ALGORITHMIC TECHNIQUES FOR OPTIMAL COLLECTIVE COMMUNICATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Richard Shapley

August 2025

ALGORITHMIC TECHNIQUES FOR OPTIMAL COLLECTIVE COMMUNICATION

Richard Shapley, Ph.D.

Cornell University 2025

Improving the performance of collective communication—algorithms for coordinated data movement in distributed computing—becomes increasingly important as the use of large-scale machine learning clusters continues to grow. In this work, we introduce techniques for synthesizing optimal algorithms for collective communication for any given host and network topology. We provide a novel multicommodity flow-based approach that allows us to find provably optimal algorithms in terms of both latency and bandwidth utilization. Critical to our techniques are two important innovations: a conversion method from fluid solutions to discrete step-based solutions, and Mirrored Dantzig-Wolfe, which exploits the high degree of symmetry often present in communication networks. We complement our theoretical contributions with experimental results, demonstrating the practical efficacy of the constructed algorithms.

BIOGRAPHICAL SKETCH

Ricky Shapley was born and raised in Tucson, Arizona. He earned his undergraduate degree in Math and Computer Science from Harvey Mudd College in 2020, working with Arthur Benjamin and Weiqing Gu. In the Fall of 2020, he began his Ph.D. at Cornell ORIE.

ACKNOWLEDGEMENTS

I am grateful to the many people who have supported me throughout my five years at Cornell.

First, I want to thank my advisor, David Shmoys, who has so patiently guided me during my graduate studies, always advocating for me and giving advice about both research and life. I can only hope to one day emulate his brilliance and productivity.

This work would not have been possible without my collaborators. I am particularly grateful to Rachit Agarwal, who brought the subject of this dissertation to my attention and has been an exceptional collaborator throughout this project. His insights, dedication, and the time and resources he has provided have been invaluable. I also want to thank Shouxu, who collaborated with me on this project for a time and contributed meaningfully to its development.

I am also grateful to Oktay and Ishan, who collaborated with me on another project that, while not included in this dissertation, was an important part of my graduate research experience.

I also want to thank my committee members, David Williamson and Sid Banerjee, for their thoughtful discussions not just about this work but more generally about research in this field.

The discrete optimization community in our department has been a source of both intellectual stimulation and friendship. I am particularly grateful to Ishan, Varun, and Priya. Special thanks go to Ishan for starting our reading group with me, and to Priya for helping me keep it running. Thanks also go to the regular at-

tendees of our reading group, especially my committee members who generously donated their time to help us all gain greater insights, as well as frequent presenters Varun, Billy, Claire, Hannane, and Mona, who made our lives as organizers much easier.

Teaching ENGRI 1101 has been an important part of my graduate experience, and I am grateful to those who helped me in this endeavor. Thanks again go to my advisor David, who developed the course and provided a great model to follow in terms of teaching. I want to thank Alyf and Priya, who both co-instructed with me and made the teaching experience both more manageable and more rewarding.

I'm also grateful to the wonderful ORIE community and all the other friends who have made my time here so meaningful. While there are far too many people to name individually, I especially want to acknowledge Tyler, Eliezer, George, Amber, Tao, and Sam, who haven't been mentioned yet but have been such important parts of my journey.

Lastly, I want to thank my parents for their consistent support throughout my academic pursuits, even as it has taken me far away from them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Recent advances in machine learning have led to a rapid increase in both model size and the scale of computational systems used for training, and this trend is expected to continue as our desire for more accurate and complex models continues [8]. While this growth has yielded incredible results in various areas including computer vision and generative AI, it has required fundamental changes to how models must be trained.

Since the number of parameters used in such models is growing at an unbelievable rate (Llama 2 [14] has 70 billion parameters, whereas Llama 3 [7] has 405 billion parameters), the models need to be distributed across an increasing number of GPUs in increasingly large distributed clusters. Notably, Llama 3 was trained on 16,000 GPUs in Meta's cluster [6], GPT-3 was trained on 12,288 GPUs in ByteDance's cluster [8], and PaLM was trained on 6,144 TPUs in Google's cluster [5]. As these systems grow to include thousands of GPUs, the challenge of coordinating data movement across diverse network topologies and hardware architectures intensifies.

In fact, coordinating communication is a critical bottleneck that is becoming increasingly important in comparison to computational power. Recent works have shown that communication operations constitute a significant portion of execution [12, 19, 3, 6]. And since improvements in computational power are outpacing

those in communications, the importance of communicating efficiently will continue to rise.

Communication operations in these distributed systems usually involves simple, well-defined operations called collective communication primitives. The challenge of determining how to transmit data in order to satisfy these communication requirements gives rise to a new class of routing optimization problems that must adapt to arbitrary system topologies while optimizing for multiple objectives such as minimizing latency and bandwidth utilization. In this thesis, we develop a comprehensive approach to this optimization challenge that achieves provable optimality across arbitrary network architectures while maintaining computational tractability at scale.

Previous approaches to collective communication optimization such as TACCL [13] and TE-CCL [10] have primarily relied on integer programming formulations. However, reliance on integer programming fundamentally limits their scalability, as these formulations become computationally intractable for large-scale systems. Consequently, these methods resort to heuristics to produce algorithms as the number of GPUs grows, sacrificing optimality guarantees for computational tractability. There are also approaches that use spanning tree packing like ForestColl [20], and although this method provides some forms of optimality guarantees, it cannot capture complex bandwidth limitations or accurately model the behavior of memory. Other approaches either use unreliable SMT solvers ([2]) or are entirely heuristic-based ([9, 18]).

The collective communication routing problem differs fundamentally from standard multicommodity flow formulations because flows from the same source (corresponding to transmissions from the same GPU) are not independent: they can be satisfied by the same data transmission. Despite this difference, as in other multicommodity flow problems, Dantzig-Wolfe decomposition emerges as a crucial algorithmic tool for achieving computational efficiency by exploiting the problem's natural block structure.

In this work, we introduce a method of synthesizing collective communication algorithms that provides provable optimality guarantees for both latency and general secondary objectives across arbitrary system topologies. We emphasize two key innovations: first, we develop a formulation that enables converting from a fluid, fully fractional solution space to a discrete step-based solution, allowing us to leverage the computational advantages of solving the fluid version via linear programming; second, we introduce a novel technique which we call Mirrored Dantzig-Wolfe that exploits symmetries in the topologies of our systems.

Beyond optimizing existing systems, our techniques provide a framework for evaluating new network topologies and can guide the design of future distributed computing architectures.

Organization

This dissertation is organized into several chapters, each building upon the previous work to develop a comprehensive framework for collective communication optimization. We will summarize each chapter below.

In Chapter 2, we provide essential context for understanding the collective communication optimization problem. We introduce various common collective communication algorithms that are in use for ML training, establish notation for representing network topologies, and discuss different communication models ranging from discrete chunk-based approaches to continuous fluid models. We also survey existing approaches and identify their specific limitations through concrete examples.

In Chapter 3, we develop the mathematical foundations of our optimization framework. Beginning with simplified scenarios, we systematically build exact linear programming formulations for collective communication problems. We start with the AllGather primitive under restrictive assumptions, then progressively generalize to handle arbitrary topologies, divisible data, and heterogeneous component capabilities. This chapter establishes the theoretical correctness of our approach and demonstrates how relaxations can be systematically strengthened to achieve optimality guarantees.

In Chapter 4, we addresses the challenge of solving these difficult discrete-time formulations by first solving a time-oblivious version of the problem, then con-

verting solutions into discrete, implementable algorithms. Our techniques separating spatial routing decisions from temporal scheduling decisions, enabling optimization over arbitrarily fine time granularities without solving prohibitively large linear programs. Special attention is given to the AllReduce collective, which presents unique challenges due to the need to coordinate both reduction and copying phases.

In Chapter 5, we introduce our novel Mirrored Dantzig-Wolfe decomposition technique. We formally define symmetry in the context of communication networks and prove that symmetric problem instances always admit symmetric optimal solutions. This insight enables us to solve only one representative subproblem per symmetry class rather than solving independent subproblems for each processing unit, resulting in dramatic computational speedups.

Finally, in Chapter 6, we present experimental validation of our techniques. We compare against state-of-the-art systems across multiple metrics including computational efficiency, solution quality, and scalability. The experiments demonstrate both the practical effectiveness of our approach and the necessity of our algorithmic innovations for achieving tractable solutions at scale.

CHAPTER 2

BACKGROUND

2.1 Collectives

There are several common collective operations, also referred to as primitives, that we aim to find algorithms for. While these collectives were originally created in the high performance computing community to describe messaging, we will describe them in terms of communication between GPUs. Each of these primitives involves moving data between GPUs in various ways. Below, we explore some of the more common collectives. We use the convention that the collectives are operating on a system with n GPUs, and the initial data size is 1 unit (where this unit may be whatever data size is convenient).

- **Broadcast**: In the Broadcast collective, we distribute the data from one GPU (which we will refer to as the root) to all other GPUs.
- **AllGather**: In AllGather, we collect data from each GPU and distribute that gathered data to all other GPUs, so that at the end of the collective operation, every GPU has a copy of all other GPUs' data. This is functionally equivalent to a Broadcast from every GPU.
- **Reduce**: The Reduce primitive performs reductions across the data from all the GPUs, and writes the result to the root GPU. We assume commutativity of reduction operations, that is, the result of the reduction operation is the

same regardless of the order in which it processes the input data. In some ways, we can think of this operation as the reverse of Broacast.

- **ReduceScatter**: ReduceScatter combines data from all GPUs by performing a reduction, then distributes unique chunks of the result to each GPU. Each GPU receives $\frac{1}{n}$ of the result. This is functionally equivalent to performing Reduce n times on $\frac{1}{n}$ of the data, where each one sends the result to a different GPU.
- **AllReduce**: In AllReduce, we perform a reduction across data from all GPUs and then distribute the complete result to every GPU, so each GPU ends up with the same reduced data. This is functionally equivalent to performing Reduce on all of the data followed by a Broadcast.
- **All-to-all**: All-to-all sends distinct data between all pairs of GPUs, including from each GPU to itself, so that each GPU receives $1/n$ of the data from each GPU.

2.2 Representation of topologies

The architecture (or topology) of a system consists of a set of components (including GPUs, memory, and switches), a set of links connecting components, and bandwidth limits on links or components. Components can store data, copy data, perform reductions, or combinations thereof.

For ease of exposition, we will typically assume that we have GPUs that can do all of the above, memory components that can store and copy data but not perform reductions, and switches that can do none of these actions. However, we emphasize that our framework can accommodate components that support any set of these actions.

We model any topology as a directed graph G, with vertices corresponding to components and edges corresponding to links. To capture bandwidth limits, we may specify a bandwidth for any set of edges. More specifically, for any set of edges L with a cumulative bandwidth limit, 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. For example, to give a single edge e a bandwidth limit b_e, we will define $b(\{e\}) = b_e$. Note that this method of defining bandwidths also allows us to place bandwidth limits on vertices by letting L be the set of out-edges from the vertex. We let $\mathcal{L}$ be the collection of all edge sets that have a bandwidth bound.

We also introduce notation to describe sets of vertices. Since components may have many possible combinations of behaviors, we address each separately. We let V_b be the set of vertices corresponding to components that can store data, V_c be the set of vertices corresponding to components that allow copying data, V_r be the set of vertices corresponding to components that allow reductions, and V_o and V_d be the origin and destination vertices determined by the collective operation. To help with our future usage of the graph G, we will also introduce self-loops for every vertex $v \in V_b$ with infinite capacity. This represents an assumption that we

are able to store an infinite amount of data at such components, or rather that we should not be worried about memory capacity as a limitation on our algorithms.

2.3 Communication models

There are several ways we may manipulate data as it is routed through a topology: Data can be *stored* at a component (GPUs and memory). It can be *reduced* according to a reduction operation, where two (or more) pieces of data, each of size M, are combined into a single piece of data of size M. Data can also be *copied*; that is, it can be transmitted along multiple outgoing links so that the same data exists in multiple places in the system. (We also say that data is copied if it is transmitted but also remains at the sender so that again, the data now exists in multiple places.)

We consider three related models of communication that differ in how much we are allowed to divide our input into smaller pieces. We describe them below and their relationships to a common routing technique known as pipelining.

