OptCCL: Scalable Synthesis of Optimal Collective Communication Algorithms

Richard Shapley
Cornell University

Rachit Agarwal
Cornell University

David Shmoys
Cornell University

Abstract

We present OptCCL, a technique to synthesize collective communication algorithms that are optimal for a given host and network hardware and topology. OptCCL is general: it enables synthesizing optimal algorithms for all existing collectives, for all existing hardware, and even for multiple concurrent collectives sharing host and network resources. And yet, OptCCL is scalable: it synthesizes optimal algorithms for hundreds of GPUs within tens of minutes.

CCS Concepts

• **Networks → Network algorithms**; **Data center networks**; • **Theory of computation → Design and analysis of algorithms**; • **Computing methodologies → Machine learning**.

Keywords

Machine Learning, Collective Communication, Scheduling

ACM Reference Format:
Richard Shapley, Rachit Agarwal, and David Shmoys. 2026. OptCCL: Scalable Synthesis of Optimal Collective Communication Algorithms. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3789240.3829207

1 Introduction

Collective communication in machine learning (ML) training and inference refers to inter-GPU communication for synchronizing gradients, parameters, and optimizer states. Recent studies have established that performance of collective communication algorithms can significantly impact end-to-end performance of ML (training and inference) jobs as well as GPU resource utilization [4, 11, 19, 26].

Designing optimal collective communication algorithms (defined formally in §2, but intuitively, one that minimizes the time taken to complete the collective) for general host and network hardware and topologies remains an open problem. The problem is further exacerbated by perpetual across-the-stack advances in ML deployments: to accommodate larger model sizes, ML deployments are using increasingly large numbers of GPUs [8, 11–13]; to improve hardware utilization, ML deployments are evolving from single-collective jobs to multi-collective jobs to multi-collective multi-tenant jobs sharing host and network resources [4, 23]; and both intra-host and inter-host network interconnects are evolving in terms of scale, hardware functionalities and topologies [11, 19, 24, 25]. With every advance—in scale, in concurrent collectives, in hardware functionality, in intra-host and inter-host topologies—it is imperative to design (presumably) new collective communication algorithms that are optimal for the new host and network hardware and topology.

Put together, the importance of collective communication and perpetual across-the-stack advances in ML deployments lead to a new exciting research direction: synthesizing collective communication algorithms that are optimal with respect to a given host and network hardware and topology. Existing works along this direction give up on optimality [5, 9, 10, 14, 15, 20, 23, 25], scalability [3, 15], and/or generality in terms of the host and network hardware that they can model (Table 1, §8).

This paper presents OptCCL, a technique to synthesize collective communication algorithms that are optimal for a given host hardware and network topology. OptCCL is general: it enables synthesizing optimal algorithms for a general class of collectives (including all existing collectives), for a large class of existing and future hardware (*e.g.*, host hardware with and without specialized intra-host interconnects, heterogeneous network bandwidths within and across hosts, network switches with and without multicast functionality, network switches with and without computation capabilities, etc.), and for multiple concurrent collectives sharing host and network resources.

OptCCL embodies a technical result that establishes a powerful structure in optimal collective communication algorithms: for the objective of minimizing the completion time for collective(s), it is possible to decouple the spatial subproblem of determining the path taken by data between GPUs from the temporal subproblem of scheduling data at each link (*e.g.*, at the granularity of fixed-sized chunks, as in modern collective communication libraries [9, 10]), while maintaining optimality. More specifically, we establish that, when the objective is to minimize the completion time for collective(s), the core optimization problem underlying the goal of synthesizing optimal collective communication algorithms is the spatial subproblem—once we identify the path taken by the data in an optimal solution, scheduling data at each individual link while maintaining optimality is straightforward. This result enables OptCCL (and any other technique) to transform an optimal solution to the spatial subproblem into an optimal end-to-end collective communication algorithm.

OptCCL builds upon the broad class of classical multicommodity flow-based formulations, but uses a novel formulation specialized for collectives. Specifically, rather than treating each sender as a commodity and modifying flow conservation constraints to capture collective structure as would be natural in classical multicommodity flow-based formulations, OptCCL treats each sender-destination pair as a separate commodity. This allows OptCCL to use standard flow conservation constraints; to capture the collective structure,

SIGCOMM '26, Denver, CO, USA

ACM ISBN 979-8-4007-2467-1/26/08
https://doi.org/10.1145/3789240.3829207

Table 1: Existing techniques on synthesizing collective communication algorithms. A ⋆ indicates that the corresponding work should be able to support the particular property with natural extensions. Detailed discussion in §8.

	Optimality	Scalability	Generality		
			Set of collectives	Hardware	Multiple collectives
NCCL [9], RCCL [10]	✗	✓	✓	✓	✓
Blink [23]	✗	✓	✗	✗	✓
TACCL [20]	✗	✓	✓	✗	✗
TACOS [25]	✗	✓	✗	✗	✓
SyCCL [5]	✗	✓	✓	✗	✗
TE-CCL [15]	✓	✗	✓	✓⋆	✓⋆
SCCL [3]	✓	✗	✓	✓⋆	✓⋆
ForestColl [27]	✓	✓	✗	✗	✗
OptCCL	✓	✓	✓	✓	✓

OptCCL instead introduces capacity constraints on certain subsets of flows. This formulation admits a nicely structured linear programming (LP) relaxation. Interestingly, we show that for a class of host and network hardware and topologies (that we precisely characterize and that includes many realistic deployments), this LP relaxation already synthesizes optimal solutions. For inputs outside this class, OptCCL iteratively strengthens the relaxation to synthesize optimal collective communication algorithms.

To further improve the time taken to synthesize an optimal solution, OptCCL introduces a technique to exploit the symmetry in the underlying host and network hardware and topology (and the collective). We prove that one of the optimal solutions to the formulation has a particular symmetry, defined with respect to the symmetry in the underlying host and network hardware and topology and the collective; thus, the time taken to synthesize an optimal solution can be improved by reducing the search space to explore only such symmetric solutions.

OptCCL's formulation enables simultaneous optimization for multiple performance objectives. Specifically, for a given input, there may be multiple solutions all of which are optimal with respect to one performance objective (*e.g.*, latency and/or throughput) but may have different performance with respect to other objectives (*e.g.*, amounts of aggregate data transfers). OptCCL enables synthesizing collective communication algorithms that optimize for secondary objectives, while being optimal for the primary objective. This feature enables OptCCL to synthesize optimal algorithms for the case of multiple concurrent collectives within or across tenants (optimality is defined in terms of the primary objective of time taken to complete all collectives).

We evaluate OptCCL across multiple host and network hardware and topologies, with single collectives, and with multiple collectives. Across most of the evaluated scenarios, state-of-the-art techniques are either unable to synthesize algorithms for more than 16–64 GPUs within 3 hours [5, 15] or synthesize algorithms that can be more than an order of magnitude worse than optimal [20]. OptCCL significantly improves upon the current state of affairs: it synthesizes the optimal algorithm for hundreds of GPUs within tens of minutes for all evaluated scenarios.

An implementation of OptCCL is available at https://github.com/ml-collectives/optccl.

2 Preliminaries

Input topologies consist of components (GPUs, CPUs, memory, and switches) and a set of links connecting components. Each (full-duplex) link incident on a GPU, CPU and/or switch has a bandwidth constraint. The links incident on memory components are non-full-duplex [22]; these links have an aggregate bandwidth constraint to accurately capture memory bandwidth.

We define four operations on data. Data can be *stored*. Data can be *transmitted*, that is, the data is forwarded on one of the outgoing links. Data can be *copied*, that is, potentially multiple copies of data at any component may be transmitted over different outgoing links. Data can be *reduced* according to a reduce operation, where two (or more) pieces of data, each of size M, are combined into a single piece of data of size M.

Our formulation enables each component to be capable of doing any subset of the above operations (§5.1). Our formulation also enables an extremely general definition of collectives—informally, any collective that can be described in terms of the four data operations defined above can be incorporated within our formulation (§5.2). Our formulation also enables multiple such collectives concurrently coexisting within the system, within and across tenants (§5.3).

From topologies to graphs. We model the input topology as a directed graph G; see Figure 1 for an example. Each vertex in G corresponds to a component. We let V_b, V_c and V_r be the set of vertices corresponding to components that can store, copy and reduce data, respectively, and V_o and V_d be the origin and destination vertices determined by the collective and the components on which the collective runs. Importantly, our formulation allows any component to be in one or more of the vertex categories (*e.g.*, V_b, V_c, V_r, etc.).

