcal{V}$, *i.e.,* $F_d(c) \subseteq \mathcal{V}$.
$r \in \{0, 1\}$	Whether chunks are reduced on destination GPUs.

Table 1: Collective denotations.

2.1 Background: Collective Communication

Large-scale ML training and inference jobs exchange data between GPUs using collective communication.

Collective communication. As shown in Table 1, a collective involves multiple GPU participants $\mathcal{V}$ and multiple data chunks C of the same size s. Each GPU $v \in \mathcal{V}$ can act as a source, a destination, or both. We use two mapping functions F_s and F_d to denote the communication patterns. A chunk $c \in C$ is initially at a source GPU g if $F_s(c) = g$, and is required by a destination GPU g if $g \in F_d(c)$.

Collective communication patterns are classified into four categories: one-to-one, all-to-one, one-to-all, and all-to-all [23]. (1) One-to-one (`SendRecv`) involves a chunk c which is initially on the source v_s ($F_s(c) = v_s$), and is demanded by the destination v_d ($F_d(c) = \{v_d\}$). (2) One-to-all involves a source v_s sending a chunk to all GPUs (`Broadcast`), or sending a different chunk to each other GPU (`Scatter`). (3) All-to-one (`Gather`/`Reduce`) involves a destination v_d receiving chunks from all GPUs. (4) In All-to-all, each GPU acts as both a source and a destination. A GPU may start with one chunk and gather/reduce all chunks from others (`AllGather`/`AllReduce`), or start with $|\mathcal{V}|$ chunks and gather/reduce a chunk from each GPU (`Alltoall`/`ReduceScatter`).

Limitations in fixed collective communication schedules. The performance of collective communication heavily depends on the schedule, *i.e.,* how chunks are transferred among GPUs. Current CCLs employ fixed schedules. Figure 2 shows NCCL's ring schedule for `AllGather`, where 8 GPUs within each server are connected in a chain, and these chains are linked to form a complete ring. However, fixed ring schedules do not achieve optimal performance in this case for two reasons. First, they maintain a fixed bandwidth ratio of 7:1 between intra-server and inter-server links. In our H800 cluster, the intra-server NVLink bandwidth is 180GBps, while each server scales out with eight 400Gbps NICs, resulting in an actual ratio of 3.6:1. As a result, the NVLink becomes the bottleneck, leading to 3.4/7=48.5% of the inter-server network bandwidth being wasted and an average bandwidth wastage of 48.5%/(3.6+1)=10.6%. Second, transfers require $|\mathcal{V}|$-1 hops to complete, resulting in high latency, especially for small sizes where latency matters. Our testbed evaluation shows that NCCL's performance at small sizes can be 4× worse than that of synthesized schedules.

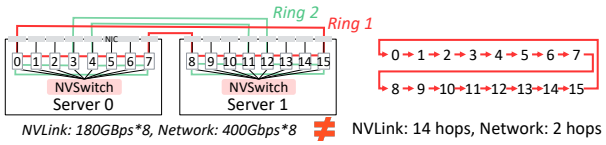


Figure 2: The fixed *ring* schedule in NCCL performs poorly on our production cluster.

Given the limitations, it is critical to synthesize optimal schedules based on network topologies and collectives.

2.2 Search Space for Collective Schedules

The potential schedules for a collective can be massive under a large-scale GPU cluster. First, a chunk may take different *routes* to reach its destinations. Once a GPU receives a chunk, it can send it to any other GPU, leading to numerous routing possibilities. Second, a collective may involve multiple chunks. When two chunks are sent over the same link, their transmission *order* matters. The possible ordering combinations for n chunks exceed $n!$. Third, chunks can be *sliced* into smaller pieces, which take different paths to exploit bandwidth. This further expands the routing and ordering choices.

As the number of GPUs, the chunk sizes, and the topology complexity increase, the search space for schedules explode.

2.3 Limitations of Existing Synthesizers

The state-of-the-art schedule synthesizers [17, 28, 33] model schedule synthesis as a Satisfiability Modulo Theory (SMT) or a Mixed-Integer Linear Programming (MILP) problem. They encode the collective, topology, and schedule as constraints, intending to minimize communication time. However, as the scale of networks and collectives increases, direct encoding of the whole problem faces the challenge of search space explosion. For example, it takes >24 hours for SCCL [17] to synthesize a 16-GPU `AllGather` schedule.

To handle this huge search space, TACCL [33] splits schedule synthesis into three smaller steps: routing, ordering and exact scheduling. It also allows users to apply rotational symmetry constraints to further reduce the search space. However, directly encoding massive symmetry constraints into the model may further complicate the MILP formulation and slowdown the solver instead. As a result, TACCL fails to create an `AllGather` schedule on a 128-GPU topology within eight hours [28].

TECCL [28] models schedule synthesis as an MILP problem by setting a fixed epoch duration and representing the schedule as discrete events within epochs. Within modern GPU clusters with various interconnections, TECCL struggles to find an appropriate epoch duration that can accurately model transmission on interconnections with different performances (See Appendix A for more details). To address scalability issues, TECCL divides communication into multiple time intervals and solves each separately with greedy heuristics. As a result, this technique is not globally optimal (*e.g.,* it can lead to 20% performance loss according to [28]).

On the other hand, new underlying network architectures [12, 30] and optimization techniques [7, 9, 15] are continually emerging. Researchers frequently adjust model hyper-parameters (*e.g.,* batch size and parallelisms) to enhance performance. These factors lead to frequent changes in collective communication calls, highlighting the necessity for quickly synthesizing effective schedules.

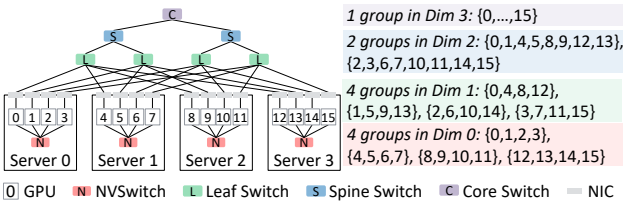


Figure 3: Example of a multi-rail GPU cluster topology. 16 GPUs are distributed across 4 servers.

Variable	Description
$\mathcal{V}$	Set of GPUs in the topology
$\mathcal{V}_0$	Set of GPUs, NICs, and switches in the topology ($\mathcal{V} \subseteq \mathcal{V}_0$)
$\mathcal{E}$	Set of links ($\mathcal{E} = \{(v_1, v_2) \mid v_1, v_2 \in \mathcal{V}_a\}$)
α_e	Latency of link $e \in \mathcal{E}$
β_e	$1/\beta_e$ is the bandwidth of link $e \in \mathcal{E}$
$\mathcal{D}$	Set of dimensions ($\mathcal{D} = \{0, 1, \ldots\}$)
$\mathcal{G}_d$	Set of groups in dimension $d \in \mathcal{D}$
$\mathcal{V}_{d,g}$	Set of GPUs in group $g \in \mathcal{G}_d$ in dimension $d \in \mathcal{D}$

Table 2: Topology detonations.

3 Insight and Design Overview

We first introduce our insight, *i.e.,* optimal schedules present certain characteristics (§3.1). We then introduce *sketch*, which leverages this insight to filter sub-optimal schedules and reduce search space. §3.3 presents the system overview.

3.1 Insight

Modern GPU clusters and collectives both exhibit a high degree of symmetry. Based on this, our key insight is that a well-performing schedule should incorporate certain symmetrical properties.

Observation 1: Topology symmetry. Modern GPU clusters leverage multi-dimensional symmetric networks to connect GPUs. Figure 3 shows a typical multi-rail topology, where GPUs with the same intra-server index (*e.g.,* GPUs 0, 4, 8, and 12) are connected to the same leaf switch. The connections between GPUs fall into four categories. Within each server, GPUs are connected by an NVSwitch. Between servers, GPUs are connected through three tiers of network switches, *i.e.,* leaf, spine, and core. The connections are symmetric. Specifically, each server has the same number of GPUs with identical connections, and switches at the same tier connect the same number of GPUs with identical links.

We formalize the topology in Table 2. We introduce a **dimension** to represent a type of inter-GPU connection and denote the set of dimensions as $\mathcal{D}$. In Figure 3, the four different connections correspond to four dimensions. Within each dimension $d \in \mathcal{D}$, GPUs are organized into **groups** according to their connectivity. The set of groups in dimension d is denoted as $\mathcal{G}_d$ and the set of GPUs in group $g \in \mathcal{G}_d$ is denoted as $\mathcal{V}_{d,g}$. Each GPU belongs to one group at each dimension. If two GPUs are reachable through direct connection, they are assigned to the same group in the corresponding dimension. In Figure 3, dimension 0 (*i.e.,* intra-server interconnections) contains four groups (*i.e.,* servers). Groups at the same dimension (*e.g.,* four servers) exhibit isomorphic characteristics, *i.e.,* their topologies are identical. Given a topology, SyCCL automatically extracts the dimensions and groups according to connectivity and connection performance. We provide other topology examples (*e.g.,* Clos) in Appendix B.

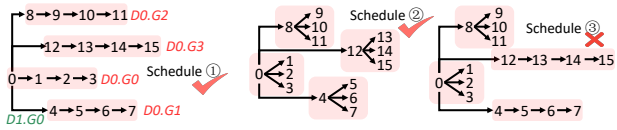


Figure 4: [Insight 1] Schedules ① and ② comprise consistent communication across isomorphic groups, while ③ lacks this consistency and is sub-optimal.

Observation 2: Collective symmetry. In one-to-all or all-to-one collectives, all GPUs, except for the source or destination, present identical communication demands. Furthermore, all-to-all collectives (*e.g.,* `AllGather`) can be decomposed into isomorphic all-to-one/one-to-all collectives (*e.g.,* `Broadcast`), each with independent chunks of the same size.

Insight: Optimal schedules consist of consistent sub-schedules across isomorphic GPU groups. Collective communication can be viewed as comprising identical communication sub-demands across isomorphic topology subsets (*i.e.,* GPU groups). Consequently, these groups should exhibit identical sub-schedules for optimal performance. Otherwise, the load becomes imbalanced in some groups.

Figure 4 shows three schedules for a 16-GPU `Broadcast`. In all cases, GPU 0 broadcasts its chunk to GPUs 4, 8, and 12 through group 0 in dimension 1 (denoted as D1.G0), and GPUs 0, 4, 8, and 12 broadcast the chunk within the servers in dimension 0. Schedules ① and ② use a single schedule type (ring or tree) across four groups in dimension 0, while ③ utilizes two types of schedules, *i.e.,* ring for groups 1 and 3, and tree for groups 0 and 2. Since the connections and communication sub-demands are consistent, the optimal sub-schedules should also be identical. Thus, either the ring or tree is preferable, making either ① or ② better than ③. Therefore, ③ is sub-optimal and should be filtered.