- **CM1**: In this first model, data is indivisible. This captures models where pipelining is not allowed: When routing data of size M via an intermediate component, we must wait for the entire M data to be received at the intermediate component before any of it can be re-transmitted to the destination.
- **CM2**: We extend CM1 by now declaring that data is divisible into some fixed

number (K) of equally sized indivisible chunks. Among other properties, this now enables pipelining: When data is routed via an intermediate component, we can begin re-transmitting data from the intermediate component as soon as the first chunk has been received. As the number of chunks increases, pipelining can lead to unbounded improvements in latency compared to the CM1.

- **CM3**: We now take the divisibility of data to its extreme. In this setting, also sometimes called the fluid model of communication, data is infinitely divisible. This model captures CM2 in the limit, that is, as $K \to \infty$. When we route data via an intermediate component, we can transmit data continuously along both links without ever waiting. This allows us to achieve the asymptotic latency that occurs when $K \to \infty$ in the previous model.

In each of these models, we fix a discrete step size s, which is a length of time, so we can consider what data we send and where we send it within the duration of a single step. Then, for each set of links with a bandwidth limit $b(L)$, we can compute strict capacities for any period of time t via the product $t \cdot b(L)$.

An algorithm consists of a sequence of steps, where every step describes all the data transmissions that occur in that step. The description of a data transmission includes what data is sent, the component sending the data, the component receiving the data, and the contents of the data transmitted. For CM1, the contents can be described entirely by the GPU(s) where the data originated. For CM2 and CM3, we also need to indicate which piece of data is being transmitted. An al-

gorithm is feasible if it performs the desired primitive, while never sending more data than the bandwidth capacity along any link (or set of links).

2.4 Objective

The primary objective we worry about is an algorithm's latency. We define the *latency* of an algorithm to be the time elapsed from the start of the algorithm to the earliest time when the collective is completed. For an algorithm on a particular step size s, we define the latency of the algorithm as the number of steps required to complete the collective multiplied by the step size. And the optimal latency under each communication model is the minimum latency over all algorithms at all choices of step size. Note that under CM3, the step size s that achieves minimum latency may become infinitesimally small.

We must also remark that the latency of algorithms with a fixed step size may be an overestimate. More specifically, our communication models and definition of latency give a pessimistic bound on the cost of any algorithm because of the constant step size. It is possible that in some algorithm, all transmissions at a particular time are on high-bandwidth links. We assume that this step requires the entire step to finish, where in reality, all transmissions could finish in less time than the step size, and so the algorithm could do better by shortening the step.

At this point, it is worth considering why we focus on step-based algorithms,

when there may be other methods that eliminate some of the drawbacks we've indicated. Indeed, in an idealized setting, we could get around the problems caused by infinitesimal step sizes, or wasted idle time by thinking of models that capture a steady state. (In fact, we will do something similar to this as we expand our techniques). However, the purpose of our algorithms is to be actually implementable for use on real systems, and the method by which algorithms are written as instructions typically requires a step-based paradigm.

There is also reason to consider other objectives other than latency. For any collective, it is likely that there are many algorithms with equivalent latency, especially when only a small portion of the algorithm determines the latency and the rest of the routing decisions are nearly unconstrained. Therefore, we will also optimize for a secondary objective among all algorithms with optimal latency. Our techniques generally allow for any objective that is a weighted sum of the amount of data transmitted along all links in the underlying topology. But for the purposes of exposition and evaluation, we will suppose that our secondary objective is to minimize the total data uploaded to and uploaded from GPUs, which is a special case of the general objective just stated. Indeed, we have reason to believe that this helps improve outcomes in the setting where we need to run multiple collectives simultaneously.

2.5 Existing methods

In this section, we will attempt to summarize the techniques from previous works on the topic. These techniques fall into several broad non-disjoint categories. We will describe these categories and give some examples of methods that fall into each.

One-size-fits-all

The first category of methods is a one-size-fits-all style strategy, where the algorithm designer ignores most of the structure of the topology, and imposes some fixed logical topology, usually with a very simple structure. Some examples of this strategy are NCCL [11] and TCCL [9]. These methods impose either a ring structure or a tree structure on any topology, and then always route data according to this predefined structure. In particular, this greatly simplifies communication because the system does not need to make any decision for how to route data through the network, because there is already a predefined path the data will follow, regardless of the collective. However, this means that the data routing strategies are far from optimal, and it is generally possible to achieve much better results by letting data routing patterns be more flexible. Even though its techniques may not be so efficient in terms of latency, NCCL is widely used in practice.

Step-based algorithm search

Another category of methods are those that try to generate algorithms by dividing time into steps and deciding what data to transmit at each step. The earliest of these methods, MSCCL [2] used a SMT (Satisfiability modulo theories) solver to determine if it was possible to route data to its destination in a given amount of time. Combined with binary search, it is possible to find algorithms that are optimal in terms of latency (and within the assumptions of their model). However, SMT solvers, as generic constraint solvers, are famously slow, and cannot guarantee a solution in any reasonable period of time. Especially since the encoding of the constraints on the generated algorithm grows quickly with the size of the input, this method is not practical for any large number of GPUs.

Some innovations on this basic idea used integer programming methods to try to achieve the same sort of result. Two of these are TACCL and TECCL ([13, 10]). These are among the most widely cited methods that go beyond a one-size-fits-all method. As they rely on integer program solvers (like Gurobi), they can make use of the decades of research spent on increasing the speed of finding solutions. However, even so, it is generally too much to ask for these methods to produce exact solutions. TACCL gets around this by not actually trying to find an optimal solution, instead separating the problem into smaller, slightly more tractable problems, and combining the parts using heuristics. TECCL also uses some heuristics, either using the integer program solver's heuristics when exceeding some time limit, or trying instead to solve a simpler greedy version of the problem. Even

with these considerations, the methods don't scale very well, still struggling as we increase the number of GPUs.

Tree-based algorithms

There are also a series of works including Blink [15] and ForestColl [20] that ignore steps completely and instead just specify a set of trees that can route all of the data. In simple settings, a tree is sufficient to describe the data routing in AllGather, as well as in ReduceScatter, among other collectives, since (in the case of AllGather) we can make the assumption that every branch in the tree corresponds to making a copy of the data, as there is never a need to perform a reduction. The reverse is true for Reduction-based collectives.

These tree-based methods inherently use a fluid model, as they never specify what steps should be taken, instead indicating what routes data should take in a steady-state to maximize throughput. One benefit to these methods is that they typically scale much better than other methods, since there are polynomial time algorithms for packing spanning trees. However, due to the reliance on spanning trees, these methods are much less flexible than the step-based methods we described earlier, as they are fundamentally unable to incorporate vertices with different purposes, complicated bandwidth limitations, while the step-based methods we discussed could be modified to account for these considerations.

2.5.1 Examples

In this section, we explore existing methods on some toy examples. We hope to illustrate various ways in which generating collective communication algorithms is challenging and highlight some drawbacks of existing works.

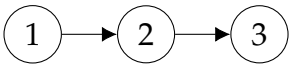

Figure 2.1: A topology to demonstrate pipelining

Step-based algorithms

We begin by looking at an example that shows step schedules always lose some performance in comparison to fluid solutions. This is not a surprise, but we give a concrete example. (ForestColl makes this argument a bit more robustly, but we take a slightly more reserved position.)

In any step schedule, we fix a minimum atomic unit of data (usually called a chunk). This quantization of data prevents us from achieving the same latency as if we could transfer infinitely small units of data. Consider the Broadcast primitive on the topology in Figure 2.1. In an optimal fluid solution, both the link from 1 to 2 and the link from 2 to 3 should be fully saturated at all times. But in any step solution, when the last chunk is transmitted from 2 to 3, the link from 1 to 2 is unused. So no step solution can achieve the same latency as a fluid solution.

Modeling memory

No existing methods provide the ability to explicitly model memory as a component in their representation of topologies. Why is explicitly modeling memory useful? Our first example shows how taking advantage of memory can produce algorithms we didn't could not consider before.

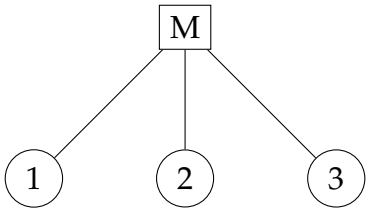

Figure 2.2: A topology to demonstrate using memory

Look at the topology in Figure 2.2. In a Broadcast from 1, without modeling memory, one optimal algorithm transmits data from GPU 1 to GPU 2 and GPU 2 to GPU 3. But based on the real structure, what this algorithm actually suggests is that we write data from 1 to memory, then read that data to GPU 2. Then we would write the data from 2 to memory, and GPU 3 reads that data. If we correctly model memory, then we can find an algorithm that writes the data from GPU 1 to memory, then reads both GPU 2 and GPU 3 read that data.

In a model with discrete chunks of data and uniform link bandwidths, the first algorithm requires four steps, where one step is the duration of a single transmission. Our second algorithm requires only two steps, because both reads from memory can be done in parallel. Even if we do not use a model with discrete data chunks, our second algorithm requires writing less data to memory.

Where packing spanning trees fails

In this part, we wish to show the ways in which ForestColl cannot handle some things that we desire, particularly with respect to memory. We should note that in these examples, we are thinking about the Broadcast primitive, while ForestColl actually doesn't claim to generate algorithms for Broadcast. It does generate algorithms for AllGather, and we are using these simpler examples for Broadcast to show ways in which the algorithms it generates for AllGather are suboptimal.

Shared bandwidth

Let's begin by showing how ForestColl and other spanning tree packing methods cannot handle more complicated bandwidth limitations. In addition to link bandwidths, we may also want to consider "component bandwidths". For our examples here, we are thinking about memory bandwidth: a limit on the amount of data that can be read and written from the host memory.

Consider a simple topology like the one in Figure 2.3a, where we have 4 GPUs all connected to a host memory, let x be the bandwidth on each of the links, and y be the memory bandwidth. For the purpose of this example, let's suppose that our only goal is to treat the memory as a switch, except that it has its own bandwidth limitation. (This means we are ignoring the fact that memory components typically allow storage of data and copying data.) Then in ForestColl's implementation, we would perform a process called vertex splitting to replace the switch with

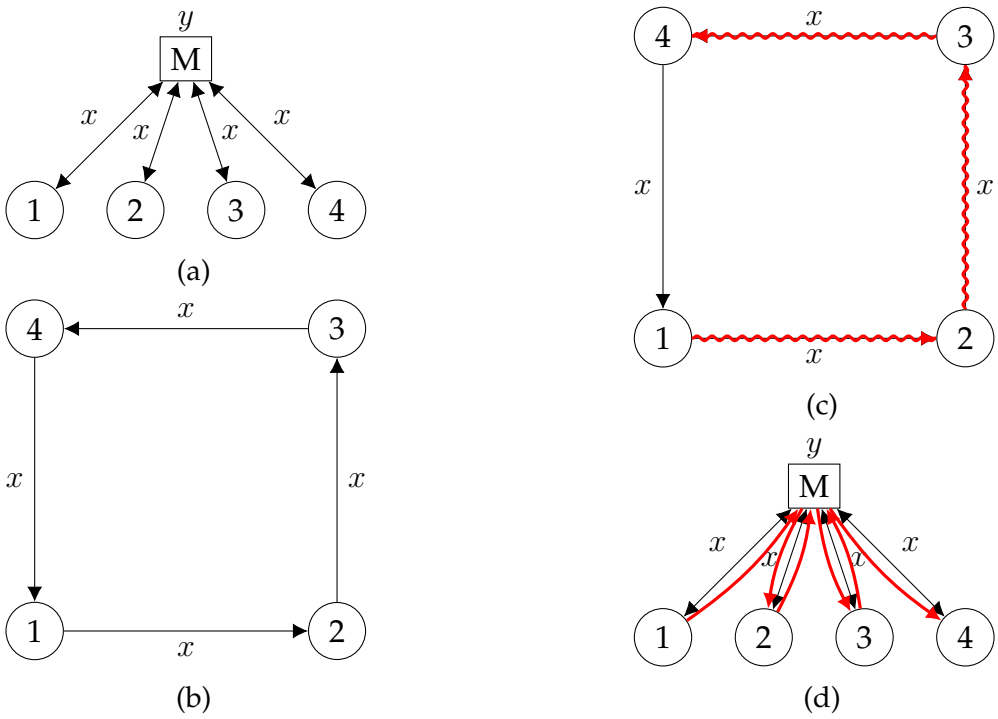

Figure 2.3: A topology to show the difficulty of modeling memory bandwidth

some subset of the implied direct edges connecting the incident vertices. The result of this vertex splitting produces a graph like Figure 2.3b, where the vertices incident to the switch now form a ring. Then some optimal solution on that topology for Broadcast might look like Figure 2.3c, where the highlighted red edges indicate how data should travel in a steady-state. And when we map this solution back to the original topology, we get an optimal spanning tree as described in Figure 2.3d. But the problem here is that we are totally ignoring the memory bandwidth: if $y < 6x$, then the algorithm described in Figure 2.3d wouldn't have a throughput of x like expected, but actually a throughput of $y/6$. So the solution it finds would not be optimal.