Each edge in G corresponds to a link in the topology; if the link is bidirectional, we create an edge in both directions. To capture the ability of vertices in V_b to store data, for every $v \in V_b$, we also create a self-loop with no bandwidth limit. We capture all bandwidth limits through a single mechanism: for any set of edges L with an aggregate bandwidth limit, we define $b(L)$ to be the bandwidth for those edges, which represents how much data can be sent through the edges in L in one unit of time. We let $\mathcal{L}$ be the collection of all edge sets that have a bandwidth bound. Taking L to be a single edge recovers the bandwidth limit for full-duplex links, and taking

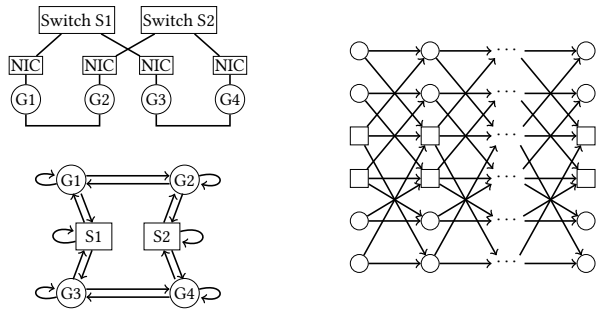

Figure 1: (top left) shows an input topology with 4 GPUs (G1, G2, G3 and G4) across two hosts—each with specialized intra-host interconnect—connected via an inter-host interconnect organized along a rail-based topology. (bottom left) depicts the transformation from the input topology to OptCCL's directed graph representation. (right) shows the corresponding time-expanded network. See §2.

L to be the set of edges incident to a memory component recovers memory bandwidth. See Figure 1 for an example.

Feasible algorithms, performance objectives, and optimal algorithms. OptCCL takes as input the topology (along with the capability of each component), the collective definition, the collective data size and a user-defined chunk size. OptCCL outputs a collective communication algorithm that specifies a set of chunks, a path for each chunk, and for each link along the path, an ordering (or, the transmission times) for chunks transmitted on that link. A collective communication algorithm is said to be feasible if it correctly executes the desired collective operations, while respecting the bandwidth constraints along each link (or set of links).

Our primary objective is end-to-end collective completion time, defined as the time elapsed from the start of the collective (when the first chunk is transmitted) to the time when the collective is completed (the last chunk reaches its destination). A collective communication algorithm is said to be optimal if, for a given input, it achieves the minimum collective completion time over all feasible collective communication algorithms for that input.

There may be multiple optimal algorithms for any given input. OptCCL allows synthesizing, among all such algorithms, an algorithm that also optimizes for a secondary objective. Any secondary objective that is a weighted sum of the amount of data transmitted along all links in the topology can be chosen. To realize this, internal to our formulation, we also use costs c_e for every edge $e \in E$; these costs indicate the cost of transmitting one unit of data along respective edges, and allow us to simultaneously optimize for the secondary objective while guaranteeing optimality with respect to the primary objective. For example, if we want our secondary objective to minimize the total data entering the network, we can set $c_e = 1$ for edges between NICs and switches, and $c_e = 0$ for all other edges. By default, we will set our secondary objective to minimize the total data uploaded to and downloaded from GPUs; thus, we set $c_e = 1$ for edges incident to a GPU, and $c_e = 0$ for all other edges. We will discuss how this secondary objective allows us to efficiently optimize for multiple concurrent collectives.

Time-expanded networks. To model routing of data across components and links over time, we use the representation of so-called time-expanded networks; see Figure 1 for an example. Let G be the input topology, and let T be a constant (we provide an interpretation for T inline, but intuitively it will represent some notion of time). Then, we construct a time-expanded network G^T with T "layers" as follows. We create $(T+1)$ copies of each vertex v in G, labeled as $(v,0), (v,1), \ldots, (v,T)$. For each edge (v_1, v_2) in G, we draw directed edges from vertex (v_1, t) to $(v_2, t+1)$ for $t \in \{0, \ldots, T-1\}$. For vertices $v \in V_b$ that have a self-loop, we also create an edge from (v,t) to $(v,t+1)$. We use V^T and E^T to refer to vertices and edges, respectively, of G^T. We will use the following notation to simplify conversions between G and G^T. For edge $e^T = ((i,t),(j,t+1)) \in E^T$, we use $\tau(e^T) = t$ to denote the time at the tail of e^T and $\mu(e^T) = (i,j) = e$ to denote the corresponding edge in G. Similarly, we use $\kappa(e,t) = \kappa((i,j),t) = ((i,t),(j,t+1)) = e^T$ to convert from e to the corresponding edge in G^T at step t. And analogously for vertices: for $v^T = (v,t) \in V^T$, let $\tau(v^T) = t$ and $\mu(v^T) = v$. We will not describe the bandwidths for edges in G^T; instead, when needed, we will refer back to edges in G.

3 The Optimality-Scalability Tussle

We start with a general formulation for synthesizing optimal collective communication algorithms. The formulation *jointly* solves two problems: routing of data between every sender-destination pair (potentially along multiple paths), and scheduling of data at each individual link at each individual step. The formulation produces optimal solutions, but is not scalable in general. We then consider a natural linear programming relaxation of the formulation that also jointly solves the routing and the scheduling problem; we discuss that this relaxation improves scalability compared to the exact formulation, but struggles with the same tussle: it is possible to achieve either optimality or scalability, but not both. These formulations provide the foundation for our optimal, scalable and practical solution in the next section. For brevity, we focus on the AllGather collective for the case where all components can perform data copies; in §5, we generalize this to incorporate scenarios where some components cannot perform copies, and to collectives that require reductions.

Our formulation builds upon the classical multicommodity flow formulations [2, Ch.17]: we are given a set of distinct commodities; each commodity has an origin and a destination vertex, and a positive demand indicating how much flow must be sent from the origin to the destination vertex. Each commodity/flow must individually satisfy flow conservation constraints; collectively, they must satisfy capacity constraints along each shared edge.

An exact integer programming formulation. For this formulation, we will assume that all transmissions happen in a sequence of T discrete time steps, each of size (or alternatively, of length of time) s. Consider a fixed number of steps T and step size s. We start with the case when all data between a sender-destination pair traverses along a single path. Our formulation creates a commodity between every origin $V_o^T = \{(v,0) : v \in V_o\}$ and every destination $V_d^T = \{(v,T) : v \in V_d\}$ vertex. Specifically, our formulation uses flow variables $f_{e,o,d}$ for every edge $e \in E^T$, and every commodity with origin-destination pair (o,d). For these commodities, flow conservation constraints on the f variables as in classical multicommodity flow formulation ensure that data is routed correctly

through G^T:

$$\sum_{e\in\delta^-(v)} f_{e,o,d} - \sum_{e\in\delta^+(v)} f_{e,o,d} = 0 \quad \text{for } o \in V_o^T, d \in V_d^T, v \in V^T \setminus \{o,d\}; \tag{1}$$

$$\sum_{e\in\delta^-(o)} f_{e,o,d} - \sum_{e\in\delta^+(o)} f_{e,o,d} = 1 \quad \text{for } o \in V_o^T, d \in V_d^T; \tag{2}$$

$$\sum_{e\in\delta^-(d)} f_{e,o,d} - \sum_{e\in\delta^+(d)} f_{e,o,d} = -1 \quad \text{for } o \in V_o^T, d \in V_d^T. \tag{3}$$

To capture collective structure, our formulation introduces additional *usage variables* $w_{e,o}$ for every edge $e \in E^T$ and every origin vertex $V_o^T = \{(v,0) : v \in V_o\}$ in G^T; these usage variables will allow us to capture the collective structure by constraining the bandwidth required to support the corresponding flows in our formulation. [1]

The relationship between f and w is specific to the collective. For example, for the AllGather collective, data sent to all destinations from the same origin is identical; when all components can perform data copies, no component must transmit multiple copies of the same data along a particular edge in an optimal solution. Therefore, it is possible to lower bound the usage w with the constraint:

$$w_{e,o} \geq f_{e,o,d} \qquad \text{for } e \in E^T, o \in V_o^T, d \in V_d^T. \tag{4}$$

Importantly, instead of setting our usage to be the sum over all flows, as would be natural, we allow our usage to be less to account for the fact that the flows may share data. We just ensure that our usage is at least the maximum flow amount, which is achieved by the constraint. Finally, for every set of links L with corresponding bandwidth limitation $b(L)$, the total usage on all links in L by data from all origins must be restricted by $s \cdot b(L)$ for each step of size s:

$$\sum_{e\in L}\sum_{o\in V_o^T} w_{\kappa(e,t),o} \leq sb(L) \quad \text{for } L \in \mathcal{L}, t \in \{0,\ldots,T-1\}. \tag{5}$$

Using the per-edge cost (for secondary objectives, §2) and imposing integrality constraints on f, we get:

$$\min_{f,w} \quad \sum_{e\in E}\sum_{t=0}^{T-1} c_e \sum_{o\in V_o^T} w_{\kappa(e,t),o} \quad \text{s.t.} \quad (1)-(5);\ f \in \{0,1\}, w \geq 0. \tag{6}$$

Importantly, the objective in the formulation does not explicitly use the primary objective (collective completion time) but rather uses the secondary objective; this is because, for a fixed s and T, the collective completion time is predetermined, and the formulation finds the solution with minimum secondary objective value among all solutions with the predetermined collective completion time. As we discuss below, the primary objective is optimized by solving the above formulation over all possible values of s and T.