3.2 SyCCL Sketch

The above insight indicates what an optimal schedule looks like. The challenge lies in applying this to synthesize an optimal schedule. To this end, SyCCL introduces a concept called **sketch**. As shown in Figure 5, a sketch divides the collective demand into smaller sub-demands in different dimensions and time stages. Each sub-demand can be satisfied by various potential sub-schedules. The sub-schedules together create the complete schedules to satisfy the original demand. By utilizing the sketch, synthesizing the optimal schedule becomes a two-phase process: (1) searching for sketches (*i.e.,* combinations of sub-demands), (2) for each sketch, determining the possible sub-schedules for each sub-demand and combining them into a schedule candidate, ultimately selecting the optimal schedule from all candidates.

This approach significantly reduces the search space in two ways. First, since sub-demands across isomorphic groups are consistent, we can eliminate inferior sketches in phase (1). Second, the optimal sub-schedules for these consistent sub-demands are also consistent, allowing us to avoid redundant calculations in phase (2). In addition, the sketch breaks down a complex optimization problem into multiple smaller ones that are solved in parallel, improving accuracy and efficiency.

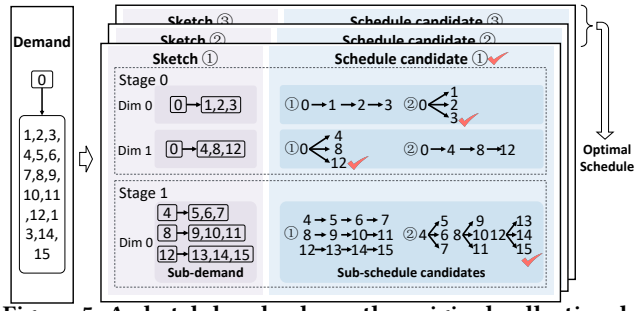


Figure 5: A sketch breaks down the original collective demand into sub-demands. Optimal sub-schedules for all sub-demands construct an optimal schedule.

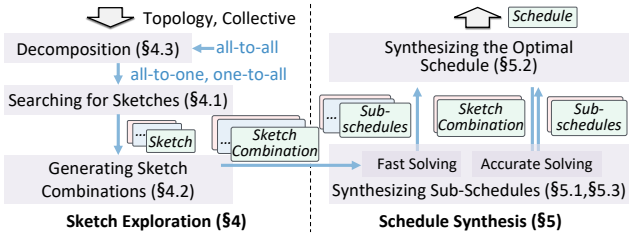


Figure 6: SyCCL workflow.

Example. Figure 5 shows an example for synthesizing a 16-GPU `Broadcast` schedule in the topology in Figure 3. Sketch ① divides the original demand (*i.e.,* GPU 0 sends its chunk to all other 15 GPUs) into three sub-demands in two stages. In stage 0, GPUs 1, 2, and 3 receive GPU 0's chunk from dimension 0, while GPUs 4, 8, and 12 receive it from dimension 1. In stage 1, remaining GPUs obtain it from dimension 0. Each sub-demand can be met by various sub-schedules. SyCCL synthesizes the optimal sub-schedule (marked with a check) for each sub-demand, combines them into schedules, and selects the best one. The three marked sub-schedules form schedule ② in Figure 4.

3.3 SyCCL Overview

As shown in Figure 6, SyCCL synthesizes the optimal schedule in two phases. In the first phase, SyCCL explores potential sketches based on the input collective and topology (§4). For one-to-all and all-to-one collectives, SyCCL searches for sketches and applies the insight from §3.1 to eliminate inferior ones (§4.1). Since a single sketch may not fully utilize the bandwidth, SyCCL combines multiple to maximize bandwidth usage, producing sketch combinations (§4.2). For all-to-all collectives, which can be decomposed into isomorphic one-to-all collectives, SyCCL searches for sketches for one one-to-all collective, replicates them according to the topology symmetry to derive all-to-all sketches, and combines them to optimize bandwidth (§4.3).

In the second phase, SyCCL synthesizes the optimal schedule given the generated sketch combinations (§5). SyCCL employs an MILP solver to generate the optimal sub-schedules for sub-demands in each combination (§5.1). These sub-schedules are merged into complete schedules. Then, SyCCL utilizes a schedule simulator to evaluate the performance of candidate schedules and selects the one with the best performance (§5.2). Furthermore, SyCCL designs

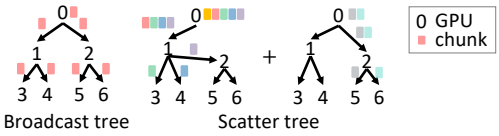


Figure 7: A `Broadcast` schedule is always a tree, while a `Scatter` schedule can always be split into trees.

a two-step solving method to improve accuracy and leverages isomorphism and parallelism to accelerate synthesis (§5.3).

4 Sketch Exploration

This section describes how SyCCL efficiently explores sketches based on the insight in §3.1. We begin with all-to-one and one-to-all collectives, while one-to-one is a specific case of one-to-all and is not discussed separately. First, we focus on scenarios where a chunk cannot be split and must follow a single path. We design an efficient sketch search method (§4.1). Next, we address real-world scenarios where chunks can be split and sent via multiple paths to optimize bandwidth usage. We convert each generated sketch into a combination of sketches, with each sketch handling part of the chunk (§4.2). Finally, we adapt the sketch exploration method to all-to-all collectives (§4.3).

4.1 Searching for Sketches

In this section, we first analyze the communication for one-to-all and all-to-one collectives. We show that their schedules follow a tree structure. Based on this, our sketch search method starts from a source GPU and explores potential paths (combinations of sub-demands) to reach other GPUs along a tree. Given the numerous possible combinations, we design efficient pruning strategies to reduce the search space.

Tree structure for schedules. We focus on one-to-all collectives (*i.e.,* `Broadcast` and `Scatter`) as all-to-one collectives (`Reduce` and `Gather`) are their inverses. When handling a single, indivisible chunk, both `Broadcast` and `Scatter` schedules can be represented with trees, where each GPU has only one predecessor. As shown in Figure 7, in a `Broadcast` schedule, a chunk is sent along a tree, with each GPU receiving it from a single predecessor. To prevent bandwidth waste, a GPU should not receive the same chunk more than once, confirming that the `Broadcast` schedule is indeed a tree. In `Scatter`, $|\mathcal{V}_g|$ chunks are sent from the root, with each node retaining its desired chunk and distributing others to its successors. Although a GPU may have several predecessors, the overall schedule can still be seen as a collection of trees.

Sketch definition. Sketch ① in Figure 5 shows a sketch example. A sketch, as detailed in Table 3, breaks down collective communication into K stages. A stage k comprises communication sub-demands $\mathcal{R}_{k,d,g}$ for each dimension d and group g. A sub-demand $\mathcal{R}_{k,d,g}$ signifies that the destination GPUs $\mathcal{V}^r_{k,d,g}$ expect to receive chunks from source GPUs $\mathcal{V}^s_{k,d,g}$. Since the schedules follow a tree structure, each GPU v either starts as a source (*i.e.,* $v \in \mathcal{V}^s_{0,d,g}$) or acts as a destination only once (*i.e.,* $v \in \mathcal{V}^r_{k,d,g}$). All sub-demands

Variable	Description
K	A sketch consists of K stages ($K \in \mathbb{Z}^+$)
$\mathcal{V}^s_{k,d,g}$	The set of GPUs in group g and dimension d which act as sources to send chunks at stage k ($\mathcal{V}^s_{k,d,g} \subseteq \mathcal{V}_{d,g}$)
$\mathcal{V}^r_{k,d,g}$	The set of GPUs in group g and dimension d which act as destinations to receive chunks at stage k ($\mathcal{V}^r_{k,d,g} \subseteq \mathcal{V}_{d,g}$)
$\mathcal{R}_{k,d,g}$	The communication sub-demand in group g in dimension d ($\mathcal{R}_{k,d,g} = \mathcal{V}^s_{k,d,g} \to \mathcal{V}^r_{k,d,g}$)

Table 3: Sketch denotations.

Stage k=0	Stage k=1
$\mathcal{R}_{0,0,0} = \{0\} \to \{1,2,3\}$, $\mathcal{R}_{0,1,0} = \{0\} \to \{4,8,12\}$	$\mathcal{R}_{1,0,1} = \{4\} \to \{5,6,7\}$, $\mathcal{R}_{1,0,2} = \{8\} \to \{9,10,11\}$, $\mathcal{R}_{1,0,3} = \{12\} \to \{13,14,15\}$

Table 4: Sketch variables corresponding to Figure 5.

construct a complete demand, *i.e.,* $\cup_{d,g}\mathcal{V}^s_{0,d,g} \bigcup \cup_{k,d,g}\mathcal{V}^d_{k,d,g} = \mathcal{V}$. Table 4 shows the variables for sketch ①.

A sketch can be represented as a graph structure, with each node representing a GPU. For example, sketch ① in Figure 8 is the graph representation of sketch ① in Figure 5. An edge from a source GPU $v_1 \in \mathcal{V}^s_{k,d,g}$ to a corresponding destination GPU $v_2 \in \mathcal{V}^r_{k,d,g}$ indicates that v_2 requires chunks from v_1. If both are destinations in the same sub-demand (*i.e.,* $v_1, v_2 \in \mathcal{V}^r_{k,d,g}$), they are connected by a dashed line to indicate potential communication. In Figure 8, GPUs 1, 2, and 3 require chunks from GPU 0 (solid line) and may receive chunks from each other (dashed line). The solid and dashed edges together represent potential communication in each group, with edge weights indicating link performance.

Enumeration-based sketch searching. To explore potential sketches, we systematically enumerate possible communication sub-demands at each stage. SyCCL introduces two variables: $\mathcal{D}_k$, representing the dimensions involved in communication at stage k, and $\mathcal{G}_{d,k}$ which specifies the groups that participate in communication in dimension $k \in \mathcal{D}_k$. The search process follows three steps for each stage: (1) Enumerate possible dimensions $\mathcal{D}_k \subseteq \mathcal{D}$ for communication, (2) For each $\mathcal{D}_k$, enumerate groups $\mathcal{G}_{d,k} \subseteq \mathcal{G}_d$ in each dimension $d \in \mathcal{D}_k$, and (3) For each $\mathcal{G}_{d,k}$, enumerate source GPUs $\mathcal{V}^s_{k,d,g}$ and their corresponding destinations $\mathcal{V}^r_{k,d,g}$ for each group $g \in \mathcal{G}_{d,k}$. A GPU can only act as a source if it has received chunks, and may act as a destination only once.

Pruning based on symmetry insights. Numerous possible combinations of sub-demands correspond to various ways of partitioning the original communication demand. However, the insight from §3.1 indicates that isomorphic groups yield consistent sub-schedules, allowing us to develop pruning strategies to eliminate sub-optimal and redundant sketches.

Pruning #1: Removing redundant sketches through isomorphism detection. Many sketches are isomorphic due to topology symmetry. For example, sketches ② and ③ in Figure 8 can be transformed into one another by swapping GPUs 9-11 with GPUs 13-15, while sketch ① is not isomorphic to them. Isomorphic sketches result in schedules with identical performance, leading to redundant solving efforts. SyCCL filters out isomorphic sketches to accelerate the solving process, defining two sketches as isomorphic if a one-to-one mapping exists between their GPU sets: $H : \mathcal{V} \to \mathcal{V}$.