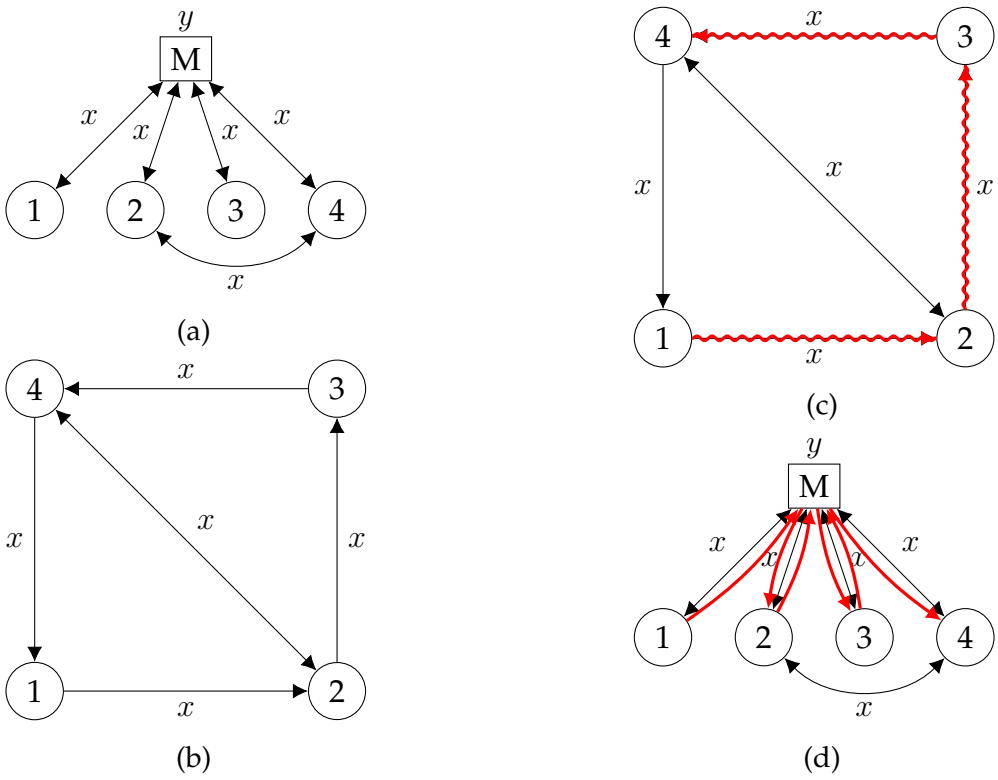

Figure 2.4: An unnatural topology to emphasize the difficulty of modeling memory bandwidth

We can see an even more extreme version of this phenomenon by considering a more unnatural topology like the one in Figure 2.4a. Though such a topology is unlikely to exist in practice, this example emphasizes the need for proper handling of memory vertices. In this example, the same basic logic holds as before: Vertex splitting would yield a representation like the one in Figure 2.4b. Figure 2.4c highlights one optimal solution on that representation, and Figure 2.4d shows the mapping back to the original topology. Not only is the solution not optimal in this case, there is a better solution that routes data along a different path: using edges $(1, 2), (2, 3), (2, 4)$, sending x units of data along each one. Since this pro-

posed solution only uses 4 edges incident to the memory component, it would have throughput equal to $\min\{y/4, x\}$. So not only is there a problem with reporting the correct optimal throughput, the path that data is transmitted along may also be different in an optimal solution.

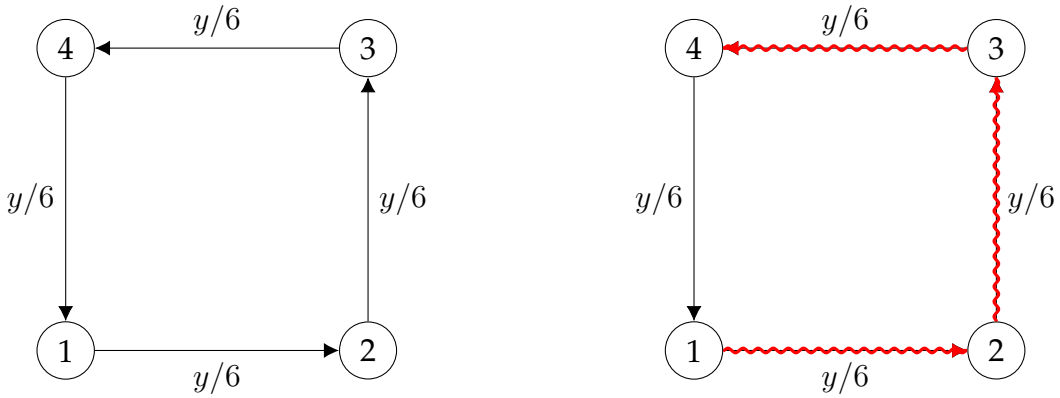

Figure 2.5: An attempt to cleverly adjust link bandwidths

One thing we might try here would be to adjust the link bandwidths when we remove the switch to actually reflect the memory bandwidth. In fact, if we try setting the link bandwidths from the example beginning in Figure 2.3 to $\min\{y/6, x\}$ like in Figure 2.5a, everything actually works in this simple example. But we cannot guarantee that doing something like this will work in all cases.

In more complex topologies, where there is some alternative way to route the data not through this depicted memory vertex (as in Figure 2.4), even this strategy ends up with a suboptimal solution. For instance, if we were to try the same thing in this case, our adjusted bandwidths look like in Figure 2.6a. An optimal spanning tree is in Figure 2.6b, and the tree in Figure 2.5b also has the same apparent

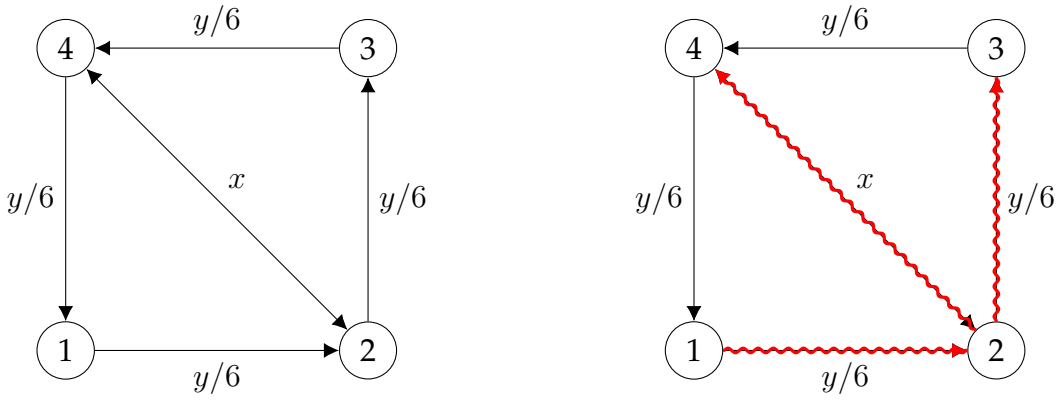

Figure 2.6: An attempt to cleverly adjust link bandwidths

throughput. But if we map Figure 2.6b back to a tree on the original graph, we can observe that this tree supports a throughput of $\min\{y/4, x\}$. So once again, the solution returned is not optimal. (And since we may return Figure 2.5b instead of Figure 2.6b, the data may even be transmitted differently from the optimal algorithm.)

One last thing we might try is to separate the memory vertex in our representation into two vertices joined by a one-directional link with the desired memory bandwidth. This correctly models our situation, but ForestColl cannot handle even this, because it has a requirement that the input graphs are Eulerian, which this modified input is not.

Memory as a non-mandatory compute vertex

Memory behaves in a way that is distinct from both GPUs and switches. It allows data to be copied, like a GPU, but it typically cannot perform reductions, like a switch. And unlike GPUs, data need not be transmitted to memory.

While it is not so hard to imagine how we would model this in a linear programming formulation to support this behavior in TE-CCL or TACCL, we will give an example to show that this concept is fundamentally incompatible with spanning tree packing.

The technique used by ForestColl finds spanning trees on its graph representation. But there are two things at odds here: to properly model memory, we want to treat it like a vertex in the graph, so that it can copy data. But we don't need data to be transmitted to the corresponding vertex, so the objects we are looking for are no longer spanning trees, but Steiner trees. And packing Steiner trees is NP-hard.

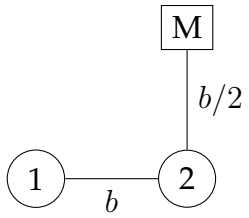

Figure 2.7: A topology to demonstrate treating memory as a GPU fails

So to use spanning tree packing, we need to either give up copying at memory (treating it as a switch), or we need to require data to be transmitted to all mem-

ory (treating it like a GPU). We've already described our inability to accurately model memory as a switch above. And Figure 2.7 shows a simple example where treating memory as a GPU creates worse solutions. For a broadcast from vertex 1, treating memory as a GPU produces a solution tree using both links with throughput $b/2$. But the optimal solution just uses the link connecting vertices 1 and 2 for a throughput of b.

CHAPTER 3

A LINEAR PROGRAMMING APPROACH

In this chapter, we present our approach to generating collective communication algorithms. We slowly build up the ideas needed to understand our approach, beginning with step-based algorithms for AllGather on a single chunk of data (CM1), then move on to divisible data (CM2 and CM3). We describe our technique to leverage our step-based formulation to optimize over all step sizes simultaneously before finally generalizing the model by adding switches.

Throughout this section, we motivate and describe several mathematical programming formulations that capture our problem. We use these formulations for two primary purposes: generating collective communication algorithms and proving lower bounds on the quality of all possible algorithms. Our first formulations are exact characterizations, which means that solving those formulations will immediately provide an optimal algorithm for the indicated collective operation. But most formulations we present are relaxations, and hence some solutions may not correspond to feasible algorithms. In such cases, we can still use the value of an optimal solution to the formulation as a lower bound, which may allow us to prove the optimality of an algorithm. And by repeatedly strengthening the relaxations, we can eventually find a provably optimal solution.

3.1 AllGather

Let us begin by considering an extremely simplified version of the problem for AllGather: We assume that all the components in our topology support the copy operation. (For example, we may consider a topology with only GPUs and memory.) We further assume that we are using communication model CM1 (i.e., we wish to construct a step-based algorithm with completely indivisible data) and that we require all transmissions to finish within the duration of a step. These assumptions are not necessary for our methods, but they will allow us to introduce the main ideas with fewer distractions.

Time-expanded networks

Since we ultimately desire a step-based algorithm, it is natural to think of routing our data through a time-expanded network. For any graph representation of a topology G, and a parameter T representing the number of steps required to complete the collective, we can construct a time-expanded network G^T as follows. We create $(T + 1)$ copies of each vertex v in G, which we will label as $(v, 0), (v, 1), \ldots, (v, T)$. Then for each edge (v_1, v_2) in G, we draw directed edges from vertex (v_1, t) to $(v_2, t + 1)$ for $t \in \{0, \ldots, T - 1\}$. In general, we will use the notation V^T and E^T to refer to the vertices and to the edges of G^T. This time-expanded network now shows how we can route data between vertices over time. For vertices $v \in V_b$ that have a self-loop to indicate that data may be stored, we

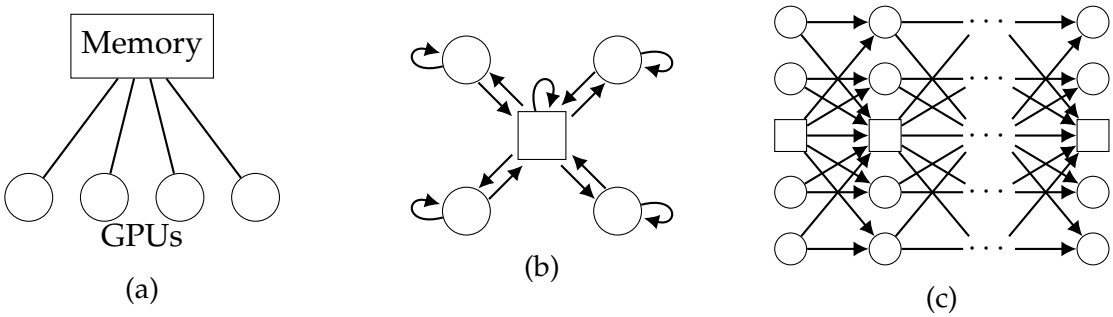

Figure 3.1: (a) shows a simple server topology with 4 GPUs and one memory. (b) depicts the directed graph representation with 5 vertices and directed arcs indicating how data can travel. (c) shows the result of converting (b) into a time-expanded network.

create a hold-over arc (that is, an arc from (v, t) to $(v, t+1)$) in our time-expanded network. Figure 3.1 illustrates the construction of a time-expanded network, starting with a physical topology (a), representing it as a graph (b), and the resulting time-expanded network (c).