[1] This is the core difference between OptCCL and prior attempts on synthesizing collective communication algorithms based on multicommodity flow-based formulations, *e.g.*, [15]. Prior attempts modify the classical multicommodity flow formulation to capture the collective structure within the flow constraints. OptCCL uses the standard flow constraints, and captures the collective structure in these usage variables.

It is straightforward to modify the above formulation to consider solutions where data between a single sender-destination pair may traverse across multiple paths. Specifically, let H be the number of uniform-sized chunks originating at a sender. The formulation now creates H commodities between every origin-destination pair, and each commodity is treated independently except for in the bandwidth constraints and in the objective function. The bandwidth constraints are scaled appropriately so that one unit of flow now corresponds to $1/H$ units of data. The formulation, denoted as MILP(17), is provided in [21, A.1].

Theorem 1. *MILP*(6) *and MILP(17) are exact formulations for AllGather for single and for H chunks, respectively.*

The proof of Theorem 1 is in [21, A.2]. As we will discuss in §5, MILP(6) and MILP(17) formulations are very general—they enable each component to be able to do any operation, bandwidth constraints on edges or set of edges, and a general class of (multiple concurrent) collectives. They also directly output the route taken by each chunk and the optimal schedule: the chunk that is transmitted on each link at each time step. However, the computational complexity of generating an optimal solution is high: not only is solving MILP computationally hard, synthesizing an optimal collective communication algorithm requires solving many MILP instances to explore optimal values of s and T. [2]

A linear programming formulation. We now consider a natural step-based relaxation of the integer program MILP(6), one that relaxes the integrality constraints on f:

$$\min_{f,w} \quad \sum_{e\in E}\sum_{t=0}^{T-1} c_e \sum_{o\in V_o^T} w_{\kappa(e,t),o} \quad \text{s.t.} \quad (1)-(5);\ f, w \geq 0. \tag{7}$$

We use the notation LP(7)$[s,T]$ to indicate formulation (7) with step size s and number of steps T when s and T are not obvious. This formulation maintains the generality of MILP(6) and MILP(17); and, while the process of creating a chunk-based schedule is less obvious than for MILP(6) and MILP(17), it is possible (we discuss in §4).

The core challenge is the tussle between optimality and scalability. Specifically, allowing fractional flow values means that we no longer require integer flow constraints and thus LP(7) can be solved in polynomial time [2, Ch.17] for any given step size s and number of steps T; however, the absolute time taken to solve LP(7) grows quickly with T—this is because both the size of the time-expanded network G^T and the number of variables in the LP grow roughly linearly with T. Thus, we want a small T. Ideally, we also want a small s: this enables efficient pipelining of data along the path, keeping every link continuously busy; indeed, suboptimality of imperfectly pipelined solutions grows linearly with s (or, more precisely, inversely proportional to T since optimal end-to-end

[2] An optimal solution may require sending more than one chunk per time step s; thus, identifying an optimal value of s requires exploring potentially $|E|$ many values (one for each unique link bandwidth). An obvious upper bound on T is the product of total number of chunks (the ratio of collective data size to chunk size) and number of components, since any given chunk must never traverse a vertex in the graph multiple times in an optimal solution; we can search over all values of T by doubling until we find a feasible solution, then doing binary search to determine the smallest feasible value. Synthesizing the optimal solution thus requires solving the MILP as many as $|E| \times \log(|V| \times (\text{collective data size}/\text{chunk size}))$ times.

collective completion time is $s \times T$). Unfortunately, since s and T are tightly coupled in these formulations, we can achieve either scalability (a small T) or optimality (a small s), but not both.

Take-away. The two formulations above—the exact integer programming formulation and the linear programming relaxation—suffer from a tussle similar to existing works [3, 5, 15, 20]: it is possible to achieve either optimality or scalability, but not both. We believe this tussle is fundamental to an inherent property of all of these formulations: by incorporating individual steps within the formulation (*e.g.*, via variable s), they tightly couple the spatial problem of routing data between sender-destination pairs (*e.g.*, via constraints (1)–(4) that capture the path taken by data) with the temporal problem of scheduling data on individual links at each time step (via constraint (5) that captures the scheduling of data while maintaining bandwidth constraints in each step). In the next section, we present a formulation that is able to break this tussle, while maintaining the generality of the above formulations.

4 OptCCL

This section presents a formulation that maintains the generality of MILP(6) and MILP(17), and yet, enables scalable syntheses of optimal collective communication algorithms. The key technical result underlying this formulation is a structure in optimal collective communication algorithms: for our primary objective of collective completion time, it is possible to decouple the spatial subproblem of determining the path taken by data between sender-destination pairs from the temporal subproblem of scheduling data at each link at each time step, while maintaining optimality. In fact, we will establish a more powerful result: for our primary objective of collective completion time, the core optimization problem underlying the goal of synthesizing optimal collective communication algorithms is the routing problem—once we identify the path taken by the data between all sender-destination pairs in an optimal solution, scheduling data at each individual link in each individual time step while maintaining optimality is straightforward.

OptCCL will decouple the spatial and the temporal subproblems—it will generate the path taken by data in an optimal solution using a formulation that combines constraints (1)–(4) along with mild bandwidth constraints sufficient to later solve the temporal problem, while maintaining optimality. Specifically, rather than enforcing bandwidth constraints for each individual step as in (5), our formulation will now impose constraints on *average bandwidth* used by the solution for any link over the entire lifetime of the collective. We will then show that it is possible to take an optimal routing solution—the path taken by data between every sender-destination pair in a manner that bandwidth constraints are satisfied *on an average* rather than on a per-step basis—and construct a step-based schedule over uniform-sized chunks (that is, scheduling data at each individual link at each individual time step at the granularity of uniform-sized chunks) while maintaining feasibility and optimality.

4.1 Decoupling data routing and scheduling

Let T be an upper bound on the total number of transmissions, across all sender-destination pairs, any piece of data might require to reach from its sender to its destination (note that data may not necessarily traverse along the shortest path). Let R be a desired bound on the completion time of our solution. Then we modify constraint (5) in LP(7) to require that bandwidth constraints are imposed on average bandwidth during the lifetime R of the collective:

$$\sum_{t=0}^{T-1} \sum_{e \in L} \sum_{o \in V_o^T} w_{\kappa(e,t),o} \leq Rb(L) \qquad \text{for } L \in \mathcal{L}. \tag{8}$$

Then our new formulation becomes:

$$\min_{f,w} \quad \sum_{e \in E} \sum_{t=0}^{T-1} c_e \sum_{o \in V_o^T} w_{\kappa(e,t),o}$$
$$\text{s.t.} \quad (1)-(4), (8);\ f, w \geq 0. \tag{9}$$

We use the notation LP(9)$[R, T]$ to indicate formulation (9) with desired completion time R and number of steps T.

The above formulation, unlike LP(7), solves only the spatial subproblem: it determines the amount of data transmitted on each path between any sender-destination pair in a manner that bandwidth constraints on each link are satisfied on an average, but does not specify the schedule for each chunk of data on each link at each time step. Nevertheless, we prove the following theorem:

Theorem 2. *LP*(9) *is equivalent to LP*(7) *in the following sense: given a solution to LP*(9)$[R_1, T_1]$*, there are values* s_1 *and* T_1' *such that we can construct a solution to LP*(7)$[s_1, T_1']$ *of equal cost and with completion time* $s_1 T_1' < R_1 + \epsilon$ *for any* $\epsilon > 0$*. And given a solution to LP*(7)$[s_2, T_2]$ *with completion time* $s_2 T_2$*, we can construct a solution to LP*(9)$[s_2 T_2, T_2]$ *of equal cost.*

The full proof of Theorem 2 is in [21, A.3]. This theorem establishes that we can convert between solutions for LP(7) and LP(9) with identical objective cost and identical completion time (up to an arbitrarily small ϵ term). Thus, we can find an optimal solution for LP(7)—over all possible choices of s—by solving LP(9). This is powerful because it demonstrates that the optimal collective communication algorithm (in fact, the entire solution space) does not change even if one ignores the problem of scheduling individual data chunks; solving the spatial subproblem is the crux.

Choosing R and T. We first determine R, the optimal completion time for our collective algorithm, which we can do independently from T. A simple upper bound for R is the size of all the output data divided by the smallest bandwidth in the input; since choosing R too small makes the LP infeasible, it is possible to compute it via bisection search. We instead use a better strategy from [27]: R can be computed directly by solving several maximum flow problems on a modified version of G (details in [21, A.4]). Once R is determined, we determine T that upper bounds the length of the longest path that any piece of data travels in an optimal solution. A trivial upper bound on this value is the number of edges in G. This means that via repeated doubling and bisection search, we can determine the best value of T in roughly $\log |E|$ attempts.