Pruning #2: Enforcing consistency across groups and stages. Based on the insight from §3.1, isomorphic groups present consistent sub-schedules. While the solver will eventually determine

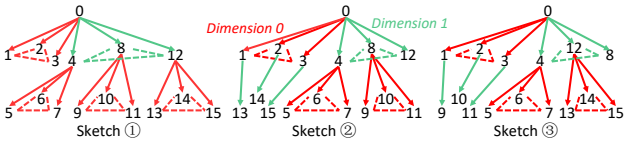


Figure 8: Example of pruning #1. Sketches ② and ③ are isomorphic to each other, while sketch ① is not.

specific sub-schedules, we can proactively identify sub-demands likely to violate this principle. SyCCL utilizes the ratio of destination GPUs to source GPUs within each group (*i.e.,* $|\mathcal{V}^r_{d,k,g}|/|\mathcal{V}^s_{d,k,g}|$) to reflect the sub-demand per group. A sketch is deemed consistent if this ratio remains uniform across isomorphic groups, while groups not communicating or in the final are excluded. Figure 9 shows three examples of a seven-server GPU cluster. Due to space limitations, the topology is shown in Figure 19 in Appendix B. Sketches ④ and ⑤ display consistent sub-demands across groups in both dimensions 0 and 1. In contrast, sketch ③ shows varying ratios in dimension 1 (*i.e.,* GPUs 1, 6, and 16 correspond to 1, 3, and 2 destination GPUs, respectively).

Pruning #3: Limiting the number of relays for `Scatter`. In a `Scatter` tree, a GPU with n descendants receives n redundant chunks from its predecessor , adding to communication overload. To reduce this issue, SyCCL restricts the maximum number of transfer hops from the root GPU to any other GPU to a parameter X. In practice, X is set to $|\mathcal{D}| - 1$, ensuring that each dimension is passed along at most once.

4.2 Generating Sketch Combinations

Each sketch generated in §4.1 represents the transmission of a single chunk and utilizes only part of the available bandwidth. In practice, chunks can be split and sent through different paths. This section introduces how to generate sketch combinations to optimize bandwidth usage.

A sketch combination $\mathcal{M}$ consists of pairs of sketches and their associated chunk size ratios. Each pair $\langle S_i, t_i \rangle$ indicates that sketch S_i transmits a fraction t_i of a chunk, ensuring that the total transmitted fraction equals one (*i.e.,* $\sum t_i = 1$). The performance of each combination is evaluated only after determining the schedule (§5), prompting SyCCL to generate all possible sketch combinations.

For small chunk sizes, a single sketch suffices, so SyCCL outputs each sketch individually as a combination. For larger sizes, SyCCL creates combinations to maximize bandwidth usage. In Figure 5 example, each sketch from ① to ③ can create its own combination, while together they form other combinations. Since it is difficult to classify chunk sizes as small or large, SyCCL generates both types of combinations for all chunk sizes. We first discuss the issue of bandwidth under-utilization in sketches and then introduce how SyCCL produces combinations for large chunk sizes.

Bandwidth under-utilization in a single sketch. For a sketch, SyCCL calculates its workload $w_{d,g}$ for group g in dimension d (denoted as Gg.Dd) as follows. For `Broadcast`, it is the total number of destination GPUs across all relevant sub-demands $\sum_{k\in\mathcal{K}} |\mathcal{V}^r_{k,d,g}|$. For `Scatter`, it is $\sum_{k\in\mathcal{K}, v\in\mathcal{V}^r_{k,d,g}} (f(v)+1)$, where $f(v)$ represents the number of descendants for GPU v. The total workload for dimension d is $w_d = \sum_{g\in\mathcal{G}_d} w_{d,g}$.

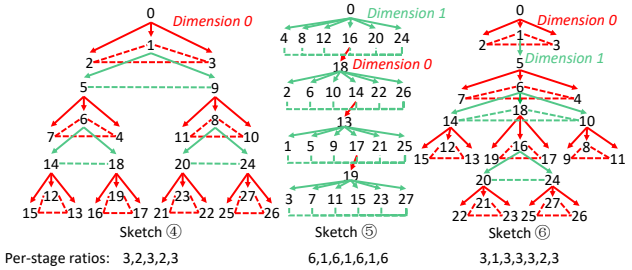


Figure 9: Example of pruning #2. Sketches ④ and ⑤ present consistent communication across isomorphic groups, while sketch ⑥ does not and should be filtered.

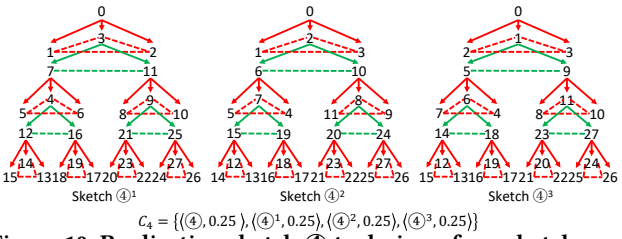


Figure 10: Replicating sketch ④ to derive a four-sketch combination to balance workload in dimension 1.

The bandwidth under-utilization is essentially caused by two types of imbalanced workload. First, when two groups in the same dimension have different workloads, the group with smaller workload experiences bandwidth under-utilization. For example, in sketch ④ in Figure 9, the workload in groups 0, 1, and 2 in dimension 1 is identical, while group 3 remains idle and is underutilized. Second, when the workloads of two dimensions do not align with their actual bandwidths, bandwidth under-utilization also happens. For example, sketch ① has a workload ratio of 12:3 across dimensions 0 and 1. If the actual ratio differs, it can lead to one dimension becoming a bottleneck while the other experiences bandwidth wastage.

To address the bandwidth under-utilization issue, SyCCL creates sketch combinations to balance workload across groups and dimensions in the following two steps.

Step 1: replicating sketches to balance workload across groups. For each sketch generated in §4, SyCCL replicates it to form a combination that balances the workload across groups in each dimension. This process utilizes a mapping strategy that allocates sub-demands to underutilized groups.

Mapping-based replication mechanism. SyCCL treats the replication as a mapping of GPUs and groups. Each group and GPU in the original sketch is mapped to its counterpart in the replicated sketch, thereby forming two one-to-one mappings (*i.e.,* $F : \mathcal{V} \to \mathcal{V}$ for GPUs and $H_d : \mathcal{G}_d \to \mathcal{G}_d$ for groups). The objective is to utilize groups with lower workload in the original sketch to balance workload.

Specifically, Starting from stage 0, at each stage k, SyCCL decides the mappings for source GPUs, groups, and destination GPUs for each sub-demand $\mathcal{R}_{k,d,g}$. At stage 0, each source GPU maps to itself, as it is still a source in the replica. In later stages, each source GPU is a destination at previous stages, therefore its mapping should be already established. From the existing mapping, the mapping for

group g containing a source GPU can be determined, *i.e.,* g is mapped to the group that contains the mapped source GPUs in dimension d. Next, SyCCL decides the mappings of destination GPUs. There are various mapping options. SyCCL prioritizes destinations that will become sources in the next stage, as their mappings impact the group mapping and overall workload. For each GPU that will be a source in the next stage, SyCCL maps it to a GPU from the group with the least workload in the same dimension. If a GPU is not a source, it can be mapped to any GPU in the mapped group. This process continues in subsequent stages, ultimately resulting in a replicated sketch.

For each sketch generated in §4.1 (denoted as S^0), SyCCL replicates it to produce sketches S^1, S^2, ... These sketches share the same fraction t. SyCCL repeats the replication process until the overall workload is balanced and creates a combination $C = \{\langle S^j, t\rangle\}$, where $0 \le j < |C|$ and $t = \frac{1}{|C|}$.

Example. Figure 10 illustrates the replication of sketch ④, whose workload is imbalanced in both dimensions 0 and 1. SyCCL replicates this sketch three times, yielding sketches ④^1, ④^2,and ④^3, which together achieve workload balance. Take the replication for sketch ④^1 as an example. The process begins with GPU 0, which acts as a source in D0.G0, SyCCL first maps GPU 0 and group D0.G0 to themselves. Next, when deciding the mappings for GPUs 1, 2, and 3, GPU 1 is prioritized since it will be a source in D1.G1 at stage 1. Since group 3 has the lightest workload in dimension 1, SyCCL maps D1.G1 to D1.G3, resulting in GPU 1 being mapped to GPU 3 in sketch ④^1. For GPUs 2 and 3, they are then mapped to the remaining GPUs in D0.G0, *i.e.,* GPUs 1 and 2. For GPUs 5 and 9, which act as sources in D0.G1 and D0.G2, respectively, SyCCL evaluates the current workload with the replicated sketch into consideration, and finds that both groups have the lightest workload, so both are mapped to themselves. The process continues in a similar manner for later mappings, with additional details omitted for brevity.

Step 2: integrating sketch combinations to balance workload across dimensions. In this step, SyCCL takes the sketch combinations obtained in step 1 as input sketches, and integrates them to produce new sketch combinations to achieve balanced workload across dimensions. With $|\mathcal{D}|$ dimensions, this process involves solving a linear problem with $|\mathcal{D}|$ variables, requiring $|\mathcal{D}|$ sketches as input.

Chunk allocation mechanism. For each $|\mathcal{D}|$-sketch candidate, SyCCL calculates the ratio of the chunk each sketch transmits, denoted as t_i for the i^{th} sketch. This allocation aims to achieve balanced workload across dimensions. If no suitable allocation is found, the candidate is deemed invalid. Ultimately, SyCCL outputs all valid sketch combinations.

Specifically, given a $|\mathcal{D}|$-sketch candidate, we calculate the workload for dimension d for each sketch, denoted as $w_{i,d}$ for the i^{th} sketch. Then the total workload proportion for dimension d is given by $w_d = \sum_i (t_i * w_{i,d}) / \sum_{i,d} (t_i * w_{i,d})$. Assuming the bandwidth proportion for dimension d relative to the topology capacity is u_d, the t_i values should satisfy that $w_d = u_d$, *i.e.,* bandwidth across all dimensions is fully utilized.

Example. Consider two dimensions (0 and 1) and two sketches (④ and ⑤). In step 1 we replicate sketch ④ and derive a sketch combination C_4 consisting of four sketches. Each sketch in C_4 transmits 1/4 of the chunk. Similarly, we replicate sketch ⑤ to form combination C_5 consisting of seven sketches, with each transmitting 1/7 of the chunk. The bandwidth ratios for dimensions 0 and 1 used by C_4 and C_5 are 21:6 and 3:24, respectively. Assume that the actual link capacity for both dimensions is 4:5. In this case, both C_4 and C_5 transmits half of the chunk. Then we combine all 11 sketches from both C_4 and C_5. In this final combination, each sketch in C_4 transmits 1/8 of the chunk, while each sketch in C_5 transmits 1/14 of the chunk.