We also introduce 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 $\mu(v^T) = v$.

Modeling collective communication as multicommodity flow

Network flow problems are optimization formulations that have long been used to model the control of traffic in networks. In a multicommodity flow problem, there is a set of distinct commodities, each with specified origin and destination vertices as well as a positive demand requirement indicating how much flow must be sent between them, and each must individually satisfy flow conservation constraints; collectively, these commodities satisfy shared capacity limitations along each edge. The multicommodity flow problem is well-studied: by using decision variables that specify the amount of flow transmitted for a given commodity for a given edge, this problem can be solved efficiently as a linear program, provided we do not require *integer* flow values. See [1, Ch.17] for a more thorough discussion of multicommodity flows.

In order to ensure that all data is routed to its desired destinations, we modify a standard multicommodity flow formulation. Our general strategy will be to create flow variables f that ensure all data is routed correctly through G^T following flow constraints, then additionally adding usage variables w that describe how much bandwidth is required to support these flows. The relationship between flow variables and usage variables is specific to the collective.

In particular, we create a commodity for each source-destination pair, where the sources are the vertices $V_o^T = \{(v, 0) : v \in V_o\}$ and the destinations $V_d^T = \{(v, T) : v \in V_d\}$. For AllGather, in which we have n GPUs, the set of origin-

destination pairs is the n^2 pairs from the Cartesian product of origin and destination vertices. (That is, there is a commodity between every origin vertex and every destination vertex.)

As discussed in §2.4, there are two factors we attempt to optimize for. The first is latency, which we control with our choices of T and s (our step size). As one of our desiderata for our generated algorithms, among those with minimum latency, we would like to find the best such algorithm with respect to some other secondary objective. To achieve this for a general class of secondary objectives, we introduce costs c_e for every edge $e \in E$, which indicate the cost-per-unit of transmitting data along e, and return a solution of minimum cost among latency-optimal solutions. For example, to minimize the amount of data uploaded from and downloaded to GPUs, we can set $c_e = 1$ for edges incident to a GPU, and $c_e = 0$ for all other edges. (One additional consideration is that we require $c_e = 0$ for any self-loops e. This is not generally much of an imposition, because we almost never think about storing data being expensive in comparison to transmitting data.)

Non-linear formulation for AllGather under CM1

We first write a mixed integer non-linear program (MINLP) that describes algorithms for AllGather under our assumptions that all vertices support copies and transmissions occur within one step. For this formulation, along with the number

of steps T, we also need to specify the step size s. We create flow variables $f_{e,o,d}$ for every edge $e \in E^T$, and every commodity with origin-destination pair (o, d).

We then impose the following standard flow conservation constraints on our f variables:

$$\sum_{e \in \delta^-(v)} f_{e,o,d} - \sum_{e \in \delta^+(v)} f_{e,o,d} = 0 \qquad \text{for } o \in V_o^T, d \in V_d^T, v \in V^T \setminus \{o, d\} \tag{3.1}$$

$$\sum_{e \in \delta^-(o)} f_{e,o,d} - \sum_{e \in \delta^+(o)} f_{e,o,d} = 1 \qquad \text{for } o \in V_o^T, d \in V_d^T \tag{3.2}$$

$$\sum_{e \in \delta^-(d)} f_{e,o,d} - \sum_{e \in \delta^+(d)} f_{e,o,d} = -1 \qquad \text{for } o \in V_o^T, d \in V_d^T \tag{3.3}$$

We also create usage variables $w_{e,o}$ for every edge and every origin vertex in G^T to represent how much bandwidth the data flows actually require.

To relate our w and f variables, we note that the data we wish to send to all destinations from the same source is identical, and since we allow copies at all vertices, there is never a reason to transmit multiple copies of the same data. In fact, the amount of data that needs to be transmitted along any link can be given by the maximum over all data flows using that link:

$$w_{e,o} = \max_{d \in D} f_{e,o,d} \qquad \text{for } e \in E^T, o \in V_o^T, . \tag{3.4}$$

Finally, we must restrict the bandwidth usage based on the actual link bandwidths. That is, for every set of links L with corresponding bandwidth limitation $b(L)$, we restrict the combined usage on all links in L by data from all sources, and

we enforce this restriction at every step. To find the correct bound, we determine the amount of data the links in L can support in the duration of one step, $sb(L)$, giving the following additional constraint.

$$\sum_{e \in L} \sum_{o \in V_o^T} w_{\kappa(e,t),o} \leq sb(L) \qquad \text{for } L \in \mathcal{L}, t \in \{0, \ldots, T-1\}. \tag{3.5}$$

Using our cost vector and imposing integrality constraints on f to ensure that flow behaves as an indivisible unit (i.e., we do not allow fractions of the data to be transmitted along different edges), our full linear program is:

$$\begin{aligned}
\min_{f,w} \quad & \sum_{e \in E} \sum_{t=0}^{T-1} c_e \sum_{o \in V_o^T} w_{\kappa(e,t),o} \\
\text{s.t.} \quad & \sum_{e \in \delta^-(v)} f_{e,o,d} - \sum_{e \in \delta^+(v)} f_{e,o,d} = 0 && \text{for } o \in V_o^T, d \in V_d^T, v \in V^T \setminus \{o, d\} \\
& \sum_{e \in \delta^-(o)} f_{e,o,d} - \sum_{e \in \delta^+(o)} f_{e,o,d} = 1 && \text{for } o \in V_o^T, d \in V_d^T \\
& \sum_{e \in \delta^-(d)} f_{e,o,d} - \sum_{e \in \delta^+(d)} f_{e,o,d} = -1 && \text{for } o \in V_o^T, d \in V_d^T \\
& w_{e,o} = \max_{d \in D} f_{e,o,d} && \text{for } e \in E^T, o \in V_o^T \\
& \sum_{e \in L} \sum_{o \in V_o^T} w_{\kappa(e,t),o} \leq sb(L) && \text{for } L \in \mathcal{L}, t \in \{0, \ldots, T-1\} \\
& f \in \{0, 1\}, w \geq 0
\end{aligned} \tag{3.6}$$

We can now argue that this formulation accurately describes our problem.

Theorem 1. *MINLP*(3.6) *is an exact formulation for AllGather under CM1.*

Proof. We argue this by showing that a minimal algorithm with latency T and step size s corresponds to a solution to 3.6, and argue that the correspondence is bidirectional. (We say that an algorithm is minimal if no data transmissions can be removed from the algorithm without making it infeasible.)

We start with an algorithm for AllGather under CM1 with step size s and T total steps. Remember that this algorithm consists of saying exactly what data is transferred at each of the T steps. Since the data is indivisible in this model, we always send 1 unit of data at each step, and we can identify pieces of data by their origin. Say we send data originating at GPU o from i to j at step t. Then let $\hat{o} = (o, 0)$, and $\hat{e} = \kappa((i, j), t)$ be the corresponding edge in G^T. We set $w_{\hat{e},\hat{o}} = 1$. Since this is a valid algorithm, the bandwidth on any set of links L is no more than $b(L)$ at each step. Which means that no more than $sb(L)$ data is sent on edges in L each step. Summing over all data identified by their origins gives exactly the constraint (3.5).

Now, we just need to find feasible f values. Since the algorithm is correct, data originating at o must be routed to every destination d. For every origin-destination pair (o, d) and edge $\hat{e} \in E^T$, we set the corresponding $f_{\hat{e},\hat{o},\hat{d}}$ variable to 1 if the data that eventually arrived at d in the algorithm was transmitted along a link corresponding to (i, j) at time t, and zero otherwise. (If the data remained in memory at the same component v for step t of the algorithm, we set $f_{\kappa((v,v),t),\hat{o},\hat{d}}$ to 1.)

This assignment of f variables clearly satisfies the flow constraints: at the des-

tination, there are only in-edges, and one of them has value 1. The reverse is true for the origin. And for all other intermediate vertices, the data leaves some component if and only if it entered the component at a previous time. It also satisfies constraint 3.4: $w_{\hat{e},\hat{o}} = 1$ when some data is transmitted along the edge at step t. By the minimality of the algorithm, this can only occur if this data eventually arrives at some destination d, and so $f_{\hat{e},\hat{o},\hat{d}} = 1$. And if $w_{\hat{e},\hat{o}} = 0$, we trivially have that $f_{\hat{e},\hat{o},\hat{d}} = 0$ for all $\hat{d}$.

For the other direction, we begin with a solution to MINLP(3.6). We use the w values to determine exactly what data to send at each step: if $w_{\hat{e},\hat{o}} = 1$, where $\hat{e} = ((i,t),(j,t+1))$, we transmit data from component i to component j at step t. It is easy to check that this is a correct algorithm: The bandwidth constraint requires the data transmitted to not exceed the bandwidth of any link. And the flow constraints ensure that the data arrives at all GPUs.

Finally, since we have established that the w variables exactly capture the amount of data transmitted between components at each step, the objectives also match. □

An integer programming formulation

We can easily write a mixed integer linear program (MILP) if we remove the non-linearity in the previous formulation by replacing constraint (3.4) with

$$w_{e,o} \geq f_{e,o,d} \qquad \text{for } e \in E^T, o \in V_o^T, d \in V_d^T. \tag{3.7}$$

Then our full integer programming formulation is

$$\begin{aligned}
\min_{f,w} \quad & \sum_{e \in E} \sum_{t=0}^{T-1} c_e \sum_{o \in V_o^T} w_{\kappa(e,t),o} \\
\text{s.t.} \quad & \sum_{e \in \delta^-(v)} f_{e,o,d} - \sum_{e \in \delta^+(v)} f_{e,o,d} = 0 && \text{for } o \in V_o^T, d \in V_d^T, v \in V^T \setminus \{o, d\} \\
& \sum_{e \in \delta^-(o)} f_{e,o,d} - \sum_{e \in \delta^+(o)} f_{e,o,d} = 1 && \text{for } o \in V_o^T, d \in V_d^T \\
& \sum_{e \in \delta^-(d)} f_{e,o,d} - \sum_{e \in \delta^+(d)} f_{e,o,d} = -1 && \text{for } o \in V_o^T, d \in V_d^T \\
& w_{e,o} \geq f_{e,o,d} && \text{for } e \in E^T, o \in V_o^T, d \in V_d^T \\
& \sum_{e \in L} \sum_{o \in V_o^T} w_{\kappa(e,t),o} \leq sb(L) && \text{for } L \in \mathcal{L}, t \in \{0, \ldots, T-1\} \\
& f \in \{0,1\}, w \geq 0
\end{aligned} \tag{3.8}$$

And naturally, we find that this formulation is also exact.

Theorem 2. *MILP*(3.8) *is an exact formulation for AllGather under CM1.*

Proof. We just need to argue a correspondence between optimal solutions to MILP(3.8) and optimal solutions to MINLP(3.6).