4.2 Synthesizing an optimal routing solution

LP(9) (as well as LP(7)) are relaxations of the exact formulation MILP(6) and MILP(17); thus, a feasible solution to the LP does not necessarily imply an optimal routing solution. To understand this, let us call a solution *LP-feasible* if it satisfies the constraints of the LP at hand. The solution is also *algorithm-feasible* if the schedule

it prescribes (using the technique discussed in §4.3) satisfies two conditions: every piece of data reaches all of its destinations, and the data crossing any link in any step never exceeds the bandwidth of the link. Below, we characterize host and network hardware and topologies for which an LP-feasible solution to LP(9) is also algorithm-feasible (that is, provides an optimal routing solution); and when such is not the case, a mechanism to synthesize an algorithm-feasible solution from a given LP-feasible solution.

Algorithm-feasibility is exactly what MILP(6) and MILP(17) encode: thus, the notions of LP-feasibility and algorithm-feasibility coincide for MILP(6) and MILP(17). However, they can diverge for our relaxations. This is because usage variables in constraint (4) are allowed to be less than the sum of the corresponding flow values based on an optimistic assumption about the structure of the optimal solution: all flows sharing that link carry identical data and can be merged into one transmission. When the optimal solution does not have that structure, the usage variables may underestimate the bandwidth required (by the step-based schedule corresponding to the LP-feasible solution) on one or more edges.

LP(9) already synthesizes optimal routing solutions for a large class of host and network hardware and topologies. We now establish that, for a class of input topologies that we call ideal topologies, the LP(9) relaxation does produce an algorithm-feasible solution. We define ideal topologies to be the class of topologies in which (1) in an optimal solution, all data traverse no more than four edges in G (that is, four physical links in the network); and (2) the bandwidth limits and objective costs are imposed on inter-host links (that is, links between switches and NICs). For ideal topologies, we prove the following theorem:

Theorem 3. *For ideal topologies, LP(9)-feasible solutions are also algorithm-feasible. Thus, for ideal topologies, LP(9) synthesizes an optimal routing solution.*

The full proof is in [21, A.5]. We provide some intuition on ideal topologies. Many existing deployments that use specialized intra-host interconnects (*e.g.*, NVLinks) have a NIC connected to each GPU with bandwidth much lower than the bandwidth of NVLinks connected to the GPU; in such cases, the bandwidth limits and objective costs are imposed on inter-host links. Now, suppose such is the case. Then, Theorem 3 implies that any topology where all paths between each pair of GPUs have no more than four physical links is an ideal topology. For example, a deployment where all GPUs within a host are connected via NVLinks and where GPUs across hosts are connected along a rail-based network topology, with a single layer of rail switches, satisfies this condition when all data transfers use the shortest path since such paths have at most four links: GPU→NIC→switch→NIC→GPU.

Synthesizing optimal routing solutions for non-ideal topologies. For non-ideal topologies, an LP(9)-feasible solution *may* not be algorithm-feasible. We can nonetheless synthesize an optimal algorithm by repeatedly strengthening the relaxation, using the value of algorithm-infeasible optimal solution as a lower bound.

We first solve LP(9), and check if the resulting solution corresponds to an algorithm-feasible one of the same cost. To perform this check, we convert the routing solution into a concrete set of trees (we describe this construction in §4.3). Each tree describes how data from one origin travels to all its destinations. Whenever a vertex in the tree has multiple outgoing edges, the data is considered to be copied. We then check for each edge whether the sum of flow values (recall, an edge may be in multiple trees) is no more than the corresponding usage value. If the check succeeds, the solution is algorithm-feasible and we have found an optimal routing solution—indeed, the optimal value to any relaxation is a lower bound on the optimal algorithm objective value.

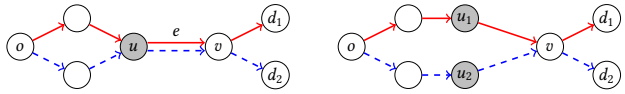

Figure 2: An example to illustrate scenarios where LP-feasible solutions may not be algorithm-feasible (left) and the vertex duplication mechanism used in OptCCL to resolve such cases (right).

When the check fails, there is some edge $e = (u, v)$ in G^T whose trees after converting the routing solution carry distinct pieces of data from a common origin o. Figure 2 (left) shows such a case in which two flows send one unit on edge e, but the LP assigns only one unit of usage value under the optimistic assumption that these data can be transmitted in a single unit (and be copied later). However, the two flows reached e along different paths and carry distinct data, so the optimistic assumption made by the LP does not hold: e must actually send two units.

To prevent this phenomenon from occurring, we strengthen the formulation: we duplicate the offending vertex u at the tail of e, splitting it into copies indexed by the destinations its outgoing data is bound for, so that the current solution will no longer be LP-feasible (*e.g.*, Figure 2 (right) splits u into u_1, u_2, forcing the two flows onto separate edges). [3] After duplicating vertices for every edge whose usage was less than the sum of the corresponding flow values, we end up with a slightly larger time-expanded network and re-solve LP(9) on it. We repeat until the solution is algorithm-feasible, and since all our relaxations are lower bounds, it is also provably optimal. We emphasize that this iterative strengthening procedure is only performed if necessary.

4.3 Synthesizing an optimal schedule

The optimal routing solution from the previous subsection specifies, for each sender-destination pair, the set of paths taken by data and the amount of data transmitted on each path. In this subsection, we describe an efficient mechanism to convert the optimal routing solution into an optimal schedule that specifies, for each sender-destination pair, data transmitted on each link at each time step.

The mechanism takes as input the host and network hardware and topology, the routing solution from the previous subsection, as well as user-defined chunk size and total data size. To maintain practicality (*e.g.*, existing collective communication libraries require algorithms that transmit data in discrete, *uniform-sized* chunks)[4], our mechanism will ensure that all transmissions are performed at

[3] This is the same technique as the exact, exponentially sized formulation of [21, A.6], which preemptively duplicates each vertex an exponential number of times; here we apply it lazily, duplicating only the few vertices needed so that the current solution is no longer feasible in the enlarged graph.

[4] Given a routing solution, one possibility to generate an optimal schedule is to use the construction from the proof of Theorem 2; however, this construction does not guarantee that all chunks will be of uniform sizes (in other words, maintaining the optimality of the routing solution may require chunks of non-uniform sizes).

the granularity of uniform-sized chunks. This schedule is the final collective communication algorithm.

At a high level, the mechanism works as follows. It starts with the routing solution, and performs a so-called path decomposition: for each sender-destination pair, it generates a set of paths used for data transmission in the routing solution (that is, paths along which the flow value for the corresponding sender-destination pair is non-zero). The mechanism then considers each sender independently, and aggregates the paths from this sender to all destinations into a collection of trees, each rooted at the sender; importantly, a path (and an edge) may appear in one or more trees. For each tree τ, the mechanism assigns a volume x_τ representing the amount of data that must be transmitted along that tree; indeed x_τ must satisfy several constraints: (1) the sum of x_τ should match the total data size to be transmitted from the sender (that is, all data from the sender must be transmitted); and (2) for each edge, the sum of x_τ across all trees that include that edge must not exceed the fraction of data assigned to that edge (that is, total data size × flow value for that edge) in the routing solution. We show that, given a routing solution, it is trivially possible to generate such set of trees; our mechanism aims to construct a set of trees with a small cardinality.

Given this set of trees, we use an adaptation of weighted fair queueing [17, 18] to construct a schedule over uniform-sized chunks. The goal is to faithfully emulate the routing solution in which each tree τ continuously carries x_τ units of data using only discrete transmissions of chunks of a fixed size C. To do so, we pick an integer parameter K that controls how finely we discretize time, and require each tree to release its chunks at a rate of $x_\tau/(KC)$ per step, so it transmits all of its data over K steps. These chunks then travel down their tree, and at each edge they contend with chunks from other trees that share that edge. To resolve this contention, we use the idea of so-called virtual time from [17, 18]: virtual time of a chunk represents the time when all data corresponding to the chunk would finish transmitting on the edge in the LP(9) solution. Each edge now maintains a queue of waiting chunks; at every step, it forwards chunks with the smallest virtual time. Intuitively, by always serving the chunks that have fallen furthest behind the LP(9) optimal routing solution, no chunk is allowed to lag too far behind. Hence, the resulting uniform-chunk schedule finishes only slightly later than the optimal LP(9) routing solution.

We provide full details in [21, A.7]. Let $x_{\min} = \min_\tau x_\tau$ be the smallest amount of data assigned to any tree in our decomposition; recall that T is an upper bound on the total number of transmissions, across all sender-destination pairs, any piece of data might require to reach from its sender to its destination, that $b_{\min}$ is the minimum bandwidth in the input topology, and that C is the fixed chunk size. Then, we establish that it is possible to find a $K = K^\star$ such that the following theorem holds:

Theorem 4. *Given a routing solution with collective completion time R, OptCCL synthesizes a collective communication algorithm with uniform-sized chunks with collective completion time at most*

$$R\,(1+\gamma)\left(1+\gamma+\frac{CT}{x_{\min}}\right)$$

where, $\gamma = \sqrt{2CT/Rb_{\min}}$.