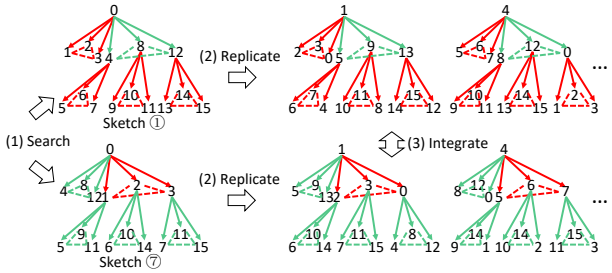


Figure 11: Example of generating `AllGather` sketches. SyCCL first generates two distinct sketches (① and ⑦), with preferential link occupation to Dimension 0 (red line) and Dimension 1 (green line), respectively. SyCCL then replicates both sketches to achieve comprehensive coverage across all groups within each dimension. Third, SyCCL decides data volumes transmitted through each sketch according to cross-dimensional bandwidth ratios.

4.3 Extending to All-To-All Collectives

An N-GPU all-to-all collective consists of N isomorphic one-to-all or all-to-one collectives. Specifically, `AllGather`, `Alltoall`, and `ReduceScatter` are broken down into `Broadcasts`, `Scatters`, and `Reduces`, respectively.

Figure 11 shows how to generate sketch combinations for these collectives. SyCCL first searches for sketches for a single decomposed collective, as described in §4.1. For each sketch searched (*i.e.,* sketches ① and ⑦ in the figure), SyCCL replicates it to create corresponding sketches for the other $N-1$ decomposed collectives, resulting in an N-sketch combination. This replication process is similar to step 1 in §4.2, with the key distinction that the source GPU is mapped to different GPUs. The N-sketch combination ensures an even workload distribution in each dimension. Finally, SyCCL integrates the two N-sketch combinations like step 2 in §4.2 to generate the final sketch combinations.

While `AllReduce` can be split into `Reduce` collectives, these collectives share data chunks and are not independent. Instead, SyCCL treats `AllReduce` as `ReduceScatter` followed by `AllGather`. As a result, synthesizing an `AllReduce` schedule is equivalent to synthesizing separate `ReduceScatter` and `AllGather` schedules and concatenating them.

5 Schedule Synthesis

This section introduces how SyCCL synthesizes the complete schedule based on the sketch combinations generated in §4. First, SyCCL synthesizes optimal sub-schedules inside each sketch combination

through MILP modeling (§5.1). SyCCL then integrates these sub-schedules into complete schedules. Since each sketch combination leads to an individual optimal schedule, SyCCL selects the best one among them (§5.2). In §5.3, SyCCL introduces two optimizations to accelerate the synthesis process while maintaining accuracy.

5.1 Synthesizing Sub-Schedules

This section introduces how SyCCL synthesizes optimal sub-schedules inside each sketch combination in §4.2.

Each sketch combination consists of multiple sub-demands across different time stages and GPU groups. Each sub-demand can be satisfied by different sub-schedules with various performance. For example, in Figure 5, each sub-demand has two potential sub-schedules (*i.e.,* ① and ②). SyCCL merges the sub-demands from the same GPU group at the same stage, as they are expected to occur simultaneously and may compete for bandwidth. Then SyCCL synthesizes an optimal sub-schedule (marked with a check in Figure 5) for each merged sub-demand through MILP modeling as follows.

First, SyCCL models each merged sub-demand as a MILP problem, following the established modeling technique from TECCL [28]. This method divides time into equal intervals, *i.e.,* **epochs**. Each transmission must occupy an integer number of epochs, allowing us to use integer variables to depict the communication process. SyCCL adopts the α-β transmission model [24]. Specifically, transmitting a chunk c with size s over a link l requires a total time of $\alpha + \beta * s$ for the chunk to reach its destination, and the link requires a time of $\beta * s$ before it can handle the next chunk. Then, SyCCL utilizes a MILP solver to calculate the minimum number of epochs required to satisfy the sub-demand and generate the optimal sub-schedule.

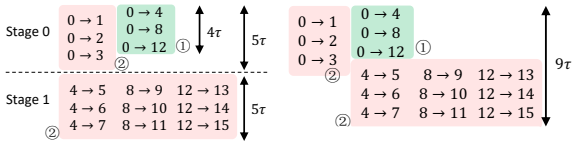
Since TECCL has provided a detailed description of the modeling and solving, we only summarize the key components and put the details in Appendix A.1. Compared with TECCL, which addresses the complete collective demand within a complete topology, SyCCL improves scalability by focusing on smaller sub-demands within smaller topology subsets, thereby minimizing the problem size.

5.2 Synthesizing the Optimal Schedule

Once optimal sub-schedules are synthesized, SyCCL merges them into a complete schedule. For example, the sub-schedules ② and ① at stage 0, and ② at stage 1 in Figure 5 are merged into a complete schedule (*i.e.,* schedule ② in Figure 4). Each sketch combination corresponds to a unique output schedule. SyCCL then evaluates the performance of all generated schedules and selects the best one.

A straightforward approach to evaluate the schedule performance is simply adding up the duration of each stage. This can lead to inaccuracies, as the communication across stages are not independent. For example, as shown in Figure 12(a), the three sub-schedules take 5τ, 4τ, 5τ, respectively. Directly adding the time required in both stages results in a total time of 10τ. However, the communication at stage 1 does not need to wait for stage 0 to finish. Specifically, GPUs 4, 8, and 12 can begin communication immediately upon receiving a chunk from stage 0, leading to a smaller completion time (9τ), as shown in Figure 12(b).

To accurately assess schedule performance, SyCCL implements a fine-grained schedule simulator based on ASTRA-sim [3]. The



(a) A naive evaluation method (b) SyCCL

Figure 12: SyCCL predicts schedule performance by simulating transmission events.

simulator adopts the same transmission model (*i.e.,* α-β model) as the solver. The simulator scans the communication events in chronological order and starts from events from the source GPUs. A transmission event can begin when the source GPU has obtained the chunk and the previous events on the link have been completed. For each event, the simulator directly computes its finish time with the α-β model. Once all events are finished, the simulator outputs the completion time as the performance of the schedule. Because every event is processed exactly once, the simulator's time-complexity is O(E), where E is the number of communication events.

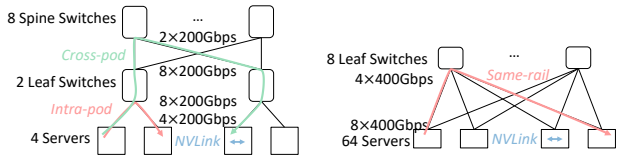
In the CCL transport implementation, a chunk is usually divided into smaller blocks during transmission in order to pipeline across multiple hops. The number of communication events is equal to the number of transmissions in the schedule times the number of blocks for the chunk size. Under such scenario, our simulator based on analytical modeling has negligible time usage compared to schedule synthesis.

5.3 Accelerating Synthesis

SyCCL faces two scalability challenges. First, MILP solvers must balance accuracy (*i.e.,* the performance of the synthesized schedule) and efficiency (*i.e.,* the speed of synthesis). For example, the TECCL solver employs epoch duration τ for modeling. A larger τ leads to faster but less accurate schedule synthesis (see Appendix A.2 for more details). While SyCCL greatly reduces the problem size by focusing on individual GPU groups, we still face the scalability challenge. For example, [30] introduces a GPU cluster with a single switch connecting 128 GPUs. In such cases, SyCCL still faces the accuracy-efficiency trade-off. Second, each sketch combination contains numerous sub-demands. Solving all of them one by one is time-consuming.

Two-step synthesis to maintain accuracy. To address the first challenge, SyCCL designs a two-step synthesis method, accelerating synthesis while maintaining accuracy. First, SyCCL employs coarse-grained solving (*e.g.,* using larger τ) to quickly generate schedules for all sketch combinations. SyCCL then filters out those with worse performance (*i.e.,* perform worse than the best by more than R_1) and retain no more than R_2 top candidates. Second, SyCCL employs fine-grained solving (*e.g.,* using smaller τ) to the selected candidates to achieve accurate synthesis.

Additionally, we found that the value of τ needs careful selection (*e.g.,* it should be a multiplier of the product of link bandwidth parameter β and the chunk size s) to satisfy link performance constraints. We introduce an auxiliary variable E, which automatically determines the appropriate τ value, with larger E leading to larger



(a) A100, Clos (b) H800, Rail

Figure 13: GPU cluster topologies used in evaluation.

τ. Due to space limitations, further details are provided in Appendix §A.3.

This two-step synthesis approach does not rely on specific modeling and solving methods. Even with more advanced methods, a trade-off between accuracy and efficiency persists, with parameters like τ and E to adjust this balance.

Utilizing isomorphism and parallelism to accelerate synthesis. To address the second challenge, SyCCL leverages isomorphism and parallelism to accelerate synthesis. Many sub-demands exhibit isomorphic properties. For example, a one-to-all sketch may comprise multiple groups with isomorphic sub-demands, while all-to-all sketches are generated by replicating a one-to-all sketch, leading to a massive number of isomorphic sub-demands. Consequently, we can generate schedules for isomorphic sub-demands by solving only one of them and mapping the solution to others. Before employing the solver for each sub-demand, we collect all sub-demands, analyze isomorphism between sub-demands in the same dimension, and partition them into isomorphic classes. Demands in the same class can be mapped to each other via appropriate GPU mappings. This way, schedules generated for these demands can be derived from each other based on the same GPU mapping. Furthermore, since the numerous demands are independent. SyCCL solves them in parallel to accelerate synthesis.

6 Implementation

The implementation of SyCCL consists of three components. First, the network profiler measures the link parameters α and β by testing various chunk sizes for links in each dimension, like TECCL [28] and TACCL [33]. Second, the schedule synthesizer, which consists of 7K lines of C++ code, implement the core logic of SyCCL containing sketch exploration, MILP based schedule synthesis and simulation. Third, the schedule executor converts synthesized schedules into XML files with varying runtime parameters like the CCL transport and the number of communication channels used. The XML is injected into MSCCL-executor [10] via a lightweight XML parser. MSCCL-executor directly executes the XML without modifying any CUDA kernel or memory management path. We open-sourced the code of SyCCL at https://github.com/aliyun/symccl.

7 Evaluation

We evaluated the accuracy (*i.e.,* performance of synthesized schedules) and efficiency (*i.e.,* synthesis time) of SyCCL under various real topologies, collective types, and data sizes. We compared the results with the state-of-the-art synthesizer TECCL [28] and the open-source NCCL [11]. To validate the effectiveness of SyCCL's design, we evaluated how different pruning and optimization strategies impact both accuracy and efficiency. Additionally, we evaluate the end-to-end performance when training different models (*e.g.,* GPT-6.7B [16] and Llama3-8B [35]) with different parallelism and overlapping mechanisms.

7.1 Setup

Topology. We utilized two topologies from our production clusters. Figure 13(a) is an A100 cluster consisting of 4 servers. Each server is equipped with 8 Nvidia A800 GPUs and 4x200Gbps RDMA NICs. The servers connect through a two-layer Clos network, where every two servers are connected to one ToR switch. Figure 13(b) is an H800 cluster made up of 64 servers. Each server is equipped with 8 Nvidia H800 GPUs and 8x400Gbps RDMA NICs. The servers are interconnected via a multi-rail network (*i.e.,* GPUs with the same ID are connected to the same switch). In both clusters, GPUs within the same server are also connected to NVSwitches via NVLinks. In the following experiments, the A100-related tests were conducted on a real testbed, while the H800-related tests utilized parameters measured from a real-world cluster and were executed through simulation.