Given a solution f, w to MINLP(3.6), let $f'_{e,o,d} = f_{e,o,d}$ for all $e \in E^T$, $o \in V^T_o$, and $d \in V^T_d$, and $w'_{e,o} = w_{e,o}$ for all e and o. Then f', w' is a feasible solution to MILP(3.8) with the same objective value, as the only difference is constraint (3.7), which still holds since $w'_{e,o} = \max_{d \in D} f'_{e,o,d}$. For the other direction, given a optimal solution f', w' to MILP(3.8), similarly let $f_{e,o,d} = f'_{e,o,d}$ for all $e \in E^T$, $o \in V^T_o$, and $d \in V^T_d$, and set $w_{e,o} = \max_{d^T \in V^t_d} f_{e,o,d}$ for all e and o. Then f, w is a feasible solution to MINLP(3.6) because the flow constraints are identical, (3.4) is true by construction, and (3.5) holds because $w_{e,o} \leq w'_{e,o}$ for all e and o. In fact, $w_{e,o} = w'_{e,o}$ for all e where $c_{\mu(e)} > 0$, because otherwise we could have reduced the value of $w'_{e,o}$ to get a better objective for MILP(3.8), violating our optimality assumption. Therefore, the objective value $\sum_{e \in E} \sum_{t=0}^{T-1} c_{\kappa(e,t)} \sum_{o \in O} w_{e,o}$ is the same. □

As handled in the proof of Theorem 2, we cannot guarantee that in an optimal solution, $w_{e,o} = \max_d f_{e,o,d}$ when $c_e = 0$. We could prevent this from ever occurring by adding a tiny value to all cost coefficients c_e so that all are strictly positive. But since the information we need to construct an algorithm is already contained in the f variables, there is no need to do this. That is to say, although $w_{e,o}$ may exceed $\max_d f_{e,o,d}$, it is trivial to calculate the true usage of edges since our f variables are all feasible, and any incorrect w values will not contribute to the cost.

Interpreting solutions

For a fixed choice of s, we can do binary search on T to determine the minimum value of T for which there are feasible solutions. These solutions correspond directly to algorithms. If we were determined to use this formulation to find an optimal step-based algorithm, we would also need to search over many choices for s. With our assumption that transmissions occur within a single step, there are at most $|E|$ choices of s that we need to consider, because if no edge requires exactly s time to transmit the data, we can reduce s for a strictly better solution. And in the case where slow transmissions can take multiple steps, we can use the least common multiple of the bandwidths as the denominator of our step size.

We now address this assumption that all transmissions can occur within a single step. It is not difficult to generalize the previous formulation to allow slow transmissions that require multiple steps by slightly modifying our time-expanded network, but it will not end up becoming important for further evolutions of our formulation.

Even so, we briefly describe how we can modify our time-expanded network to support transmissions that take more than one step. We construct the vertices of G^T as before, creating $(T+1)$ copies of each vertex v in G, which we label as $(v,0),(v,1),\ldots,(v,T)$. Then for each edge (v_1,v_2) in G, for each $t \in \{0,\ldots,T\}$, instead of adding one edge, we will draw edges from vertex (v_1,t) to (v_2,t') for all $t' < t \leq T$.

Later, when we enforce our bandwidth limitations, we will need to enforce more complicated constraints. In particular, for every interval of time (t_1, t_2) and for every $L \in \mathcal{L}$, we will need to enforce that

$$\sum_{(v_1,v_2)\in L} \sum_{o\in V_o^T} \sum_{t=t_1}^{t_2-1} \sum_{t'=t+1}^{t_2} w_{((v_1,t_1),(v_2,t_2)),o} \leq (t_2 - t_1) sb(L)$$

instead of the previous constriant (3.5).

Divisible data

We can continue towards a more general solution by removing another one of our simplifying assumptions: that we could not divide our data. We can easily modify our previous formulation to account for any arbitrary number of chunks (CM2). If we wish to allow dividing the data into K equal-sized pieces, we can simply create K copies of each commodity, each treated independently except for in the bandwidth constraints and in the objective function. We'll have to scale the bandwidth constraints appropriately so that one unit of flow now corresponds to $1/K$ units of data. To do this, we adjust our f and w variables so that they are also indexed by the chunk number k.

$$
\begin{aligned}
\min_{f,w} \quad & \sum_{e\in E}\sum_{t=0}^{T-1} c_e \sum_{o\in O}\sum_{k\in[K]} w_{\kappa(e,t),o,k} \\
\text{s.t.} \quad & \sum_{e\in\delta^-(v)} f_{e,o,d,k} - \sum_{e\in\delta^+(v)} f_{e,o,d,k} = 0 && \text{for } o\in V_o^T,\, d\in V_d^T,\, v\in V^T\setminus\{o,d\},\, k\in[K] \\
& \sum_{e\in\delta^-(o)} f_{e,o,d,k} - \sum_{e\in\delta^+(o)} f_{e,o,d,k} = 1 && \text{for } o\in V_o^T,\, d\in V_d^T,\, k\in[K] \\
& \sum_{e\in\delta^-(d)} f_{e,o,d,k} - \sum_{e\in\delta^+(d)} f_{e,o,d,k} = -1 && \text{for } o\in V_o^T,\, d\in V_d^T,\, k\in[K] \\
& w_{e,o,k} \geq f_{e,o,d,k} && \text{for } e\in E^T,\, o\in V_o^T,\, d\in V_d^T,\, k\in[K] \\
& \sum_{e\in L}\sum_{o\in V_o^T}\sum_{k\in[K]} w_{\kappa(e,t),o,k} \leq K\,sb(L) && \text{for } L\in\mathcal{L},\, t\in\{0,\dots,T-1\} \\
& f\in\{0,1\},\, w\geq 0 && (3.9)
\end{aligned}
$$

Theorem 3. *The MILP* (3.9) *is an exact formulation for AllGather under CM2 with K chunks.*

We do not provide a proof here, because it does not differ from the proof to Theorem 2 in any meaningful way.

Of course, the above model still relies on the assumption that no transmissions take longer than one step. This assumption is less onerous in this setting, as if this is not the case, we can increase the value of K until the assumption holds. However, if we did wish to dispose of the assumption, we could, as before, modify our time-expanded network and adjust our bandwidth constraints. However, we will not write out the formulation here, as it does not add to our discussion.

Fluid model

In our previous formulations, we required that we fixed some number of indivisible chunks before formulating the problem. Now we will consider the fluid model (CM3) in which we do not provide any limitations on the smallest indivisible unit. One way we might think about solving this is by taking limit as the number of chunks becomes very large in our previous formulation.

However, this is not a good idea because our integer program will become extremely large. Instead, we will simply use the same formulation as in MILP(3.8), except with the integrality constraint on f relaxed:

$$
\begin{aligned}
\min_{f,w} \quad & \sum_{e\in E}\sum_{t=0}^{T-1} c_e \sum_{o\in V_o^T} w_{\kappa(e,t),o} \\
\text{s.t.} \quad & \sum_{e\in\delta^-(v)} f_{e,o,d} - \sum_{e\in\delta^+(v)} f_{e,o,d} = 0 && \text{for } o\in V_o^T, d\in V_d^T, v\in V^T\setminus\{o,d\} \\
& \sum_{e\in\delta^-(o)} f_{e,o,d} - \sum_{e\in\delta^+(o)} f_{e,o,d} = 1 && \text{for } o\in V_o^T, d\in V_d^T \\
& \sum_{e\in\delta^-(d)} f_{e,o,d} - \sum_{e\in\delta^+(d)} f_{e,o,d} = -1 && \text{for } o\in V_o^T, d\in V_d^T \\
& w_{e,o} \geq f_{e,o,d} && \text{for } e\in E^T, o\in V_o^T, d\in V_d^T \\
& \sum_{e\in L}\sum_{o\in V_o^T} w_{\kappa(e,t),o} \leq sb(L) && \text{for } L\in\mathcal{L}, t\in\{0,\dots,T-1\} \\
& f, w \geq 0 && (3.10)
\end{aligned}
$$

Now since we have removed all of our integrality constraints, it is possible to solve the above formulation efficiently. We should also note that the removal of

the integrality requirements means that we no longer need to worry about transmissions that take longer than a step: we can simulate them by sending a fraction of the data each step until the transmission is completed.

We should note that LP(3.10) is actually a relaxation, and is not guaranteed to produce feasible algorithms for general topologies. Of course, we still retain the property that if the LP solution is indeed a feasible algorithm, it is guaranteed to be optimal. The reason LP(3.10) is a relaxation is because we cannot guarantee that flows can always combine if they share the same source. In particular, our formulation assumes that if two flows from the same source use the same edge concurrently, then the usage amount is the maximum of these two flow values. However, if these flows represent different segments of the data, they may actually require sending both flows. In this way, the solution to this relaxation could produce usage values that are not feasible. To remedy this, we will first describe an exponentially sized linear program that will prevent flows from combining unless they share their entire path from their source.

We will begin by modifying our time-expanded network to have $2^{|V_d|}-1$ copies, one for every non-empty subset of V_d. And we will modify our edge set so that if $(v_1, v_2) \in E$, then our time-expanded network will contain edges from (v_1, t, S_1) to (v_2, t, S_2) for all subsets $\emptyset \neq S_2 \subseteq S_1 \subseteq V_d$. Then we formulate a nearly-standard multicommodity flow problem on this modified graph, with joint capacities. The

formulation has objective:

$$\min \sum_{(v_1,v_2)=e\in E} \sum_{t=0}^{T-1} \sum_{S_1,S_2:S_1\subseteq S_2\subseteq V_d} c_e \sum_{o\in V_o^T} f_{((v_1,t,S_1),(v_2,t+1,S_2)),o}.$$

We have flow constraints where we define source vertices of the form $(v, 0, V_d)$ for $v \in V_o$ and destination vertices of the form (v, T, v) for $v \in V_d$. To simplify writing our flow constraints, we will let $\hat{O} = \{(v, 0, V_d) : v \in V_o\}$ be the origin vertices in our new time-expanded network, and similarly let $\hat{D} = \{(v, T, v) : v \in V_d\}$. We also require an analogous bandwidth constraint that looks like

$$\sum_{(v_1,v_2)\in L} \sum_{S_1,S_2:S_1\subseteq S_2\subseteq V_d} \sum_{o\in\hat{O}} f_{((v_1,t,S_1),(v_2,t+1,S_2)),o} \le Rb(L) \qquad \text{for } L \in \mathcal{L}, t \in \{0,\dots,T-1\}.$$

Then our formulation is

$$\begin{aligned}
\min_f \quad & \sum_{(v_1,v_2)=e\in E} \sum_{t=0}^{T-1} \sum_{S_1,S_2:S_1\subseteq S_2\subseteq V_d} c_e \sum_{o\in\hat{O}} f_{((v_1,t,S_1),(v_2,t+1,S_2)),o} \\
\text{s.t.} \quad & \sum_{e\in\delta^-(v)} f_{e,o,d} - \sum_{e\in\delta^+(v)} f_{e,o,d} = 0 && \text{for } o \in \hat{O}, d \in \hat{D}, v \in V^T \setminus \{o, d\} \\
& \sum_{e\in\delta^-(o)} f_{e,o,d} - \sum_{e\in\delta^+(o)} f_{e,o,d} = 1 && \text{for } o \in \hat{O}, d \in \hat{D} \\
& \sum_{e\in\delta^-(d)} f_{e,o,d} - \sum_{e\in\delta^+(d)} f_{e,o,d} = -1 && \text{for } o \in \hat{O}, d \in \hat{D} \\
& \sum_{(v_1,v_2)\in L} \sum_{S_1,S_2:S_1\subseteq S_2\subseteq V_d} \sum_{o\in\hat{O}} f_{((v_1,t,S_1),(v_2,t+1,S_2)),o} \le Rb(L) \\
& && \text{for } L \in \mathcal{L}, t \in \{0,\dots,T-1\} \\
& f, w \ge 0 && (3.11)
\end{aligned}$$

It is tedious but not difficult to show that formulation LP(3.11) captures exactly feasible solutions to our fluid model, by noticing that a vertex corresponding to subset S is only allowed to have outgoing flow that eventually reaches exactly the terminals in S.

Fortunately for us, typical topologies have some special structure that means we do not need to worry about the above exponential size formulation. We will not write a rigorous proof, but instead provide some intuition.

There is only one reason that LP(3.10) may not be exact: a solution may cheat by aggregating some flows that cannot be combined. However, we can leverage properties of our underlying topologies to say that this cheating should not happen. In slightly more detail, typical topologies have the property that in an optimal feasible algorithm, the bottleneck bandwidths (which we can determine by solving many flow problems on G) are edges in one of two locations: either to/from our terminals (GPUs), or between blocks in a natural block-structure in our underlying topology. In both of these cases, we can prove (though we do not do so here) that we can find a feasible algorithm of equal cost that has none of the "cheating" described earlier. Therefore, LP(3.10) is typically sufficient for our purposes.