We prove a better, albeit more non-intuitive, bound in [21, A.7]. The theorem suggests that, for inputs where the slowest link is capable of transmitting data much larger than $C \times T$ during the lifetime of the collective ($R \times b_{\min} \gg C \times T$) and for routing solutions that admit each tree in the decomposition transmitting data much larger than $C \times T$ ($x_{\min} \gg C \times T$), it is possible to convert the optimal routing solution into a collective communication algorithm with uniform-sized chunks in a manner that the optimal collective completion time is inflated by a tiny amount. Note that both C and T are expected to be tiny; thus, the two conditions are likely to hold for most realistic deployments and inputs. Our evaluation in §7 suggests that this is indeed the case: OptCCL synthesizes a schedule that is within 1% of the optimal collective completion time for all evaluated scenarios. Moreover, technology trends are favorable: we find that the inflation in latency reduces with collective sizes (since R and $x_{\min}$ increase roughly proportionally) and with scale (since R increases more rapidly than T).

5 OptCCL generality

Our discussion so far has focused on an input topology with all components capable of performing copy operations and on a single AllGather collective. In this section, we provide details on OptCCL generality: using OptCCL to synthesize optimal algorithms for different hardware functionalities (§5.1), for different collectives (§5.2), and for multiple concurrent collectives (§5.3).

5.1 OptCCL for different hardware functionalities

OptCCL enables specifying the capability for each individual component in terms of the subset of four operations (§2)—transmit, store, copy, reduce. Our LP(9) formulation assumes that all components are capable of transmit, store and copy operations. In particular, it (correctly) enforces that multiple copies of the same data are never sent along a single link; in other words, it enforces the constraint that the bandwidth usage of a link is bounded by the maximum of the f variables. This constraint may not necessarily hold when one or more components are not capable of performing copies.

We now extend our formulation to scenarios where some components may not support the copy operation (*e.g.*, switches that do not support multicast). We start by explaining an approach similar to the one used for synthesizing optimal routing solutions for non-ideal topologies (§4.2)—using duplicate vertices—to handle scenarios when some components are not capable of performing copies; this approach may unnecessarily impact scalability even when the underlying topology is an ideal topology. We then present an approach that achieves the same goal as the general approach, but duplicates vertices only when needed—that is, when an LP-feasible solution is not algorithm-feasible. Components that do not support reduce operations are handled symmetrically by reversing solutions, as we discuss in §5.2.

Formulation via duplicate vertices. Recall that the difficulty in handling components not capable of performing copies is that LP(9) is free to route two flows from the same origin carrying different data out of a non-copy vertex v as though v had simply copied one incoming transmission. For example, in Figure 3 (left), two

Figure 3: An example where LP-feasible solutions do not correctly handle components that do not support copy (left) and the strengthening via vertex duplication to resolve such cases (right).

flows carrying different data leave o, share edge e, and split at the non-copy vertex v; the LP sets usage variables for e assuming that it transmits only one unit of data, but because v cannot copy, the algorithm must send both pieces across e separately. We prevent this by making duplicate copies of v. Let $\overline{V_c} = V \setminus V_c$ be the set of vertices in G that do not allow copies. When we build the time-expanded network, we replace each $v \in \overline{V_c}$ with $|V_c|$ copies, one tied to each copy-capable vertex; all of v's incoming edges enter every copy, but each copy's outgoing edges lead only to its associated vertex in V_c. Flows leaving v toward different destinations are thus forced through different copies, and the LP can no longer merge them for free. Using LP(9) on this new time-expanded network G_*^T leads to a routing solution for AllGather on topologies with components that cannot perform copies. For a full description, see [21, A.8].

Lazy vertex duplication. This approach is practical only if there are few components that cannot perform copies. But in the worst case, this modification will result in almost squaring the number of vertices and edges in our time-expanded network, which makes the size of the formulation grow very quickly. To address this, we will not duplicate vertices up front. We start with LP(9) and add a flow-like constraint on usage variables (w) to LP(9) in an effort to prevent copies at vertices in $\overline{V_c}$:

$$\sum_{e \in \delta^-(v)} w_{e,o} - \sum_{e \in \delta^+(v)} w_{e,o} = 0 \qquad \text{for } o \in V_o^T, v \in \overline{V_c}^T. \tag{10}$$

Unfortunately, there are corner-case examples where adding this constraint by itself does not guarantee that an LP-feasible solution is algorithm-feasible. For such corner-case scenarios, we again use our formulation strengthening technique described in §4.2: solve the formulation, check whether its solution is algorithm-feasible, and if not, strengthen it and re-solve. To strengthen, we duplicate the offending vertex v (in this case, the head of the edge; Figure 3) as described above so that the diverging flows are split across different copies, then re-solve on the enlarged time-expanded network. In the worst case, we might need to create many duplicate vertices; however, lazy duplication improves scalability whenever possible.

5.2 OptCCL for all collectives

OptCCL techniques are general in terms of collectives: OptCCL enables synthesizing optimal collective communication algorithms for any collective that can be described in terms of the four data operations (§2). In this subsection, we describe how to use OptCCL to synthesize optimal algorithms for known collectives: Broadcast, Reduce, ReduceScatter, All-to-All, and AllReduce.

All-to-All. Optimal algorithms for the All-to-All collective require neither copies nor reductions. Thus, OptCCL formulation for this collective is much simpler since usage variables are not required—the amount of data transmitted along an edge can be directly inferred from flow variables. Specifically, an optimal algorithm for the All-to-All collective can be synthesized by changing constraint (8) in LP(9) to

$$\sum_{t=0}^{T-1} \sum_{e \in L} \sum_{o \in V_o^T} f_{\kappa(e,t),o,d} \le Rb(L) \qquad \text{for } L \in \mathcal{L}. \tag{11}$$

Thus, the new linear program is:

$$\min_{f} \quad \sum_{e \in E} \sum_{t=0}^{T-1} c_e \sum_{o \in V_o^T} \sum_{d \in V_d^T} f_{\kappa(e,t),o,d}$$
$$\text{s.t.} \quad (1)-(3), (11);\ f \ge 0. \tag{12}$$

ReduceScatter. ReduceScatter requires reduction operations, and does not require copy operations. Thus, the standard technique for generating solutions for ReduceScatter is to reverse the solutions for AllGather. One assumption that enables this technique is that the set of vertices in G that allow data to be copied are exactly the same set of vertices that allow data to be reduced. Unfortunately, this assumption may not always hold true—while GPUs typically allow both actions, switches may not support copy and reductions, and memory may not support reductions. Fortunately, OptCCL techniques do not require such an assumption: in our formulations, we decouple the ability to store, copy and/or reduce data (via well-defined sets of vertices V_b, V_c and V_r). In particular, we use exactly the same formulation as for AllGather with vertices in V_c replaced with a new set of vertices V_r that are allowed to perform reductions (and reverse direction of all edges in G).

Broadcast and Reduce. Broadcast is the same as AllGather; the only change is in defining the set of origin vertices, with V_o set to be the single vertex that broadcasts the data. Similarly, Reduce is the same as ReduceScatter, where we let V_d be the single vertex that receives the data.

AllReduce. An optimal algorithm for AllReduce may require both copies and reductions of data. Unfortunately, we cannot create an exact formulation for AllReduce by duplicating some of the vertices in our time-expanded network as we could for AllGather or ReduceScatter. This is because the graph created by duplicating switch vertices, say for AllGather, is not symmetric with respect to time—*e.g.*, the in-degree of a duplicate switch vertex is not the same as the out-degree. So duplicating vertices can only support copy operations or reduction operations, but not both simultaneously. We thus introduce a new way to determine the usage on an edge (w) from corresponding flow values (f).

We first establish a structure in optimal AllReduce algorithms: for the objective of collective completion time, there exists an optimal solution where all reductions take place before copies.

Theorem 5. *For the primary objective of collective completion time, the following holds: given an optimal AllReduce algorithm in which any piece of data is copied and later reduced, we can find an optimal algorithm with identical completion time where all pieces of data are first reduced and copied later.*

We emphasize that the theorem does not guarantee that there is an optimal solution with reductions before copies for any arbitrary

objective; it is only for the collective completion time objective. The proof of the theorem, presented in [21, A.9], establishes a stronger result: we can find an algorithm where the reduction happens before copies with identical collective completion time *and* smaller amount of total data transmitted (where we sum the total amount of data sent over all edges).

Theorem 5 suggests that it may be possible to synthesize optimal AllReduce algorithms within our formulation using the standard approach of implementing AllReduce [9]—performing ReduceScatter (that may require reductions but no copies) followed by AllGather (that may require copies but no reductions). It turns out, though, that naïvely using this standard approach may not be optimal due to two reasons. First, this assumes the intermediate state of the solution, namely that equal amounts of the reduction result are present on each GPU. In non-symmetric topologies, this need not be the best intermediate state. Secondly, although we argued that for each piece of data, the reduce-before-copy criterion should be true, waiting for all reductions to finish before beginning any copy operations may lead to suboptimality. Indeed, it may be possible to benefit by parallelizing the beginning of the AllGather algorithm and the end of the ReduceScatter algorithm.