Synthesis. We run the schedule synthesis process for both SyCCL and TECCL on a server with 192-core Intel Xeon Platinum 8469C CPU and 1TB memory.

For TECCL, we initially use the parameter settings provided in the examples of their open-source codebase, selecting the best output among different parameters for each case. We set a timeout of 10 hours total for solving. For larger cases where these parameters leads to timeout, we manually tune the epoch duration τ to ensure a solution can be found. We start with a large epoch duration τ which ensures that a feasible solution can be found within the time limit. Then we gradually decrease τ to improve synthesis precision until either (1) no feasible solution can be found or (2) τ is smaller than a threshold τ_{min} specified by TECCL, *i.e.,* $\beta * s$ of the fastest link, and output the result.

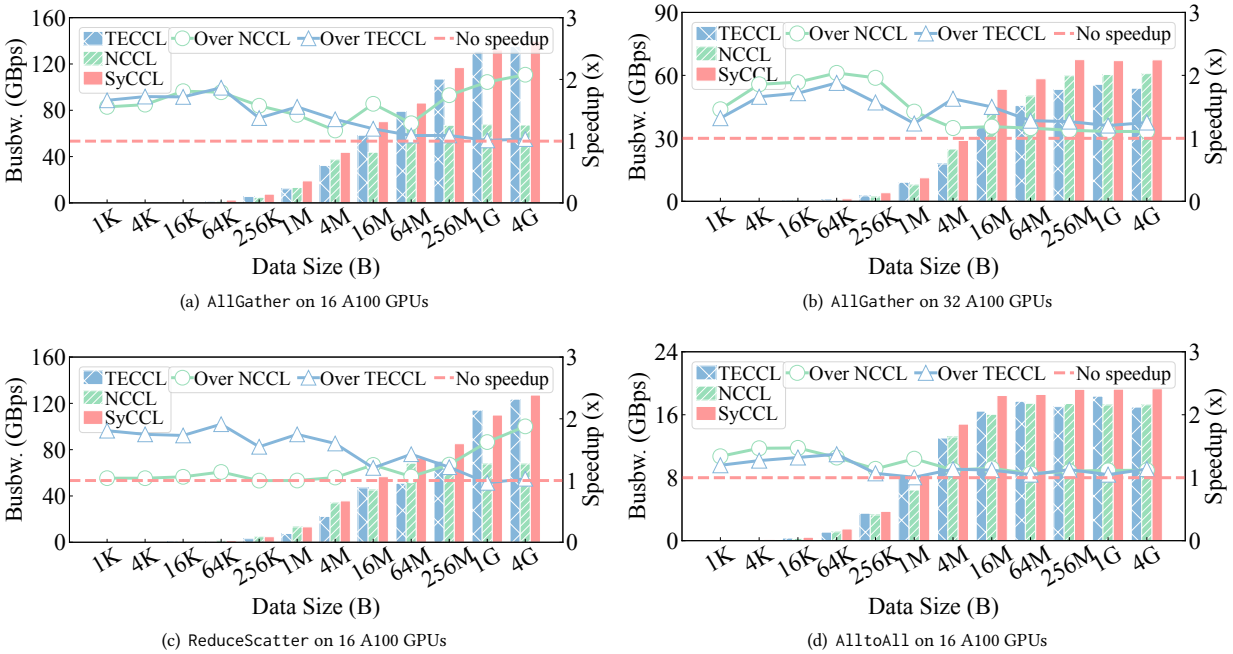
For SyCCL, we employ a two-step deterministic synthesis process as described in §5.3. We set $E_1 = 3.0$ and $E_2 = 0.5$ to automatically derive the epoch duration τ in both steps and $R_1 = 20\%$, $R_2 = 8$. For the 512 H800 case, we set $E_2 = 1.0$. This configuration successfully solved all cases within 4 hours.

Metric. We take Bus Bandwidth (busbw. [4]) as the metric to measure the performance of collective schedules. Busbw reflects the extent to which the overall bandwidth is utilized.

7.2 Schedule Performance

We compared the performance of schedules synthesized by SyCCL, TECCL, and NCCL for different collective types (`AllGather`, `Alltoall`, and `ReduceScatter`) over a range of data sizes (1KB to 4GB) and different topology scales (from a single server with 8 GPUs to 64 servers with 512 GPUs).

`AllGather` and `ReduceScatter` on A100 topology. We first evaluated `AllGather` and `ReduceScatter` performance of SyCCL, TECCL, and NCCL on the A100 testbed. As shown in Figures 14(a) and 14(b), SyCCL outperforms TECCL and NCCL for `AllGather` under both 16-GPU and 32-GPU topologies. Under the 16-GPU topology, for data sizes ≤1MB, SyCCL improves busbw by 0.43×-0.81× compared to NCCL, and 0.37×-0.86× compared to TECCL. Both SyCCL and TECCL demonstrate much better performance



(a) `AllGather` on 16 A100 GPUs

(b) `AllGather` on 32 A100 GPUs

(c) `ReduceScatter` on 16 A100 GPUs

(d) `AlltoAll` on 16 A100 GPUs

Figure 14: Schedule performance on A100 topology.

for smaller data sizes compared to NCCL. This is because that NCCL's fixed ring or tree schedules contain excessive hops, which is inefficient for smaller sizes, where link latency (α) dominates.

For data sizes >1MB, SyCCL achieves 0.17×-1.08× improvement over NCCL and up to 0.36× improvement over TECCL. We observe that the 16-GPU ring schedules in NCCL utilize a fixed ratio of 7:1 for NVLink and network bandwidths, resulting in a bottleneck for network bandwidth and redundancy in nvlink bandwidth. In contrast, SyCCL alleviates the bottleneck by minimizing the network bandwidth used, utilizing a bandwidth ratio of 14:1.

As the number of GPUs increases from 16 to 32, TECCL struggles to generate a competitive schedule. Due to the increase of MILP model size, TECCL sacrifices accuracy in order to meet the time limit. In contrast, SyCCL achieves up to 1.04× improvement over NCCL and 0.20×-0.88× over TECCL.

For `ReduceScatter`, the gain of SyCCL compared to NCCL is slightly smaller than that observed in `AllGather`. We believe this is due to NCCL's more efficient coordination between communication and reduction, which is tailored to its static schedules. In contrast, MSCCL's runtime, designed to support flexible schedules, does not incorporate such tailored optimizations for a given schedule. Therefore, although SyCCL's schedules benefit from higher bandwidth utilization and lower latency, these advantages can be diminished by a sub-optimal runtime implementation, especially for smaller sizes. This showcases the importance of deeper runtime optimization to complement schedule optimization [40] and fully realize performance gains.

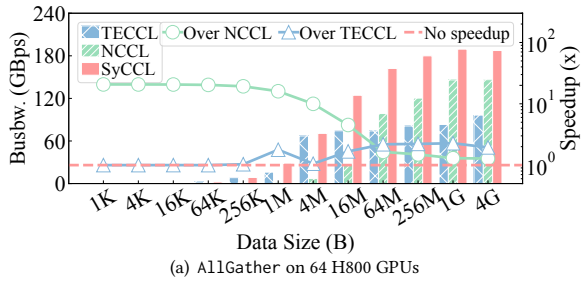
`Alltoall` on A100 topology. Figure 14(d) shows 16-GPU `Alltoall` performance. Compared to NCCL, SyCCL increases busbw by up to 0.71×.

The schedule performance of SyCCL is similar to that of TECCL for `Alltoall` because this collective decomposes into many independent point-to-point transfers that offer limited room for schedule reordering. On the relatively small, non-oversubscribed topologies in our testbed, NCCL's built-in schedule already utilizes all links efficiently, and TECCL's LP formulation [28] finds a near-optimal solution.

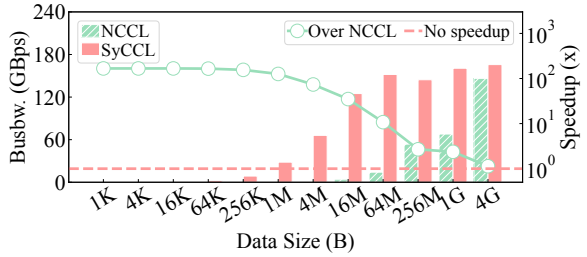
`AllGather` on H800 topology. Figure 15(a) and Figure 15(b) show the `AllGather` performance under larger topologies (64 GPUs and 512 GPUs) through simulation. TECCL timed out for 512 GPUs and produced no valid output. As the number of GPUs increases, the data in `AllGather` becomes more fragmented and the link latency dominates. For example, with a total size of 1GB distributed across 512 GPUs, each GPU only handles 2MB. Thus the 511 hops in NCCL's Ring schedule severely limit the performance. Instead, SyCCL synthesized a two-dimension schedule, by allowing each source GPU to first broadcast along one dimension (*e.g.,* NVLink) then letting each GPU broadcast all their data received from the first dimension to other dimensions (*e.g.,* rail between servers). This strategy results in significantly better performance for smaller sizes. At larger sizes, SyCCL's sketches utilize bandwidth more efficiently than NCCL's fixed schedules.

As a result, SyCCL achieves up to 1.27× improvement compared to TECCL for 64 GPUs, and has significant improvement over NCCL for 512 GPUs.

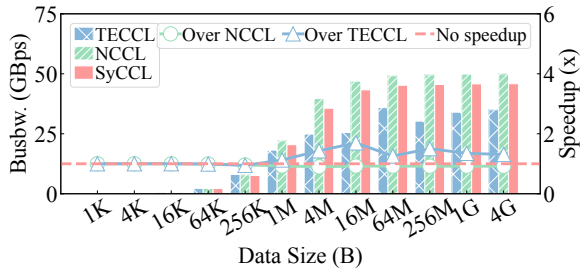
`Alltoall` on simulated H800 topology. Figure 15(c) shows the `Alltoall` performance under 64 H800 GPUs. The NCCL PXN `Alltoall` schedule is near-optimal in H800 topologies. Because of uncertainties of the solver and precision issues in the LP model, SyCCL perform slightly worse than NCCL. In contrast, TECCL models the whole collective communication and faces more severe



(a) `AllGather` on 64 H800 GPUs



(b) `AllGather` on 512 H800 GPUs (TECCL timed out with no solution output)



(c) `AlltoAll` on 64 H800 GPUs

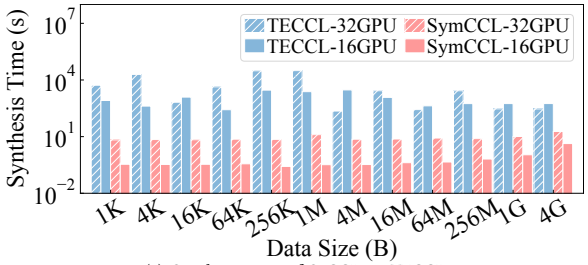
Figure 15: Schedule performance on H800 topology.

precision issues. As a result, SyCCL improves performance by up to 0.69× compared to TECCL.