In order to take full advantage of arbitrarily small pieces of data, we also need to be able to make our step sizes arbitrarily small. But that presents a problem: we cannot make the step size infinitely small, and even approximating with a tiny step size causes computational problems by exploding the number of steps

required, and hence the size of the input graph. We address this problem in Chapter 4.

Handling switches

We are now ready to do away with our remaining assumption on our AllGather problem. Previously, we assumed that all vertices support the copy operation. This does not hold generally; in particular, switches may not be multicast (i.e., they may not allow copies) and instead data may only pass directly through, more like traditional flows. Furthermore, we will eventually produce algorithms that require reductions (Reduce and ReduceScatter) by reversing solutions. In this setting, vertices that support copies become those that support reductions, and many vertices (e.g. memory, switches) do not typically support reductions. Therefore, it is important to be able to correctly represent vertices that do not allow copies.

The following formulations focus on enforcing that copies cannot be created at switches. We first describe methods of handling this problem exactly, then provide a more tractable relaxation. Although relaxations are not exact and may produce infeasible algorithms, we can still use them to find provably optimal algorithms. Toward the end of the section, we will describe a process to iteratively strengthen the relaxations until we arrive at a provably optimal solution.

First, we should identify why our previous formulations no longer work. Our previous assumption that it is never beneficial to send multiple copies of the same

data along a single link is no longer necessarily true. It is not hard to imagine a scenario where this might occur: if we were sending the same data to two destinations via a switch, we would need to transmit the same data on the source to switch path twice. Therefore, we cannot be sure that the bandwidth usage of a link is determined by the maximum of our f variables.

Formulation via duplicate vertices

Instead of reworking our formulation, we can reuse our existing formulations by modifying our time-expanded network. First, let $\overline{V_c}$ be the set of vertices in G that do not allow copies (i.e., $\overline{V_c} = V \setminus V_c$). Our approach is to make multiple copies of vertices in $\overline{V_c}^T$. This allows us to ensure that if flows exit $v \in \overline{V_c^T}$ toward different vertices, then they travel through different copies of v.

In particular, we can modify our time-expanded network G^T so that for every $v \in \overline{V_c}$ and every step t, we make $|V_c|$ vertices, one associated with each vertex in V_c. We can denote these (v, t, u) for $u \in v_c$. Now, we describe how we add edges in this adjusted time-expanded network. For edges $\{(v_1, v_2) : v_1, v_2 \in V_c\}$ between vertices in V_c, we add edges from (v_1, t) to $(v_2, t+1)$ as before. For edges $\{(v_1, v_2) : v_1 \in \overline{V_c}, v_2 \in V_c\}$ to vertices in V_c, we add edges from (v_1, t) to (v_2, t, u) for each $u \in V_c$. For edges $\{(v_1, v_2) : v_1, v_2 \in \overline{V_c}\}$ between vertices in $\overline{V_c}$, we add edges (v_1, t, u) to (v_2, t, u) for each $u \in V_c$. And finally, for edges $\{(v_1, v_2) : v_1 \in \overline{V_c}, v_2 \in V_c\}$ to vertices in $\overline{V_c}$, we only add edges (v_1, t, v_2) to $(v_2, t+1)$. Call this modified time-expanded network G_*^T. Then we can use LP(3.10) on G_*^T to find algorithms

on topologies with switches for AllGather under CM3.

This approach is reasonable if there are only a constant number of components that do not support copy (or reduce). But in the worst case, we are almost squaring the number of vertices and edges in our time-expanded network, which makes the size of the formulation grow very quickly.

A relaxation

To address the scalability issues of the previous formulation, we turn to relaxations. We start with the standard multicommodity flow-based model we used previously, and add additional constraints or variables to try to prevent copies at vertices $v \in \overline{V_c}$ in our model. One natural attempt is to impose a flow-like constraint on the usage variables (w) at switch vertices, by adding to LP(3.10) the constraint

$$\sum_{e \in \delta^-(v)} w_{e,o} - \sum_{e \in \delta^+(v)} w_{e,o} = 0 \qquad \text{for } o \in V_o^T, v \in \overline{V_c}^T. \tag{3.12}$$

This gives the following linear program:

$$
\begin{aligned}
\min_{f,w} \quad & \sum_{e\in E}\sum_{t=0}^{T-1} c_e \sum_{o\in V_o^T} w_{\kappa(e,t),o} \\
\text{s.t.} \quad & \sum_{e\in\delta^-(v)} f_{e,o,d} - \sum_{e\in\delta^+(v)} f_{e,o,d} = 0 && \text{for } o\in V_o^T, d\in V_d^T, v\in V^T\setminus\{o,d\} \\
& \sum_{e\in\delta^-(o)} f_{e,o,d} - \sum_{e\in\delta^+(o)} f_{e,o,d} = 1 && \text{for } o\in V_o^T, d\in V_d^T \\
& \sum_{e\in\delta^-(d)} f_{e,o,d} - \sum_{e\in\delta^+(d)} f_{e,o,d} = -1 && \text{for } o\in V_o^T, d\in V_d^T \\
& w_{e,o} \geq f_{e,o,d} && \text{for } e\in E^T, o\in V_o^T, d\in V_d^T \\
& \sum_{e\in L}\sum_{o\in V_o^T} w_{\kappa(e,t),o} \leq sb(L) && \text{for } L\in\mathcal{L}, t\in\{0,\ldots,T-1\} \\
& \sum_{e\in\delta^-(v)} w_{e,o} - \sum_{e\in\delta^+(v)} w_{e,o} = 0 && \text{for } o\in V_o^T, v\in \overline{V_c}^T \\
& f, w \geq 0 && (3.13)
\end{aligned}
$$

Before we move on, it is important to understand why this formulation is a relaxation. While the formulation indeed ensures that copies are only made at vertices in V_c, by adding constraint (3.12), our w variables are no longer independent of each other, so we can no longer guarantee that the w variables become the maximum of the f variables at optimal solutions. Therefore, the w variables may misrepresent (by underestimating) the amount of data transmitted along a link. So transmitting data according to the flow variables in an optimal solution could require sending more data across a link than its capacity allows. Alternatively, since our w values may not reflect the traffic along an edge, we may get a solu-

tion whose actual cost is more than what the relaxation told us. In both cases, we cannot use the solution to the relaxation to generate an feasible algorithm.

There are more sophisticated ways of further constraining our relaxation to try to correctly capture bandwidth usage for arbitrary inputs. These strategies typically have many more constraints, mitigating the benefits in comparison to a duplicate vertex formulation.

We will include one such attempt below. In this formulation, we try to keep tighter control of the usage variables w. To do this, we create variables $x_{a,e,o}$ for integers $a = 1, \ldots, n$, where we want $x_{a,e,o}$ to represent the amount of data on the edge that is intended to arrive at a destinations. If all the data on the edge will only end up at a single destination (like on the last edge before the destination), $x_{1,e,o}$ should equal the entire data flow on the edge. And if the data on the edge will be copied to end up at all n destinations, then $x_{n,e,o}$ should equal the amount of data on the edge. More specifically, we want to ensure that $\sum_{a=1}^{n} x_{a,e,o} = w_{e,o}$ and $\sum_{a=1}^{n} a x_{a,e,o} = \sum_{d \in D} f_{e,o,d}$. We also will add a constraint to address that the number of destinations for a piece of data is non-increasing over time. In full, this yields the following augmented formulation which strengthens the previous relaxation.

$$
\begin{aligned}
\min_{f,w} \quad & \sum_{e\in E}\sum_{t=0}^{T-1} c_e \sum_{o\in V_o^T} w_{\kappa(e,t),o} \\
\text{s.t.} \quad & \sum_{e\in\delta^-(v)} f_{e,o,d} - \sum_{e\in\delta^+(v)} f_{e,o,d} = 0 && \text{for } o\in V_o^T, d\in V_d^T, v\in V^T\setminus\{o,d\} \\
& \sum_{e\in\delta^-(o)} f_{e,o,d} - \sum_{e\in\delta^+(o)} f_{e,o,d} = 1 && \text{for } o\in V_o^T, d\in V_d^T \\
& \sum_{e\in\delta^-(d)} f_{e,o,d} - \sum_{e\in\delta^+(d)} f_{e,o,d} = -1 && \text{for } o\in V_o^T, d\in V_d^T \\
& \sum_{a=1}^{n} x_{a,e,o} = w_{e,o} && \text{for } e\in E^T, o\in V_o^T \\
& \sum_{a=1}^{n} a x_{a,e,o} = \sum_{d\in V_d^T} f_{e,o,d} && \text{for } e\in E^T, o\in V_o^T \\
& w_{e,o} \geq f_{e,o,d} && \text{for } e\in E^T, o\in V_o^T, d\in V_d^T \\
& \sum_{e\in\delta^+(d)} x_{1,e,o} = 1 && \text{for } o\in V_o^T, d\in V_d^T \\
& \sum_{e\in\delta^-(v)}\sum_{a'=a}^{n} x_{a',e,o} \leq \sum_{e\in\delta^+(v)}\sum_{a'=a}^{n} x_{a',e,o} \\
& && \text{for } a\in[n], o\in V_o^T, v\in V^T\setminus(\overline{V_c}^T\cup V_o^T\cup V_d^T) \\
& \sum_{e\in\delta^-(v)} x_{a,e,o} = \sum_{e\in\delta^+(v)} x_{a,e,o} && \text{for } a\in[n], o\in V_o^T, v\in\overline{V_c}^T \\
& \sum_{e\in L}\sum_{o\in V_o^T} w_{\kappa(e,t),o} \leq sb(L) && \text{for } L\in\mathcal{L}, t\in\{0,\dots,T-1\} \\
& f, w \geq 0 && (3.14)
\end{aligned}
$$

Finding provably optimal algorithms

We now outline a procedure to find provably optimal algorithms by leveraging relaxations. We first solve our chosen relaxation, and check if the resulting solution corresponds to a feasible algorithm of the same cost; we can do this by computing whether the data transmissions suggested by our w variables actually routes all data to its desired destination. If the algorithm is feasible, we are done and have proved that the solution we generated is optimal, because the optimal value to any relaxation is a lower bound on the optimal algorithm objective value.

Otherwise, at some point in the algorithm, our relaxation solution indicated usage (w) on some edge that did not match the usage implied by flow variables. If this happened due to trying to improperly combine flows, we can duplicate vertices in the fashion used for formulation LP(3.11) so that this combining would no longer be a feasible solution. Note that we do not need to create copies for all subsets, just enough so that the current solution would become infeasible.

We can also have improper usage variable values because of switch behavior. If the previous kind of error in usage values did not occur, the problem must occur at a switch vertex. (If not incident to a switch, $w_{e,o} = \max_d f_{e,o,d}$ would be feasible and correctly describe the usage on the edge.) Therefore, we can duplicate this switch vertex as described in the duplicate vertex formulation so that our current w values would be infeasible. As before, we do not need to make all duplicate copies, just enough so that the flows contributing to the incorrect w value would

now be handled by different vertices.

After doing this for all switches where the w variables do not accurately capture usage, we end up with a slightly larger time-expanded network, where the size of the graph has increased by the number of duplicate vertices we added. We can then re-solve our relaxation on this new graph, and repeat this process until we find a solution where all w variables correctly match the usage implied by the flow variables, which will be provably optimal.

3.2 Other collectives

In this section, we now use the techniques we have developed for AllGather to create formulations for all the collectives we have discussed, except for AllReduce, which we will defer to the next chapter, as some of the discussion relies on techniques we have not yet introduced.

Broadcast

Broadcast does not require any changes from AllGather. The formulations given in the previous section work the same as for AllGather with one modification: we must adjust our set of origin vertices V_o to be a single vertex, corresponding to the root GPU from which data is broadcasted.

Reduce and ReduceScatter

For ReduceScatter (and also Reduce), the standard technique for generating solutions is to reverse solutions for AllGather (or Broadcast). One assumption that allows this reversal to happen 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. While GPUs typically allow both actions, and switches tend to support neither, memory tends to only support copying data but not performing reductions. Fortunately, this is not a problem for us, since in our formulations, particularly LP(3.13) and LP(3.14) we were able to decouple the ability to store data and copy data, we can simply use the set V_c to indicate the set of vertices that can perform reductions, use our formulations as intended for generating solutions to AllGather, then reverse the solution in time.