Now we describe how we modify our existing formulations to address both of these concerns. At a high level, we include a set of flow and usage variables, along with their corresponding flow constraints for each of AllGather and ReduceScatter. To address the first concern, we introduce a variable (z) that describes what portion of the reduction result is completed at each vertex, modifying the demands for the flow constraints accordingly. As in LP(9), we will have a constraint bounding the average bandwidth on every edge, considering usage from both constituent collectives. Finally, when generating a chunk-based schedule, we concatenate ReduceScatter and AllGather on a chunk level: as soon as a chunk completes its schedule for ReduceScatter, it is re-enqueued at the beginning of the AllGather scheduling process. In this way, the resulting algorithm performs reductions before copies for each piece of data while parallelizing ReduceScatter and AllGather as much as possible.

See [21, A.9] for the full formulation and proof of its optimality.

5.3 OptCCL for multiple concurrent collectives

OptCCL is able to synthesize optimal algorithms for any arbitrary combination of collectives, each operating on any subset of vertices, as long as the primary objective is to optimize *makespan*: the total time required to complete all collectives.

To handle concurrent collectives, we use the same approach as in the AllReduce collective (§5.2) with the difference being that we now allow the collectives to run concurrently rather than concatenating them. In particular, we create a unified formulation as follows. For each constituent collective, we introduce a set of flow variables, usage variables (if necessary), and corresponding flow constraints. Then, aggregating over variables from all collectives, we write shared objective and bandwidth constraints (similar to (8)) that bound the average bandwidth usage on each edge. From the resulting optimal routing solution, we produce an optimal chunk-based schedule as if there was a single collective (§4.3); thus, all collectives run concurrently, with their combined transmissions on each edge respecting its bandwidth.

6 Exploiting symmetry

The sizes of all our linear programs so far are polynomial in the size of our underlying graph G; however, they still may become intractable as the size of the input grows. For example, for AllGather, the number of variables needed in LP(9) grows as $\Omega(n^3)$, where n is the number of GPUs (where flows originate). In this section, we present techniques to improve computation time for synthesizing our optimal routing solution.

Dantzig-Wolfe decomposition [2] is a standard technique used to speed up the process of solving linear programs with certain block structure (that we discuss below). It turns out that our linear programs have that structure: they can be described as many nearly-independent subproblems, each of which restricts the routing of data from a single origin. Applying Dantzig-Wolfe decomposition to our linear programs, we can iteratively converge to an optimal solution by solving each (much smaller) subproblem, albeit with an iteration-specific objective function. Unfortunately, while Dantzig-Wolfe decomposition improves the synthesis time for optimal routing solutions, it remains slow for large inputs.

Below, we present a new technique that we call Mirrored Dantzig-Wolfe that significantly reduces the time taken to synthesize routing solutions. We start by defining a notion of symmetry in the context of our input. We then establish that, if the input is symmetric, we can always find optimal solutions that are also symmetric; and, by constraining our search space to symmetric solutions, each subproblem in our previous Dantzig-Wolfe decomposition is isomorphic—that is, a solution to one subproblem can be mapped to an equivalently feasible solution to each other subproblem. Using this result, we are able to make equivalent progress solving only one subproblem (as opposed to n subproblems) each iteration. The size of each subproblem is roughly a factor of n^2 smaller than the original problem, so we can solve the original linear program and synthesize optimal routing solutions much faster.

Dantzig-Wolfe decomposition. To give a slightly more detailed overview, Dantzig-Wolfe decomposition is a reformulation of structured linear programs as a master problem and subproblems, where variables in each subproblem are only connected by a few linking constraints. For each subproblem, we use the fact that any polytope can be described as a convex combination of its extreme points. In the master problem, we substitute the variables with their description as a convex combination and use the convex combination coefficients as our new variables. This converts our problem into one with only a few constraints, but (exponentially) many variables. We can solve this converted problem using column generation, where we set the objective for each subproblem to try to find new extreme points that improve the solution.

Our formulations are ideal candidates for Dantzig-Wolfe decomposition. For example, consider the AllGather formulation in LP(9). We observe that the variables for each choice of origin $o \in V_o^T$ are mostly independent; that is, nearly all of the constraints only connect variables with the same choice of o. Only the bandwidth constraints involve variables with different o values, and even then, only the w variables are linked, not the more numerous f variables.

Applying Dantzig-Wolfe decomposition to our AllGather formulation, we have a subproblem (S_o) for each $o \in V_o^T$, and we wish to minimize $\sum_{e \in E} \sum_{t=0}^{T-1} \hat{c}_e w_{\kappa(e,t),o}$, subject to:

$$\sum_{e\in\delta^-(v)} f_{e,o,d} - \sum_{e\in\delta^+(v)} f_{e,o,d} = 0 \qquad \text{for } d \in V_d^T, v \in V^T \setminus \{o,d\};$$
$$\sum_{e\in\delta^-(o)} f_{e,o,d} - \sum_{e\in\delta^+(o)} f_{e,o,d} = 1 \qquad \text{for } d \in V_d^T;$$
$$\sum_{e\in\delta^-(d)} f_{e,o,d} - \sum_{e\in\delta^+(d)} f_{e,o,d} = -1 \qquad \text{for } d \in V_d^T;$$
$$w_{e,o} \geq f_{e,o,d} \qquad \text{for } e \in E^T, d \in V_d^T;$$
$$f, w \geq 0, \tag{13}$$

where the $\hat{c}_e$ are costs dependent on the master problem solution in the current iteration. To state the master problem, let J_o index the set of extreme points of the polyhedron defined by (S_o). And let $\bar{w}^j_{e,o}$ be the $w_{e,o}$ values at the jth extreme point of the polyhedron. Then our full master problem is:

$$\min_{\lambda} \quad \sum_{e\in E}\sum_{t=0}^{T-1} c_e \sum_{o\in V_o^T}\sum_{j\in J_o} \lambda_o^j \bar{w}^j_{\kappa(e,t),o}$$
$$\text{s.t.} \quad \sum_{t=0}^{T-1}\sum_{e\in L}\sum_{o\in V_o^T}\sum_{j\in J_o} \lambda_o^j \bar{w}^j_{\kappa(e,t),o} \leq Rb(L) \text{ for } L \in \mathcal{L};$$
$$\sum_{j\in J_o} \lambda_o^j = 1 \qquad \text{for } o \in V_o^T;$$
$$\lambda_o^j \geq 0 \qquad \text{for } o \in V_o^T, j \in J_o. \tag{14}$$

For more details on how to use this decomposition to generate the final solution, please see [2]. Indeed, this technique easily extends to all our formulations.

Mirrored Dantzig-Wolfe. [5] Even with Dantzig-Wolfe decomposition with our formulations, it may take a prohibitively long time to synthesize optimal routing solutions. However, we observe that in many cases our Dantzig-Wolfe decomposition is made up of subproblems that seem remarkably similar, especially when our inputs are symmetric in a sense that we describe below. We next outline how we can leverage this symmetry to greatly improve the time taken to synthesize the optimal routing solutions.

We define symmetry through the concept of indistinguishability. Two GPUs are *indistinguishable* if, after removing all component and link identifiers from our input, the corresponding GPUs are isomorphic; we provide a precise definition in [21, A.10], but intuitively, isomorphism here is defined with respect to both the input topology and the collective structure. We further introduce a *Symmetry Partition*, $\mathcal{P}$; the sets in $\mathcal{P}$ are disjoint subsets of vertices in V_o (each referred to as an "indistinguishability class") such that for each $P \in \mathcal{P}$ and for any $g_1, g_2 \in P$, g_1 and g_2 are indistinguishable. For each indistinguishability class $P \in \mathcal{P}$, we define a mapping function π_P that maps each component (and each link) in the input topology to one or more components (and edges) in the input topology. Given each of the π_P, we say that a solution to our formulation is symmetric if usage and flow variables on any edge e are the same as the usage and flow variables on all edges to which any of the π_P maps e. Given these definitions, we prove the following theorem:

Theorem 6. *Given a solution to LP*(9)*, Symmetry Partition $\mathcal{P}$, and an associated function π_P for each $P \in \mathcal{P}$, we can find a symmetric solution to LP*(9) *of the same completion time and objective value.*

The proof is presented in [21, A.10]. The existence of a symmetric optimal solution allows us to reduce the size of our formulation by eliminating (now redundant) variables and constraints, while maintaining optimality. Specifically, when integrated with Dantzig-Wolfe decomposition, our symmetry definition along with Theorem 6 allows us to guarantee that for $o_1, o_2 \in V_o^T$, if $\mu(o_1)$ and $\mu(o_2)$ are in the same indistinguishability class, then subproblems S_{o_1} and S_{o_2} are isomorphic; that is, they are identical up to the relabeling of variables and remain identical after each iteration. Therefore, instead of solving $|V_o|$ subproblems each iteration, we only need to solve one subproblem per indistinguishability class ($|\mathcal{P}|$ subproblems). And in the special case where all $v \in V_o$ are indistinguishable, we need only solve one subproblem per iteration. All these scalability benefits can be achieved while maintaining optimality.