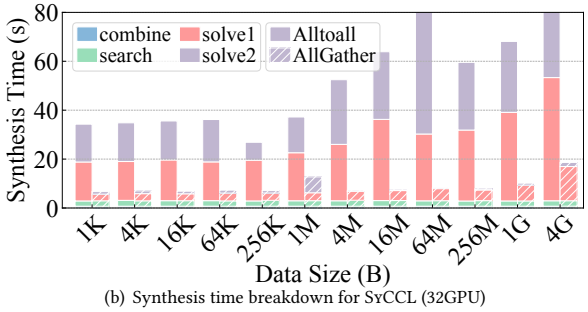
7.3 Synthesis Time

SyCCL vs. TECCL. Figure 16(a) shows the synthesis time for SyCCL and TECCL to synthesize an `AllGather` schedule on 16 and 32 A100 GPUs. For various data sizes, SyCCL requires only 0.3s-2.6s for 16 GPUs and 6.2s-19.1s for 32 GPUs, while TECCL requires 4.4min-49.5min and 3.8min-8.7h, respectively. SyCCL accelerates synthesis by 2 to 3 magnitudes compared to TECCL. We summarize the synthesis time in various scenarios in Table 5. For configuration with up to 64 GPUs, SyCCL requires at most 29.8s, while TECCL can take up to 18h. In various cases, SyCCL reduces synthesis time by 915× to 17286× compared to TECCL.

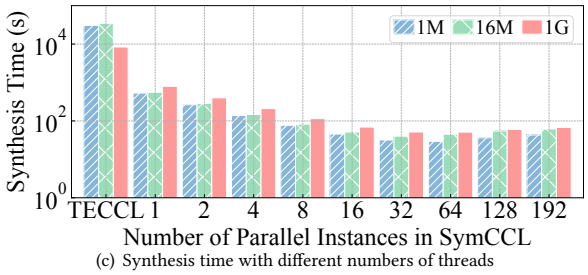
While SyCCL significantly reduces schedule generation time by leveraging symmetry, extremely large-scale scenarios (e.g., 512 H800 GPUs) can still incur notable synthesis time, *i.e.,* 37min. Profiling shows that this residual cost stems almost entirely from invoking TECCL to solve schedules within each symmetric GPU group; the sketch-exploration phase itself scales sub-second. Hence the bottleneck lies in the TECCL solver, not in SyCCL 's core algorithm, and future work on faster intra-group solvers would improve both systems orthogonally.



(a) Synthesis time of SyCCL vs. TECCL



(b) Synthesis time breakdown for SyCCL (32GPU)



(c) Synthesis time with different numbers of threads

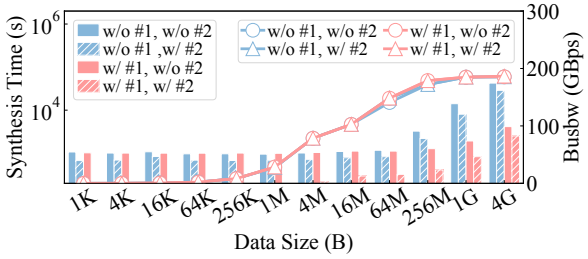
Figure 16: Synthesis time of SyCCL and TECCL on A100 topology.

Scenario	Min/Max/Mean Time (s)		Speedup
	TECCL	SyCCL	
16 A100, AG	266/2972/1193	0.3/4.3/0.8	1554×
16 A100, A2A	1042/26206/15759	1.4/8.3/3.6	4321×
32 A100, AG	226/31293/8200.2	6.7/18.6/9.0	915×
64 H800, AG	1225/67615/28200	0.4/10.8/1.6	17286×
64 H800, A2A	3698/59044/29371	1.3/29.8/5.7	5135×
512 H800, AG	Time Out	85.5/14146/2246	N/A

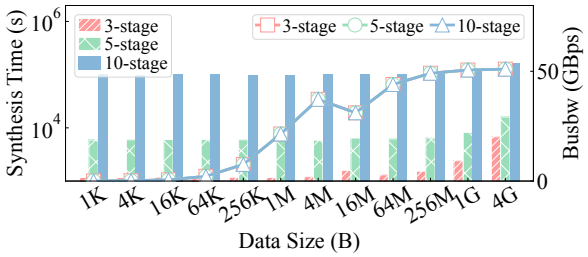
Table 5: Synthesis time (s) of SyCCL and TECCL. `AG/A2A` stands for `AllGather/Alltoall`.

Synthesis time breakdown of SyCCL. We profile the time spent on each step of the SyCCL synthesis process, including sketch search, sketch combining, and schedule synthesis in two steps. As shown in Figure 16(b), the majority of the synthesis time is attributed to schedule synthesis. The time needed for sketch search and combination generation remains steady regardless of the data size, as these steps are not affected by it. However, schedule synthesis time increases with larger data sizes due to the need for more complex schedules to effectively utilize bandwidth.

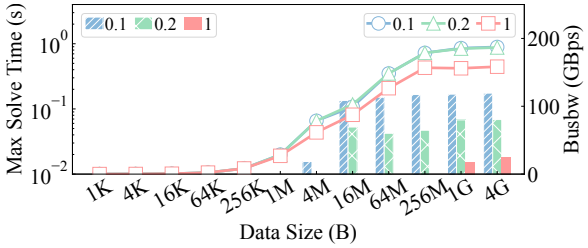
Effect of parallelism in SyCCL. SyCCL employs multiple parallel instances to solve sub-demands and run schedule simulations simultaneously. In contrast, TECCL relies on a single instance to address a large-scale problem, resulting in limited scalability. For



(a) With and without Pruning #1 and #2 in §4.1



(b) With and without Pruning #3 in §4.1



(c) Various E_2 value in §5.1

Figure 17: Impact of synthesizing policy on synthesis time and schedule performance.

both SyCCL and TECCL, we enable the built-in multi-thread optimization of MILP solver, which allows multiple threads to run within a single instance to accelerate solving. We compare the time it takes by SyCCL with different numbers of instances against the time taken by TECCL. As shown in Figure 16(c), with more parallel instances, SyCCL significantly reduces synthesis time.

7.4 Impact of Varying Synthesizing Policy

To evaluate the effectiveness of each part of SyCCL's design, we conducted a microbenchmark to compare the results of enabling and disabling each part of design. We scaled down the H800 server from 8 GPUs to 4 GPUs and utilized 6 servers, while maintaining the same topology. This allowed us to evaluate SyCCL in a smaller yet complex setup.

Effect of pruning strategies in sketch exploration. To measure the impact of pruning strategies in §4.1, we compare the synthesis time and performance of synthesized schedules with and without the application of these pruning strategies. Figure 17(a) illustrates that the redundancy remover and consistency enforcement strategies can significantly reduce the synthesis time by 20.8% to 48.1%, with minimal impact on the performance of the synthesized schedules. Figure 17(b) shows that setting `Alltoall` stage to ≤ 3 in the current topology does not affect the final schedule performance. Additionally, this adjustment can significantly reduce synthesis time, *e.g.,* saving 95% to 97% compared to allowing 10 stages.

Effect of epoch duration τ setting in schedule synthesis. Figure 17(c) shows that how changing E_2 (*i.e.,* E value during step-2 synthesis) affects synthesis time and schedule performance. We employ the maximum solve time across demands, instead of the total synthesis time, to demonstrate epoch duration's impact on the solver while solving a single demand. We can see that $E_2 = 0.2$ is an appropriate value. Using a larger E_2 (*e.g.,* 1) leads to a decrease in schedule performance, while using a smaller E_2 (*e.g.,* 0.1) will result in longer solving times without any performance improvement.

Model	NCCL	TECCL	SyCCL	vs NCCL	vs TECCL
GPT3-6.7B, DP16	1307.0	1312.0	1246.9	4.6%	5.0%
GPT3-6.7B, TP16	428.3	422.4	393.1	8.2%	7.0%
GPT3-6.7B, TP32	530.4	613.9	470.8	11.2%	23.3%
Llama3-8B, TP16	808.3	806.8	780.9	3.4%	3.2%

Table 6: Iteration time (ms) and speedup of model training using NCCL, TECCL and SyCCL.

7.5 End-to-end Performance

We evaluate SyCCL on distributed training of two models, *i.e.,* GPT-6.7B [16] and Llama3-8B [35] on A100 GPUs. We measure the training iteration time across different parallelism configurations (*i.e.,* data parallelism (DP) with distributed optimizer enabled, tensor parallelism (TP), and with computation-communication overlap.) As shown in Table 6, SyCCL improves performance by up to 11.2% compared to NCCL and up to 23.3% compared to TECCL under various configurations.

8 Discussions and limitations

Optimality of synthesized schedules. Although the introduction of sketch significantly enhances the speed and accuracy of SyCCL compared to previous efforts, SyCCL does not ensure that the synthesized schedules are optimal. This limitation arises because SyCCL employs MILP modeling to solve each decomposed sub-demand, like previous efforts, and this modeling inherently does not ensure optimality. On one hand, there is a trade-off between accuracy and efficiency in modeling, and increased scale tends to reduce accuracy. On the other hand, the complexity of real-world networks (*e.g.,* congestion control and jitter) means that theoretical models cannot fully capture all aspects of reality. Despite these limitations, SyCCL is not restricted to any specific modeling techniques. As more advanced and accurate solvers are developed in the future, SyCCL can incorporate them to enhance accuracy.

Furthermore, SyCCL employs carefully-designed pruning strategies to discard non-viable sketches early to reduce search space. Throughout §7.4 we empirically showed that our domain-specific pruning rules do not degrade the performance of the schedules generated. Nevertheless, pruning inherently embodies a speed–accuracy trade-off: by discarding a subset of candidate sketches early, we shorten the synthesis time from hours to minutes, but we also concede the possibility that an unseen sketch could marginally

outperform the retained search space. Therefore, SyCCL exposes user-configurable pruning strategies to balance synthesis time and solution quality. One may even disable pruning to exhaustively enumerate all sketches, at the cost of higher synthesis overhead.

Comparison to expert-optimized schedules. The state-of-the-art approach for collective scheduling in large clusters is for domain experts to manually design communication schedules based on their understanding of the workload and the topology. To achieve optimal performance, in-depth optimizations such as GPU kernel co-design and network tuning are usually performed based on the schedule. SyCCL aims to aid this process by providing insights during the schedule designing stage. We compare the performance of SyCCL's schedule and expert-optimized schedules in Appendix C and showcase how SyCCL can guide the process.

Adaptability to asymmetric collective workloads. Some ML models, such as Mixture of Experts (MOE) [26], use asymmetric collectives, such as Alltoall(v) and AllGather(v), where different GPUs may send or receive data of varying sizes. As a result, the collective symmetry does not hold and SyCCL encounters scalability challenges like existing synthesizers. For these highly irregular and dynamic collective patterns, we believe heuristic-based schedule synthesis is more appropriate than relying on symmetry-aware modeling. Nevertheless, SyCCL can still aid those scenarios by providing a base solution for a symmetric sub-demand in the original collective, or generating sketches without the symmetric constraints. Exploring such heuristic based approaches is left for future work.