3.2.1 All-to-all

All-to-all is a simpler collective operation, since algorithms require neither copies nor reductions. Therefore, we don't even need usage variables in our formulation, we can determine how much data is transmitted along an edge directly from flow variables. With this in mind, we can find All-to-all solutions by solving a standard multicommodity flow problem with our added bandwidth constraints.

We just need to modify constraint (3.5) in LP(3.10) to be

$$\sum_{e \in L} \sum_{o \in V_o^T} \sum_{d \in V_d^T} f_{\kappa(e,t),o,d} \leq sb(L) \qquad \text{for } L \in \mathcal{L}, t \in \{0, \ldots, T-1\} \tag{3.15}$$

and similarly modify our objective, so our new linear program is:

$$\begin{aligned}
\min_{f,w} \quad & \sum_{e \in E} \sum_{t=0}^{T-1} c_e \sum_{o \in V_o^T} \sum_{d \in V - d^T} f_{\kappa(e,t),o,d} \\
\text{s.t.} \quad & \sum_{e \in \delta^-(v)} f_{e,o,d} - \sum_{e \in \delta^+(v)} f_{e,o,d} = 0 && \text{for } o \in V_o^T, d \in V_d^T, v \in V^T \setminus \{o, d\} \\
& \sum_{e \in \delta^-(o)} f_{e,o,d} - \sum_{e \in \delta^+(o)} f_{e,o,d} = 1 && \text{for } o \in V_o^T, d \in V_d^T \\
& \sum_{e \in \delta^-(d)} f_{e,o,d} - \sum_{e \in \delta^+(d)} f_{e,o,d} = -1 && \text{for } o \in V_o^T, d \in V_d^T \\
& \sum_{e \in L} \sum_{o \in V_o^T} \sum_{d \in V_d^T} f_{\kappa(e,t),o,d} \leq sb(L) && \text{for } L \in \mathcal{L}, t \in \{0, \ldots, T-1\} \\
& f, w \geq 0.
\end{aligned} \tag{3.16}$$

CHAPTER 4

COMPUTING STEPS VIA TREES

4.1 Flow schedules via step-based solutions

We next describe how we can find solutions with minimum latency given arbitrarily small step sizes without solving a very large linear program. To do this, we separate the spatial problem of determining how to route the data from the temporal problem of when to transmit each piece of data. Suppose that we are told what route every piece of data takes (i.e. what sequence of links it is transmitted along), ignoring bandwidth constraints. Given this information, we will construct a solution for an arbitrarily large number of chunks K in which all chunks of data follow the route provided and no bandwidth constraints are violated.

We fix some value of T as an upper bound on the total number of transmissions any piece of data might require to reach its destination. We also choose a value R to be the desired latency of our solution. There are several ways we can determine the best value of R, including binary search or solving an auxiliary flow problem, but we will not include details here. Then we formulate a modified linear program.

We keep the same flow conservation constraints and constraints relating f and w from LP(3.10). The only change is to how we express our bandwidth constraints. Instead of requiring them to be true for every time step, we just want a collective

bound over all time steps to ensure that the *average* throughput of each link does not exceed the bound implied by the desired latency R.

$$\sum_{t=0}^{T-1} \sum_{e \in L} \sum_{o \in V_o^T} w_{\kappa(e,t),o} \leq Rb(L) \qquad \text{for } L \in \mathcal{L} \tag{4.1}$$

Then our new linear program to determine how to route the data is as follows:

$$\begin{aligned}
\min_{f,w} \quad & \sum_{e \in E} \sum_{t=0}^{T-1} c_e \sum_{o \in V_o^T} w_{\kappa(e,t),o} \\
\text{s.t.} \quad & \sum_{e \in \delta^-(v)} f_{e,o,d} - \sum_{e \in \delta^+(v)} f_{e,o,d} = 0 && \text{for } o \in V_o^T, d \in V_d^T, v \in V^T \setminus \{o,d\} \\
& \sum_{e \in \delta^-(o)} f_{e,o,d} - \sum_{e \in \delta^+(o)} f_{e,o,d} = 1 && \text{for } o \in V_o^T, d \in V_d^T \\
& \sum_{e \in \delta^-(d)} f_{e,o,d} - \sum_{e \in \delta^+(d)} f_{e,o,d} = -1 && \text{for } o \in V_o^T, d \in V_d^T \\
& w_{e,o} \geq f_{e,o,d} && \text{for } e \in E^T, o \in V_o^T, d \in V_d^T \\
& \sum_{t=0}^{T-1} \sum_{e \in L} \sum_{o \in V_o^T} w_{\kappa(e,t),o} \leq Rb(L) && \text{for } L \in \mathcal{L} \\
& f, w \geq 0. && (4.2)
\end{aligned}$$

Now we claim that instead of trying to optimize LP(3.10) over all possible choices of T and s, it is sufficient to optimize LP(4.2). To avoid confusion arising from choices of parameters, we will use the notation LP(3.10)$[s, T]$ to indicate the formulation (3.10) using step size s and number of steps T. Similarly, we will use LP(4.2)$[R, T]$ to indicate formulation (4.2) with desired latency R and number of steps T.

Theorem 4. *LP(4.2) is equivalent to LP(3.10) in the following sense: Given a solution to LP(4.2)[R_1, T_1], there are values s_1 and T_1' such that we can construct a solution to LP(3.10)[s_1, T_1'] of equal cost and with latency $s_1 T_1' < R_1 + \epsilon$ for any $\epsilon > 0$. And given a solution to LP(3.10)[s_2, T_2] with latency $s_2 T_2$, we can construct a solution to LP(4.2)[$s_2 T_2, T_2$] of equal cost.*

This theorem says that we can convert between solutions for LP(3.10) and LP(4.2) with identical objective cost and identical latency (up to an arbitrarily small ϵ term). Thus, we can find an optimal solution for LP(3.10) over all possible choices of s by solving LP(4.2). The proof of Theorem 4 is notationally cumbersome, so we will first outline the general principle so that it is clear why these two formulations should be equivalent before providing the details.

It suffices to describe the conversion in both directions and verify that all constraints still hold. The conversion going from LP(3.10)[s, T] to LP(4.2)[sT, T] is trivial, so we will worry more about the other direction.

In particular, we must argue that we can take a solution (f, w) to LP(4.2)[R, T] and use it to construct a solution to LP(3.10)[s', T'] for some choice of s' and T'. Our general idea will be to divide the data into small chunks, then route each chunk according to (f, w), staggering the transmissions so that each chunk is transmitted at a different time. This will allow us to argue that since our constructed solution obeys the bandwidth constraints on average, it also obeys them at each step.

First, we choose an integer K according to $K > (T-1)/\epsilon$, and also set $s' = R/K$. We begin our construction by dividing the data into K identically sized chunks, though these chunks need not be indivisible. We then begin routing one chunk at every step for the first K steps, where the route of each chunk is determined by (f, w). In particular, if (f, w) indicates that we should transmit x units of data along edge e at time t, in our new solution, for each chunk $k \in [K]$, we will transmit x/K units of data along edge e at time $t + k - 1$.

Using this strategy, we can route all our data in $T' = T + K - 1$ steps.

Now we provide the complete proof.

Proof of Theorem 4. To prove the theorem, we need to convert between solutions for LP(3.10) and LP(4.2).

Let's begin with the simpler direction: taking a solution (f, w) to LP(3.10)$[s, T]$ and constructing a solution to LP(4.2)$[sT, T]$ with identical objective value. This is very simple; we claim that (f, w) is also a feasible solution for LP(4.2)$[sT, T]$. Note that both formulations have the same objective and constraints (3.1), (3.2), (3.3), and (3.7). And constraint (3.5) implies constraint (4.1) because $T \geq 1$.

Now we will more rigorously describe and justify the more complicated direction of the equivalence.

We will first need to introduce some new notation. In a time-expanded network G^T, let E^t be the set of edges that originate at time t. Specifically, $E^t =$

$\{((i,t'),(j,t'+1)) \in E^T : t' = t\}$.

As previously stated, fix some integer $K > (T-1)/\epsilon$ to be the number of chunks we divide our input data into, let step size $s' = R/K$, and let $T' = T+K-1$.

Now, we will describe how to use a solution (f, w) to LP(4.2)$[R, T]$ to construct a feasible solution to program LP(3.10)$[s', T']$. In particular, we will describe values (f', w') that satisfy this LP.

Let $\hat{e} \in E^{T'}$ and $t' = \tau(\hat{e})$. Then for every $\hat{e}$, $o \in V_o^{T'}$ and $d \in V_d^{T'}$,

$$f'_{\hat{e},o,d} = \begin{cases} \sum_{t=t'-K+1}^{t'} \frac{f_{\kappa(\mu(\hat{e}),t),o,d}}{K} + \frac{K-1-t'}{K} & \text{if } t' \leq K-1 \text{ and } \mu(\hat{e}) = (\mu(o), \mu(o)) \\ \sum_{t=t'-K+1}^{t'} \frac{f_{\kappa(\mu(\hat{e}),t),o,d}}{K} + \frac{t'-T+1}{K} & \text{if } t' \geq T \text{ and } \mu(\hat{e}) = (\mu(d), \mu(d)) \\ \sum_{t=t'-K+1}^{t'} \frac{f_{\kappa(\mu(\hat{e}),t),o,d}}{K} & \text{otherwise} \end{cases}$$

where $f_{\kappa(\mu(\hat{e}),t),o,d} = 0$ if $t < 0$ or $t \geq T$.

Similarly, we define

$$w'_{\hat{e},o} = \begin{cases} \sum_{t=t'-K+1}^{t'} \frac{w_{\kappa(\mu(\hat{e}),t),o}}{K} + \frac{K-1-t'}{K} & \text{if } t' \leq K-1 \text{ and } \mu(\hat{e}) = (\mu(o), \mu(o)) \\ \sum_{t=t'-K+1}^{t'} \frac{w_{\kappa(\mu(\hat{e}),t),o}}{K} + \frac{t'-T+1}{K} & \text{if } t' \geq T \text{ and } \mu(\hat{e}) = (\mu(d), \mu(d)) \text{ for } d \in V_d^T \\ \sum_{t=t'-K+1}^{t'} \frac{w_{\kappa(\mu(\hat{e}),t),o}}{K} & \text{otherwise} \end{cases}$$

where again, $w_{\kappa(\mu(\hat{e}),t),o} = 0$ if $t < 0$ or $t \geq T$.

These expressions come from overlaying the K staggered solutions (f, w), where each one is only on $1/K$ of the data. The extra terms in the first two cases capture the edge-cases where some data has yet to leave its origin, or some data has already arrived at its destination.

We first argue that (f', w') is feasible for all the constraints in LP(3.10)$[s', T']$.

For given $o' \in V_o^{T'} d \in V_d^{T'}$, first define $o = (\mu(o'), 0) \in V_o^T$ and $d = (\mu(d'), T) \in V_d^T$. Then for any $v \in V^{T'}$, where $\mu(v)$ is neither $\mu(o')$ nor $\mu(d')$,

$$\begin{aligned}
\sum_{\hat{e} \in \delta^-(v)} f'_{\hat{e},o',d'} - \sum_{e \in \delta^+(v)} f'_{\hat{e},o',d'} &= \sum_{\hat{e} \in \delta^-(v)} \sum_{t=\tau(\hat{e})-K+1}^{\tau(\hat{e})} \frac{f_{\kappa(\mu(\hat{e}),t),o,d}}{K} - \sum_{\hat{e} \in \delta^+(v)} \sum_{t=\tau(\hat{e})-K+1}^{\tau(\hat{e})} \frac{f_{\kappa(\mu(\hat{e}),t),o,d}}{K} \\
&= \sum_{e \in \delta^-(\mu(v))} \sum_{t=\tau(v)-K}^{\tau(v)-1} \frac{f_{\kappa(e,t),o,d}}{K} - \sum_{e \in \delta^+(\mu(v))} \sum_{t=\tau(v)-K+1}^{\tau(v)} \frac{f_{\kappa(e,t),o,d}}{K} \\
&= \sum_{t=\tau(v)-K+1}^{\tau(v)} \left(\sum_{e \in \delta^-(\mu(v))} \frac{f_{\kappa(e,t-1),o,d}}{K} - \sum_{e \in \delta^+(\mu(v))} \frac{f_{\kappa(e,t),o,d}}{K} \right) \\
&= \sum_{t=\tau(v)-K+1}^{\tau(v)} \left(\sum_{\hat{e} \in \delta^-((\mu(v),t))} \frac{f_{\hat{e},o,d}}{K} - \sum_{\hat{e} \in \delta^+((\mu(v),t))} \frac{f_{\hat{e},o,d}}{K} \right).
\end{aligned}$$

Note in the second line, the e under the sum is in G, not $G^{T'}$.