7 Evaluation

We now evaluate OptCCL and compare it with state-of-the-art systems—TE-CCL, SyCCL and TACCL—over two topologies: A100 with 8 GPUs per host [7] and DOE with 4 GPUs per host [16]; in both cases, we use a multi-rail inter-host network topology [24].

Setup. We evaluate performance in terms of algorithmic bandwidth, defined as the ratio of collective data size to collective completion time (higher is better). We use a server with 32-core Intel Xeon Gold 6234 CPU and 314GB memory to measure the synthesis time.

We evaluate TE-CCL, SyCCL, and TACCL using their publicly available code. We were unable to run SyCCL on scale-up (single host) topologies, as their implementation appears to be designed exclusively for multi-host settings with a limited set of link types. TACCL has bugs in its All-to-All implementation (as is noted in their GitHub issues). Each of these systems requires, as input, a fixed number of chunks; typically, the synthesis time worsens and the performance improves with number of chunks. We present results for the largest number of chunks for which each system was able to synthesize an algorithm within a three-hour limit.

We set up OptCCL with existing hardware features. That is, GPUs and CPUs have the ability to perform each of the above four operations. Memory can store, transmit and copy, but cannot perform reductions. Switches can store and transmit but cannot perform copies and reductions. For measuring synthesis time, we generate one representative schedule per symmetry class which contains all the same information as a full schedule.

7.1 OptCCL for single collectives

Figure 4 and Figure 5 show results for the two topologies, both with NVLinks, for AllGather and All-to-All collectives, respectively. Across both topologies and collectives, TE-CCL was unable to synthesize algorithms within three hours for more than 16–32 GPUs;

[5] Our Mirrored Dantzig-Wolfe technique may be of general interest to the optimization community working on multicommodity flow problems. In particular, existing decomposition-based techniques in such problems reduce the search space by creating one decomposition for each commodity, and creating a subproblem for each decomposition. We show that such techniques can be improved by creating a decomposition for each symmetry class instead; since each symmetry class contains multiple commodities, this drastically reduces the number of subproblems.

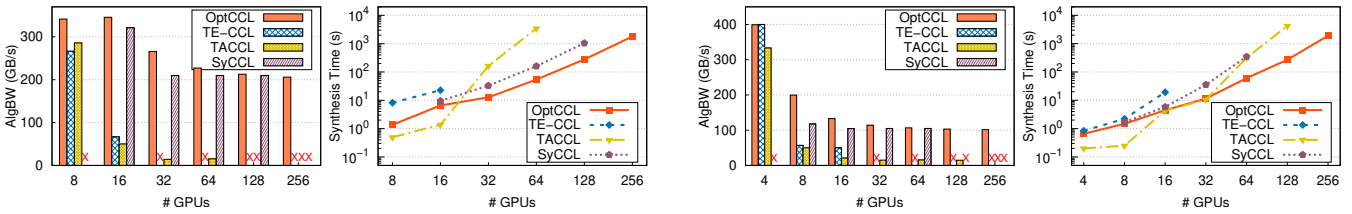

Figure 4: Performance in terms of algorithmic bandwidth and synthesis time for the A100 (left) and the DOE topologies (right) for the AllGather collective. OptCCL synthesizes algorithms that have 1–5×, 1–1.6×, 1.2–18× better performance than TE-CCL, SyCCL and TACCL, respectively. OptCCL also requires 3–60× lower synthesis time than SyCCL and TACCL on large topologies.

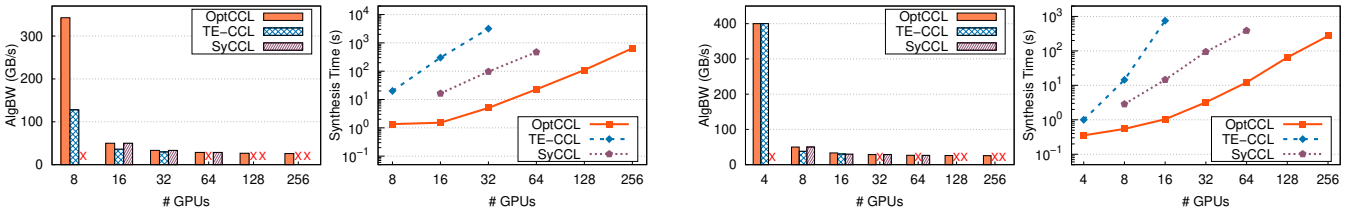

Figure 5: Performance in terms of algorithmic bandwidth and synthesis time for the A100 (left) and the DOE topologies (right) for the All-to-All collective. OptCCL synthesizes algorithms that have 1–1.3× and 1–1.3× better performance than TE-CCL and SyCCL, respectively. OptCCL also requires 2–500× lower synthesis time than TE-CCL and SyCCL.

the algorithms synthesized by TE-CCL are also far from optimal in most cases because it does not scale to inputs with a large number of chunks (performance improves with larger number of chunks). SyCCL significantly improves upon TE-CCL, both in terms of performance and synthesis time; however, it remains limited: it is unable to synthesize algorithms for more than 64–128 GPUs and often produces suboptimal algorithms. TACCL has scalability similar to SyCCL, but produces algorithms that have significantly worse performance than optimal even for moderate number of GPUs.

OptCCL consistently synthesizes better algorithms—1–5× better than TE-CCL, 1–1.6× better than SyCCL, and 1.2–18× better than TACCL. Moreover, beyond a small number of GPUs, OptCCL also significantly improves upon synthesis time; for example, for 64 or more GPUs, OptCCL synthesizes algorithms 3–32× faster than SyCCL and 5–60× faster than TACCL. Much of OptCCL's benefit in terms of synthesis time is due to the Mirrored Dantzig-Wolfe technique introduced in §6; see [21, A.11].

7.2 OptCCL for multiple concurrent collectives

For multiple collectives, we use an 8-host (64-GPU) A100 topology organized around 2 pods of 32 GPUs each. The setup uses 8 All-to-All collectives (one within each host), 16 ReduceScatter collectives (8 per pod, one for each rank within the pod), and 32 point-to-point communications (one for each pair of GPUs with the same rank across pods); the total data size for ReduceScatter and point-to-point communications is 2× the data size for the All-to-All collective. We evaluate the total time required to complete all the collectives.

Table 2 compares performance of various methods for synthesizing algorithms for multiple collectives. "Obvious" refers to first generating algorithms for individual collectives, and then executing them sequentially; this is the only approach we know for existing techniques to handle multiple concurrent collectives. "Overlaid" refers to solving each collective separately, then optimally scaling them so that the collectives run concurrently. And, "All-in-one" captures the result from optimizing for all collectives in a single formulation (§5.3). For each approach, we use OptCCL.

Table 2: Performance for multiple concurrent collectives on 8 host A100 topology (with 64 GPUs), normalized by corresponding all-in-one performance. Discussion in §7.2.

	With NVLinks	Without NVLinks
Obvious	6.25×	4.87×
Overlaid	1×	1.31×
All-in-one	1×	1×

With NVLinks, just overlaying the results already gains a 6× improvement in performance (thus, OptCCL performance for multiple concurrent collectives will be at least this much better than existing techniques), and solving the unified linear program from §5.3 gives no extra benefit. In this particular topology, we believe the overlaid solution is optimal because by optimizing the individual collectives for our secondary objective, we reduce contention in important links, which allows the overlaid solution to be better.

Without NVLinks, overlaying the results still yields more than 3× improvement; in addition, our all-in-one formulation gains an additional 1.3× improvement in performance.

8 Related work

We have already evaluated OptCCL against existing state-of-the-art techniques for synthesizing collective communication algorithms. We briefly discuss them, and other related works below.

Heuristics. The most widely-deployed systems for collective communication today [9, 10] use algorithms that are not optimized for the underlying host hardware and network topology, and are known to perform far from optimal. Several recent works [14, 20, 23, 25] present techniques to synthesize heuristics that empirically perform better than widely-deployed collective communication libraries;

however, as shown in our evaluation, even the state-of-the-art of these [20] can perform far from optimal and have poor scalability.

Optimality-scalability tussle. SCCL [3] uses an SMT solver to generate optimal algorithms. While the original SCCL paper did not focus on generality, it is not too hard to extend SCCL to model various hardware functionalities and concurrent collectives—after all, it uses an SMT solver that enables writing arbitrarily complex constraints. However, SMT solvers, without specialized optimizations, are notoriously slow. Unsurprisingly, many prior studies have already established that SCCL is unable to generate algorithms beyond a small number of GPUs.