Adaptability to heterogeneous clusters. Some studies [21, 36] propose to train ML models using heterogeneous GPUs with asymmetric topologies. For efficient training, these clusters should retain some symmetry. For example, identical GPUs are placed together and connected to the same switches, resulting in a topology which combines symmetric sub-topologies. In this context, SyCCL can further categorize groups in each dimension to reduce search space. Groups within the same category exhibit similar communication behavior. However, when asymmetry increases—such as when the same dimension must be further divided into multiple categories, each with its own set of similar groups—the topology effectively becomes more heterogeneous and higher-dimensional. In highly irregular or unstructured networks that resist such decomposition, the advantages of SyCCL diminish as its core assumption of exploitable symmetry breaks down. We plan to explore methods to extend SyCCL to these more general cases in future work.

Adaptability to dynamic network environments. When network conditions fluctuate (e.g., due to hardware failures or variable bandwidth in shared clusters), SyCCL can regenerate communication schedules if such changes are infrequent and timing constraints are relaxed. However, in more dynamic environments—such as multi-tenant clusters with frequent topology changes or degraded links—SyCCL may be less effective, as discussed above. These scenarios often lead to asymmetric and unpredictable communication patterns, limiting the applicability of symmetry-based schedule synthesis.

Independence from specific GPU architectures. Our synthesizing approach is not tied to any specific GPU architectures, *e.g.*, AMD and NVIDIA. This is because the GPU-related factors are already included in the link performance parameters (*i.e.,* α and β). While our experiments utilized NVIDIA A100 and H800 GPUs, the results obtained are broadly applicable, irrespective of the hardware specifics.

9 Related Work

In this section, we introduce related work on collective communication performance optimization.

Collective schedule optimization. Current research on collective schedule optimization can be divided into three categories. First, open-source CCLs (*e.g.,* NCCL [11], RCCL [14], Gloo [8], and HCCL [5]) implement common collective schedules (*e.g.,* Ring and Tree) and allow for tuning among schedules during runtime. Second, the specific optimizers like Blink [38] and Themis [32] focus on optimizing schedules for specific network topologies or collectives. These two types of work are limited to specific scenarios and cannot adapt to diverse network architectures and models. Third, synthesizers like SCCL [17], TACCL [33], and TECCL [28] automatically synthesize schedules for various network topologies, collective types, and data sizes by encoding the collective communication process. As network size and complexity increase, they face the challenge of an exploding search space and fail to scale. SyCCL is also a synthesizer that addresses this challenge by leveraging topology and collective symmetry.

Multi-collective scheduling. Some research [18, 27, 29, 31, 39, 42] focuses on how to schedule multiple collectives occurring simultaneously or co-optimize computation and communication in a network to optimize the overall performance. SyCCL is complementary to these works.

10 Conclusion

This paper presents SyCCL, a scalable collective communication scheduler synthesizer that automatically synthesizes schedules for production-scale ML jobs. SyCCL leverages topology and collective symmetries to divide large communication demand into smaller sub-demands, and combines a schedule solver and a schedule simulator to synthesize schedules efficiently and accurately. Evaluation results show that SyCCL improves schedule performance by up to 127% while reducing synthesis time by 2 to 4 orders of magnitude compared to state-of-the-art methods.

Acknowledgements

We thank our shepherd, Minjia Zhang, and the anonymous reviewers for their insightful comments. Ennan Zhai is the corresponding author.

References

[1] Double binary tree, 2019. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/.
[2] Multi-rail topology, 2022. https://developer.nvidia.com/blog/doubling-all2all-performance-with-nvidia-collective-communication-library-2-12/.
[3] Astra-sim 2.0, 2024. https://github.com/astra-sim/astra-sim.
[4] Bus bandwith computation, 2024. https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md.
[5] cann-hccl, 2024. https://gitee.com/ascend/cann-hccl.
[6] Clos network, 2024. https://en.wikipedia.org/wiki/Clos_network.
[7] Deepspeed, 2024. https://github.com/microsoft/DeepSpeed.
[8] Gloo, 2024. https://github.com/facebookincubator/gloo.
[9] Megatron-lm, 2024. https://github.com/NVIDIA/Megatron-LM/.

[10] Msccl-executor-nccl, 2024. https://github.com/Azure/msccl-executor-nccl.

[11] Nccl, 2024. https://github.com/NVIDIA/nccl.

[12] Nvidia gb200 nvl72, 2024. https://www.nvidia.com/en-us/data-center/gb200-nvl72/.

[13] Nvlink and nvlink switch, 2024. https://www.nvidia.com/en-us/data-center/nvlink/.

[14] rccl, 2024. https://github.com/ROCm/rccl.

[15] vllm, 2024. https://github.com/vllm-project/vllm.

[16] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared 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 M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher 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. *CoRR*, abs/2005.14165, 2020.

[17] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In Jaejin Lee and Erez Petrank, editors, *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, pages 62–75. ACM, 2021.

[18] 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, ACM SIGCOMM 2024, Sydney, NSW, Australia, August 4-8, 2024*, pages 1–15. ACM, 2024.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[21] Yifan Ding, Nicholas Botzer, and Tim Weninger. Hetseq: Distributed GPU training on heterogeneous infrastructure. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 15432–15438. AAAI Press, 2021.

[22] Jianbo Dong, Bin Luo, Jun Zhang, Pengcheng Zhang, Fei Feng, Yikai Zhu, Ang Liu, Zian Chen, Yi Shi, Hairong Jiao, Gang Lu, Yu Guan, Ennan Zhai, Wencong Xiao, Hanyu Zhao, Man Yuan, Siran Yang, Xiang Li, Jiamang Wang, Rui Men, Jianwei Zhang, Huang Zhong, Dennis Cai, Yuan Xie, and Binzhang Fu. Boosting large-scale parallel training efficiency with C4: A communication-driven approach. *CoRR*, abs/2406.04594, 2024.

[23] William D. Gropp, Ewing L. Lusk, and Anthony Skjellum. *Using MPI - Portable Parallel Programming with the Message-Passing Interface, 3rd Edition*. Scientific and engineering computation. MIT Press, 2014.

[24] Roger W. Hockney. The communication challenge for MPP: intel paragon and meiko CS-2. *Parallel Comput.*, 20(3):389–398, 1994.

[25] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 103–112, 2019.

[26] Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural Comput.*, 3(1):79–87, 1991.

[27] Juncai Liu, Jessie Hui Wang, and Yimin Jiang. Janus: A unified distributed training framework for sparse mixture-of-experts models. In Henning Schulzrinne, Vishal Misra, Eddie Kohler, and David A. Maltz, editors, *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, pages 486–498. ACM, 2023.

[28] Xuting Liu, Behnaz Arzani, Siva Kesava Reddy Kakarla, Liangyu Zhao, Vincent Liu, Miguel Castro, Srikanth Kandula, and Luke Marshall. Rethinking machine learning collective communication as a multi-commodity flow problem. In *Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM 2024, Sydney, NSW, Australia, August 4-8, 2024*, pages 16–37. ACM, 2024.

[29] Kshiteej Mahajan, Ching-Hsiang Chu, Srinivas Sridharan, and Aditya Akella. Better together: Jointly optimizing ML collective scheduling and execution planning using SYNDICATE. In Mahesh Balakrishnan and Manya Ghobadi, editors, *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, pages 809–824. USENIX Association, 2023.

[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 2024, Sydney, NSW, Australia, August 4-8, 2024*, pages 691–706. ACM, 2024.

[31] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. CASSINI: network-aware job scheduling in machine learning clusters. In Laurent Vanbever and Irene Zhang, editors, *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*. USENIX Association, 2024.

[32] 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 Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 581–596. ACM, 2022.

[33] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. TACCL: guiding collective algorithm synthesis using communication sketches. In Mahesh Balakrishnan and Manya Ghobadi, editors, *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, pages 593–612. USENIX Association, 2023.

[34] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[35] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023.

[36] 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 Saurabh Bagchi and Yiying Zhang, editors, *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 563–578. USENIX Association, 2024.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[38] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil R. Devanur, and Ion Stoica. Blink: Fast and generic collectives for distributed ML. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of the Third Conference on Machine Learning and Systems, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020.

[39] 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 Mahesh Balakrishnan and Manya Ghobadi, editors, *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, pages 739–767. USENIX Association, 2023.

[40] Guanbin Xu, Zhihao Le, Yinhe Chen, Zhiqi Lin, Zewen Jin, Youshan Miao, and Cheng Li. {AutoCCL}: Automated collective communication tuning for accelerating distributed and parallel {DNN} training. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 667–683, 2025.

[41] Shiqing Zhang, Yijiao Yang, Chen Chen, Xingnan Zhang, Qingming Leng, and Xiaoming Zhao. Deep learning-based multimodal emotion recognition from audio, visual, and text modalities: A systematic review of recent advancements and future prospects. *Expert Syst. Appl.*, 237(Part C):121692, 2024.

[42] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. Optimal direct-connect topologies for collective communications. *CoRR*, abs/2202.03356, 2022.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A MILP Modeling and Solving

SyCCL synthesizes sub-schedules through MILP modeling, like TECCL [28]. Since TECCL has provided a detailed description of this modeling, we briefly summarize the key components in this section.

A.1 Solving and modeling details

SyCCL uses an MILP formulation to encode the initial status (*i.e.,* which GPUs hold which chunks) and the collective communication demand (*i.e.,* which GPUs require which chunks), the topology, and the transmission process.

Time is divided into equal intervals, called **epochs**, all with a fixed duration denoted as τ. Each communication event (*i.e.,* sending a chunk over a link) must fit into an integer number of epochs. As shown in Figure 18, a chunk is transmitted over a link at the beginning of an epoch. The transmission satisfies: (1) the bandwidth constraint, *i.e.,* a link transmits at most a volume of $\frac{\tau}{\beta}$ within an epoch, where β is the reciprocal of the link's bandwidth, and (2) the latency constraint, the total transfer time t for a chunk of size s over a link is calculated as $\alpha + \beta \cdot s$, where α is a constant transmission delay. The chunk can reach the destination and be sent over the next link only after $\lceil \frac{t}{\tau} \rceil$ epochs.

We then apply the MILP solver to find a solution that minimizes the total number of epochs required to satisfy the demand. Finally, we obtain a schedule which includes the start epoch and end epoch of each communication event (*i.e.,* a GPU sends a data chunk to another GPU).

A.2 Epoch duration τ: a knob to balance the search efficiency and accuracy

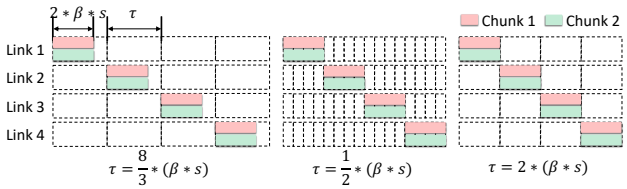
In the MILP modeling, the setting of the τ value is crucial, which is essentially a knob to balance the search efficiency and accuracy.

A larger τ reduces the time required to solve the problem but at the expense of decreased accuracy. Specifically, the number of MILP variables increases linearly with the number of epochs. A larger τ means fewer epochs, simplifying the problem by reducing the number of variables involved, which in turn can speed up solving times. However, a larger τ also leads to more communication events to occur simultaneously within an epoch without any scheduling, which negatively affects accuracy. Previous efforts encode the whole collective and topology into an MILP problem. In such cases, neither a large nor small τ successfully achieves an balance between accuracy and efficiency.

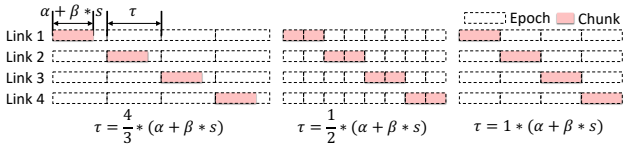
To address this issue, SyCCL introduces a two-step synthesis approach (§5.3), which first exploits coarse-grained solving for quick synthesis and selects the candidates with better performance, and then exploits fine-grained solving for accurate synthesis.

A.3 Determining τ automatically

The multi-dimensional network complicates the selection of an appropriate τ value. On the one hand, τ should satisfy the bandwidth constraint for each link type. As shown in Figure 18(a), τ should



(a) Bandwidth constraint: τ should be a multiplier of $\beta * s$



(b) Latency constraint: τ should be a multiplier of $\alpha + \beta * s$

Figure 18: Epoch duration τ should a multiplier of $\alpha + \beta * s$ and $\beta * s$. Otherwise, the total epoch time does not equal the actual time needed for chunk transmission.

be a multiplier of $\beta * s$, *i.e.,* $\tau = r * \beta * s$, where either r or $1/r$ is an integer. This ensures that the the link's total capacity within an epoch equals to the volume of r chunks ($r = 2$) or $1/r$ epochs are required to transmit one chunk ($r = \frac{1}{2}$). If $r = 8/3$, which is not either an integer or a fraction, the actual transmission time does not equal the total time of the occupied epochs, leading to inaccurate modeling. On the other hand, τ should satisfy the latency constraint for each link type. As shown in Figure 18(b), τ should also be a multiplier of $\alpha + \beta * s$. Under topologies with different link types having different characteristics, it is difficult for existing synthesizers to determine a suitable τ to satisfy the bandwidth and latency constraints for accurate modeling, while keeping τ big enough to prevent a large model size.

SyCCL reduces inaccuracies caused by dealing with multiple link types as each sub-demand in SyCCL exists in a single GPU group, which only involves a single link type. However, the chosen τ must still satisfy both bandwidth and latency constraints for that link type.

To address this, SyCCL introduces an auxiliary variable E to adjust the trade-off between accuracy and efficiency. A larger E means better efficiency, while a smaller E improves accuracy. Based on the selected E value, SyCCL automatically determines the optimal τ value which satisfy both constraints. Specifically, SyCCL prioritizes the bandwidth constraint by setting $\tau = r * \beta * s$, where r is a parameter which will be determined later, and either r or $\frac{1}{r}$ is an integer. To address the latency constraint, SyCCL defines a function $f(r) = \frac{\alpha + (\beta * s)}{r * \beta * s}$, with $\lceil f(r) \rceil$ representing the number of epochs required to transmit a single chunk. The goal is to minimize $g(r) = \lceil f(r) \rceil - f(r)$ to meet the latency constraint. SyCCL fine-tunes the trade-off between accuracy and efficiency by setting $\lceil f(r) \rceil = E$, where E is a adjustable parameter and finding r which minimizes $g(r)$.

B Other GPU cluster topology examples

Figure 19 shows a larger multi-rail topology cluster, consisting of seven 4-GPU servers connected to four leaf switches. GPUs with the same intra-server index are connected to the same leaf switch. For example, GPUs 0, 4, 8, ... are connected to the first leaf switch. This

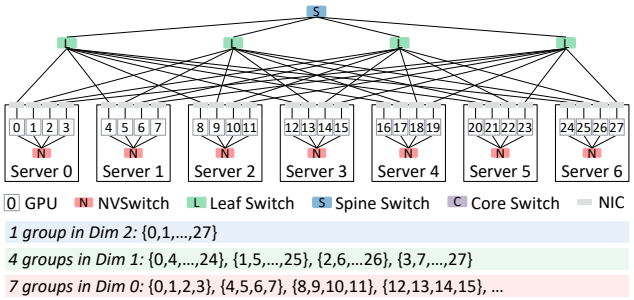


Figure 19: Example of a larger multi-rail topology.

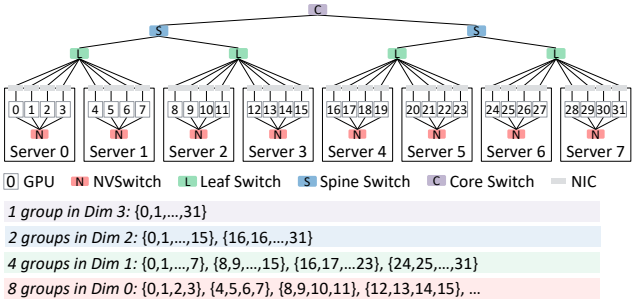


Figure 20: Example of a Clos topology.

topology is organized into three dimensions. Dimension 0 contains seven GPU groups corresponding to the seven servers. Dimension 1 contains four GPU groups, where each groups includes GPUs connected to the same switch. Dimension 2 contains a single group that encompasses all GPUs connected through a spine switch.

Figure 20 shows a Clos topology, which consists of eight 4-GPU servers. In this topology, each pair of servers connects to the same leaf switch, and each pair of leaf switches connect to the same spine switch. Additionally, two spine switches connect to a core switch. This topology includes four dimensions. Dimension 0 contains eight GPU groups corresponding to eight servers. Dimension 1 contains four GPU groups, each including GPUs connected to the same leaf switch. Dimension 2 contains two groups, and each group consists of GPUs connected to the same spine switch. Dimension 3 contains one group that includes all GPUs connected by a single core switch.

C Comparison to expert-optimized schedules

A common approach for collective scheduling in large clusters is for domain experts to manually design communication schedules based on their understanding of the workload and the topology. After designing the collective schedule, they often implement it at the GPU kernel level, coupled with in-depth optimizations like network tuning. Since SyCCL focuses on optimizing the schedule itself, we compare results only at the schedule level. Nevertheless, compared to other MILP based methods where the complex schedules generated by the solver may not be easily understandable, we expect SyCCL's high-level sketches to be readable by users and capable of being further implemented and optimized manually for extreme performance.

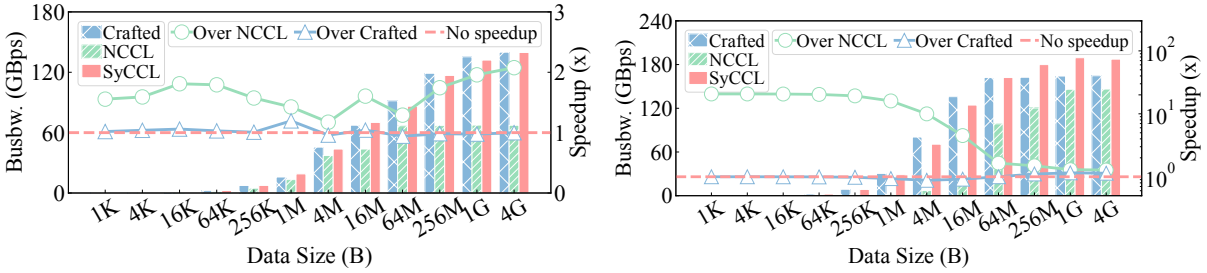
We use `AllGather` as an example here. We generate three commonly used schedules: ring schedule, direct schedule, and hierarchical schedule. For the ring schedule, we try to utilize all available bandwidth in the topology by using multiple rings to cover all inter-machine links. Direct schedule sends all data chunks directly to other GPUs in order. Hierarchical schedule decomposes the original `AllGather` into multiple smaller ones performed inside each topology group. We implement the hierarchical schedule inside one collective kernel, instead of using multiple collective calls, to achieve the best performance. The transmission order of each schedule is carefully designed to avoid contention. For each collective size, we collect the best performance among all hand-crafted schedules and compare it with SyCCL's output.

Figure 21(a) shows that SyCCL's performance is similar to that of hand-crafted schedules on the 16-A100 testbed. Upon inspection, we found that SyCCL uses the same hierarchical schedule as the hand-crafted schedules for large data sizes, and the same direct schedule for smaller data sizes.

Figure 21(b) shows that SyCCL outperforms hand-crafted schedules on the 64-H800 testbed for larger data sizes. This improvement is attribute to the bandwidth ratio between inter-machine and inter-machine links in the 64-H800 testbed not aligning well with the hand-crafted hierarchical schedule. In contrast, SyCCL utilizes sketch combination and alternative sketches to almost perfectly match the bandwidth ratio, resulting in a higher utilization.

We also found out that the best sketch combinations in the 64-H800 testbed mainly utilizes an alternative hierarchical `AllGather` schedule. The conventional hierarchical `AllGather` schedule involves sending every chunk to all GPUs in the same rail first, followed by sending the chunk to all other GPUs in the same machine, or vice versa. SyCCL utilized an sketch where a chunk is first sent to one other GPU in the same machine. Then, the two GPUs send the chunk to other GPUs in the same rail. Finally, the two GPUs holding the chunk in each machine send it to other GPUs in the same machine, with each GPU issuing three sends.

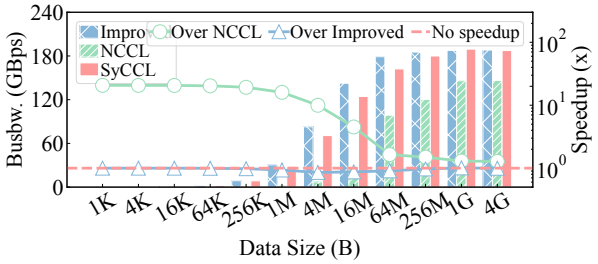
As this sketch matches the bandwidth ratio of our H800 testbed well, even without sketch combination, we tried to additionally hand-craft such schedules. Figure 22(a) shows that after incorporating this hand-crafted schedule, we achieve comparable performance to SyCCL for large sizes. This approach provides the advantage of a more structured schedule with a guarantee of no contention, compared to potentially complex and inaccurate schedules generated by the MILP solver, which could explain SyCCL's performance degradation in some cases. We expect such workflow of optimizing the best sketch combinations being the best practice of using SyCCL.



(a) `AllGather` on 16 A100 GPUs

(b) `AllGather` on 64 H800 GPUs

Figure 21: Hand-crafted schedule performance on A100 and H800 topologies.



(a) `AllGather` on 64 H800 GPUs

Figure 22: Improved hand-crafted schedule performance on H800 topology.