The works closest to us are TE-CCL [15] and SyCCL [5]. The core similarity between TE-CCL and OptCCL is that both of them build upon the broad class of multicommodity-flow-based formulation. However, beyond that, they are fundamentally different. TE-CCL captures collective structure within the flow constraints, enforcing integer constraints on flow variables which eventually limits their scalability. TE-CCL, admitting its scalability limitations, also presents heuristics for inputs with more than a few GPUs (in which case, it loses the benefits of synthesizing optimal algorithms).

SyCCL [5] introduces new techniques to improve TE-CCL's scalability by exploiting symmetry in host hardware and network topologies. SyCCL defines symmetry with respect to the hierarchical structure of commonly used network topologies, and classifies GPUs into isomorphic groups with respect to sets of identical links. SyCCL enforces symmetry *within* the route of each piece of data: the pattern used to route data within one GPU group must be the same in every isomorphic group. Unfortunately, as briefly mentioned in the SyCCL paper (and as we formally establish in the technical report [21, A.12]), enforcing the symmetry in such a manner can no longer guarantee optimality. Thus, SyCCL helps TE-CCL improve scalability but only by giving up on optimality.

OptCCL uses a fundamentally new formulation that enables it to break the optimality-scalability tussle; indeed, OptCCL always synthesizes optimal algorithms while maintaining generality and achieving improved scalability. OptCCL also uses a different indistinguishability-based definition of symmetry: via isomorphisms between labelings of the entire graph and using this symmetry to simplify the formulation while preserving optimality guarantees. Indeed, OptCCL techniques may be useful to improve TE-CCL's generality and scalability.

Scalable and optimal; but, give up on generality. A work concurrent to us—ForestColl [27]—introduces intriguing combinatorial techniques that enable synthesis of optimal collective communication algorithms for a subset of the collectives. By focusing on a certain set of collectives that can be modeled as data transmission over a collection of spanning trees, ForestColl is able to achieve scalability (in many cases, even better than OptCCL); see [21, A.13]. Unfortunately, by exclusively relying on spanning trees, ForestColl gives up significantly in terms of generality. First, ForestColl is unable to synthesize collective communication algorithms for some of the collectives (*e.g.*, All-to-All): any spanning tree packing based technique only works on collectives that can be represented as spanning trees; All-to-All, and point-to-point communication in general cannot be represented in this way. Second, as evaluated in the technical report [21, A.13], ForestColl is unable to model shared bandwidth constraints in memory components without giving up on scalability and/or optimality. Finally, ForestColl is unable to generate algorithms when collectives do not span the entire set of GPUs in the topology (*e.g.*, in concurrent collective case)—this is again quite fundamental to the techniques used in ForestColl (packing of spanning trees): if some collective does not span all the GPUs, then their approach will require packing of so-called (directed) Steiner trees which is known to be a computationally hard problem, and does not even admit a good approximation [6].

9 Limitations and future directions

In this section, we outline several limitations of OptCCL and discuss interesting future directions that remain open.

Improving OptCCL generality. As discussed in §5, OptCCL formulation is general—it supports existing hardware functionalities, a broad definition of collectives, and supports optimizing completion time for multiple concurrent collectives. We outline several interesting directions to further improve OptCCL's generality. First, while we provide OptCCL implementation for all existing collectives, extending OptCCL to new collectives will require carefully defining usage variables for that collective. Second, our proofs and evaluation primarily focus on large message sizes where transmission delays dominate; it would be interesting to extend our proofs and evaluation to the case of small messages where propagation delays may dominate (*e.g.*, under the so-called α–β model). We believe this is fairly straightforward (using techniques similar to [15]): when all transmissions use fixed-size chunks, the time to send a chunk across a link is a fixed quantity; the propagation delay can thus be folded into an effective link bandwidth, after which the rest of OptCCL's techniques apply unchanged. Finally, the problem of synthesizing collective communication algorithms for multiple concurrent collectives that are optimal with respect to objectives beyond makespan (*e.g.*, flow-time or weighted completion time as in the coflow literature [1]) remains an intriguing open problem.

Real-world implications. OptCCL evaluation uses simulations. The natural next step here is to integrate OptCCL with existing collective communication libraries [9, 10], and validate OptCCL's performance benefits over real testbeds and over end-to-end ML training and inference applications.

10 Conclusion

OptCCL is a technique that, for any given host and network hardware and topology, synthesizes optimal collective communication algorithms. OptCCL is general: it enables synthesizing optimal algorithms for all existing collectives, for a large class of existing and future hardware, and for multiple concurrent collectives sharing host and network resources. Evaluation of OptCCL across multiple host and network hardware and topologies suggests that OptCCL is also scalable: it consistently synthesizes optimal collective communication algorithms for hundreds of GPUs within tens of minutes.

Acknowledgments

We would like to thank SIGCOMM reviewers for their insightful comments. This research was in part supported by NSF grant CNS-1704742. This work does not raise any ethical issues.

References

[1] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. 2018. Sincronia: Near-optimal network design for coflows. In *SIGCOMM*.

[2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. 1993. *Network flows - theory, algorithms and applications*. Prentice Hall, Upper Saddle River, NJ.

[3] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2021. Synthesizing optimal collective algorithms. In *PPoPP*.

[4] Jiamin Cao, Yu Guan, Kun Qian, Jiaqi Gao, Wencong Xiao, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. 2024. Crux: GPU-efficient communication scheduling for deep learning training. In *SIGCOMM*.

[5] Jiamin Cao, Shangfeng Shi, Jiaqi Gao, Weisen Liu, Yifan Yang, Yichi Xu, Zhilong Zheng, Yu Guan, Kun Qian, Ying Liu, Mingwei Xu, Tianshu Wang, Ning Wang, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. 2025. SyCCL: Exploiting symmetry for efficient collective communication scheduling. In *SIGCOMM*.

[6] Joseph Cheriyan and Mohammad R Salavatipour. 2006. Hardness and approximation results for packing Steiner trees. *Algorithmica* 45, 1 (2006), 21–43.

[7] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. Nvidia A100 tensor core GPU: Performance and innovation. *IEEE Micro* (2021).

[8] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research* (2023).

[9] NVIDIA collective communications library (NCCL). 2026. https://developer.nvidia.com/nccl.

[10] ROCm communication collectives library (RCCL). 2025. https://github.com/ROCm/rccl.

[11] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. 2024. RDMA over Ethernet for distributed training at Meta scale. In *SIGCOMM*.

[12] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *ATC*.

[13] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *NSDI*.

[14] Heehoon Kim, Junyeol Ryu, and Jaejin Lee. 2024. TCCL: Discovering better communication paths for PCIe GPU clusters. In *ASPLOS*.

[15] Xuting Liu, Behnaz Arzani, Siva Kesava Reddy Kakarla, Liangyu Zhao, Vincent Liu, Miguel Castro, Srikanth Kandula, and Luke Marshall. 2024. Rethinking machine learning collective communication as a multi-commodity flow problem. In *SIGCOMM*.

[16] NERSC. 2026. Perlmutter architecture. https://docs.nersc.gov/systems/perlmutter/architecture/.

[17] A.K. Parekh and R.G. Gallager. 1993. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking* (1993).

[18] A.K. Parekh and R.G. Gallager. 1994. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Transactions on Networking* (1994).

[19] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling distributed machine learning with in-network aggregation. In *NSDI*.

[20] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. 2023. TACCL: Guiding collective algorithm synthesis using communication sketches. In *NSDI*.

[21] Richard Shapley, Rachit Agarwal, and David Shmoys. 2026. *OptCCL: Scalable synthesis of optimal collective communication algorithms*. Technical Report. https://github.com/ml-collectives/optccl

[22] Midhul Vuppalapati, Saksham Agarwal, Henry Schuh, Baris Kasikci, Arvind Krishnamurthy, and Rachit Agarwal. 2024. Understanding the host network. In *SIGCOMM*.

[23] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. 2020. Blink: Fast and generic collectives for distributed ml. *Proceedings of Machine Learning and Systems* (2020).

[24] Weiyang Wang, Manya Ghobadi, Kayvon Shakeri, Ying Zhang, and Naader Hasani. 2024. Rail-only: A low-cost high-performance network for training LLMs with trillion parameters . In *HOTI*.

[25] William Won, Midhilesh Elavazhagan, Sudarshan Srinivasan, Swati Gupta, and Tushar Krishna. 2024. TACOS: Topology-aware collective algorithm synthesizer for distributed machine learning . In *MICRO*.

[26] Yongji Wu, Yechen Xu, Jingrong Chen, Zhaodong Wang, Ying Zhang, Matthew Lentz, and Danyang Zhuo. 2024. MCCS: A service-based approach to collective communication for multi-tenant cloud. In *SIGCOMM*.

[27] Liangyu Zhao, Saeed Maleki, Yuanhong Wang, Zezhou Wang, Ziyue Yang, Hossein Pourreza, and Arvind Krishnamurthy. 2026. ForestColl: Throughput-optimal collective communications on heterogeneous network fabrics. In *NSDI*.