Now, since $\mu(v)$ is neither o' nor d', by constraint (3.1) in LP(4.2)$[R, T]$, the expression evaluates to zero, so constraint (3.1) in LP(3.10)$[s', T']$ holds.

The other cases are slightly more complicated. Say $\mu(v) = \mu(o')$. If $\tau(v) \geq K$, the expression still evaluates to zero. Otherwise, if $0 < \tau(v) \leq K - 2$, there is an additional $\frac{K-\tau(v)}{K}$ in-flow term from one edge, and a $\frac{K-1-\tau(v)}{K}$ outflow term. This gives us:

$$\sum_{\hat{e}\in\delta^-(v)} f'_{\hat{e},o',d'} - \sum_{e\in\delta^+(v)} f'_{\hat{e},o',d'} = \sum_{t=\tau(v)-K+1}^{\tau(v)} \left(\sum_{\hat{e}\in\delta^-((\mu(v),t))} \frac{f_{\hat{e},o,d}}{K} - \sum_{\hat{e}\in\delta^+((\mu(v),t))} \frac{f_{\hat{e},o,d}}{K} \right) - \frac{1}{K}$$

For all values of t except $t = 0$, the summand evaluates to zero by constraint (3.1) in LP(4.2)$[R, T]$. When $t = 0$, by constraint (3.2) in LP(4.2)$[R, T]$, the summand evaluates to $\frac{1}{K}$. So constraint (3.1) in LP(3.10)$[s', T']$ still holds.

And if $\tau(v) = 0$, the same summand evaluates to $\frac{1}{K}$ for exactly one choice of t, but the extra term in this case is a positive $\frac{K-1}{K}$ instead of $-\frac{1}{K}$. So constraint (3.2) in LP(3.10)$[s', T']$ also holds.

A symmetric argument for the cases where $\mu(v) = d$ shows that constraint (3.3) in LP(3.10)$[s', T']$ also holds.

We need to be slightly more careful in the case where $\mu(o') = \mu(d')$, as the claim that the sum evaluates to zero if $K \leq \tau(v)$ is not true, but nothing about the argument fundamentally changes.

For constraint (3.7) in LP(3.10), we have the following, given $\hat{e}$ falls into the "otherwise" case for the definitions of f' and w':

$$
\begin{aligned}
w'_{\hat{e},o'} &= \sum_{t=\tau(\hat{e})-K+1}^{\tau(\hat{e})} \frac{w_{\kappa(\mu(\hat{e}),t),o}}{K} \\
&\geq \sum_{t=\tau(\hat{e})-K+1}^{\tau(\hat{e})} \frac{\max_d f_{\kappa(\mu(\hat{e}),t),o,d}}{K} \\
&\geq \max_d \sum_{t=\tau(\hat{e})-K+1}^{\tau(\hat{e})} \frac{f_{\kappa(\mu(\hat{e}),t),o,d}}{K} \\
&= \max_d f'_{\hat{e},o',d'}
\end{aligned}
$$

In the other two cases, the additional additive term is the same for both w' and f'. So the constraint holds in these cases as well.

For constraint (3.5) in LP(3.10), we find that for any $L \in \mathcal{L}$ and any $t' \in \{0, \ldots, T-1\}$,

$$
\sum_{e \in L} \sum_{o' \in V_o^{T'}} w'_{\kappa(e,t'),o'} = \sum_{e \in L} \sum_{o \in V_o^T} \sum_{t=t'-K+1}^{t'} \frac{w_{\kappa(e,t),o}}{K} \leq \sum_{e \in L} \sum_{o \in V_o^T} \sum_{t=0}^{T-1} \frac{w_{\kappa(e,t),o}}{K} \leq \frac{Rb(L)}{K} = s'b(L),
$$

where we are using the fact that we have no bandwidth limitations on self-loops.

The last task is to show that our conversion preserves objective cost. We can write:

$$\begin{aligned}
\sum_{e\in E}\sum_{t'=0}^{T'-1} c_e \sum_{o'\in V_o^{T'}} w'_{\kappa(e,t'),o'} &= \sum_{e\in E} c_e \sum_{t'=0}^{T+K-2} \sum_{o\in V_o^T} \sum_{t=t'-K+1}^{t'} \frac{w_{\kappa(e,t),o}}{K} \\
&= \sum_{e\in E} c_e \sum_{o\in V_o^T} \sum_{t=0}^{T-1} \sum_{t'=t}^{t+K-1} \frac{w_{\kappa(e,t),o}}{K} \\
&= \sum_{e\in E} c_e \sum_{o\in V_o^T} \sum_{t=0}^{T-1} K\frac{w_{\kappa(e,t),o}}{K} \\
&= \sum_{e\in E}\sum_{t=0}^{T-1} \sum_{o\in V_o^T} c_e w_{\kappa(e,t),o}.
\end{aligned}$$

Notice that we don't need to worry about the other cases in the definition for w' because those edges are guaranteed to have zero cost. □

4.2 AllReduce

We are now ready to discuss generating algorithms for AllReduce. AllReduce is more difficult to generate algorithms for compared to other primitives because we cannot guarantee that an optimal algorithm has no reductions, or that it makes no copies of data; in fact, both are necessary. This means that even if we are given the routes of all the data through the topology, it is not obvious how much bandwidth the solution requires. In particular, given flow values analogous to the f variables in our previous formulations, there is no obvious way to determine corresponding usage values.

We will write an exact formulation, where we explicitly keep track of both where data is copied and where data is reduced, along with what sources of data have been reduced together.

We introduce flow variables $f_{e,U,S}$ for every edge e, every subset U of V_o^T, and every subset S of V_d^T. Let $\mathcal{P}$ be the set of all subpartitions of V_d^T and $\mathcal{Q}$ be the set of all subpartitions of V_o^T. We also introduce y variables for every vertex $v \in V^T$ and every subpartition $P \in \mathcal{P}$; these indicate how much data is copied at a vertex toward which destinations. Similarly, we introduce z variables for every $v \in V^T$ and every subpartition $Q \in \mathcal{Q}$ to indicate how much data is reduced from which sources. Recall that we have defined V_c as the set of vertices that allow copies. We will also now introduce V_r as the set of vertices that allow reductions.

First, we will have to develop different flow conservation constraints based on our new flow variables. First, we should understand that each flow variable, indexed by U and S, indicates the reduction of data from all inputs U, which is destined to be copied to all destinations S. In these, the change in the quantity of data of time U, S at a vertex for a particular U, S pair is determined not just by the net flow, but the net copy operation, captured by

$$\sum_{P \in \mathcal{P}: \cup P = S} y_{v,U,P} - \sum_{P \in \mathcal{P}: S \in P} y_{v,U,P}$$

and the net reduction operation, captured by

$$\sum_{Q \in \mathcal{Q}: U \in Q} z_{v,Q,S} - \sum_{Q \in \mathcal{Q}: \cup Q = U} z_{v,Q,S}.$$

We also give constraints to prevent operations where they are not possible, and the relevant bandwidth constraints.

We have the following formulation:

$$\min_{f,y,z} \sum_{e\in E}\sum_{t=0}^{T-1} c_e \sum_{U\subseteq V_o^T}\sum_{S\subseteq V_d^T} f_{\kappa(e,t),U,S}$$

$$\text{s.t.} \quad \sum_{e\in\delta^-(v)} f_{e,U,S} - \sum_{e\in\delta^+(v)} f_{e,U,S} + \sum_{P\in\mathcal{P}:\cup P=S} y_{v,U,P} - \sum_{P\in\mathcal{P}:S\in P} y_{v,U,P} + \sum_{Q\in\mathcal{Q}:U\in Q} z_{v,Q,S} - \sum_{Q\in\mathcal{Q}:\cup Q=U} z_{v,Q,S} = \begin{cases} 1 & \text{if } S = V_d^T \text{ and } U = \{v\} \\ -1 & \text{if } S = \{v\} \text{ and } U = V_o^T \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{for } U \subseteq V_o^T,\ S \subseteq V_d^T,\ v \in V^T$$

$$y_{v,U,P} = 0 \qquad \text{for } U \subseteq V_o^T,\ P \in \mathcal{P},\ v \in \overline{V_c}^T$$

$$z_{v,Q,S} = 0 \qquad \text{for } S \subseteq V_d^T,\ Q \in \mathcal{Q},\ v \in \overline{V_r}^T$$

$$\sum_{e\in L}\sum_{U\subseteq V_o^T}\sum_{S\subseteq V_d^T} f_{\kappa(e,t),U,S} \le sb(L) \qquad \text{for } L \in \mathcal{L},\ t \in \{0,\dots,T-1\}$$

$$f, y, z \ge 0$$

Unfortunately, this formulation is prohibitively massive. The number of y and z variables scales with the number of subpartitions of n (or equivalently the number of partitions of $n+1$), which grows faster than 2^n (see Bell numbers in e.g. [17]).

We also note that we cannot create an exact formulation by duplicating some of the vertices in our time expanded graph like 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. As an easy example, the in-degree of a switch vertex copy is not the same as the out-degree. So duplicating vertices in this manner can only support copy operations or reduction operations, but not both simultaneously.

Relaxations

As it is difficult to write a reasonably sized exact formulation, we will begin with the simplest relaxation. We return to our standard techniques of adding usage variables on top of a standard multicommodity flow formulation. Here, we redefine our w variables to be just dependent on the edge, but not on the data's origin or destination, because we know that the maximum of all f variables on an edge is a lower bound for the total usage on the edge.

$$
\begin{aligned}
\min_{f,w} \quad & \sum_{e \in E} \sum_{t=0}^{T-1} c_e w_{\kappa(e,t)} \\
\text{s.t.} \quad & \sum_{e \in \delta^-(v)} f_{e,o,d} - \sum_{e \in \delta^+(v)} f_{e,o,d} = 0 && \text{for } o \in V_o^T, d \in V_d^T, v \in V^T \setminus \{o, d\} \\
& \sum_{e \in \delta^-(o)} f_{e,o,d} - \sum_{e \in \delta^+(o)} f_{e,o,d} = 1 && \text{for } o \in V_o^T, d \in V_d^T \\
& \sum_{e \in \delta^-(d)} f_{e,o,d} - \sum_{e \in \delta^+(d)} f_{e,o,d} = -1 && \text{for } o \in V_o^T, d \in V_d^T \\
& w_e \geq f_{e,o,d} && \text{for } e \in E^T, o \in V_o^T, d \in V_d^T \\
& \sum_{e \in L} \sum_{o \in V_o^T} w_{\kappa(e,t),o} \leq sb(L) && \text{for } L \in \mathcal{L}, t \in \{0, \ldots, T-1\} \\
& f, w \geq 0
\end{aligned}
$$

Unfortunately, when evaluating this formulation on real inputs, we find that it performs extremely poorly. Our usage variables completely fail to accurately describe the usage of edges, so the resulting solutions are nearly unusable. Naturally, we then attempt to find a stronger relaxation.

The second formulation we introduce attempts to capture the usage on each edge by explicitly enumerating all possible reductions. To do this, we introduce variables $w_{e,S}$ for every edge $e \in E^T$ and every subset $S \subseteq V_o^T$, which represent the usage on the edge by a reduction of exactly the inputs in S. To govern how data passes through the time-expanded network, we have a constraint that says if a reduction of inputs S leaves some vertex v, all inputs $o \in S$ must have arrived at v as a component of some other reduction $S' \subseteq S$. In particular, for any set of

inputs S, it should be true for every origin $o \in S$ that the total amount of data from o arriving to v, when considering all possible intermediate reductions to S that include o, should be enough so that we can send enough data out of the vertex without violating conservation on the o commodity. This gives the constraint:

$$\sum_{e' \in \delta^+(v)} \sum_{S' \subseteq S : o \in S'} w_{e',S'} \geq w_{e,S} \quad \text{for } S \in 2^{V_o^T}, o \in S, v \in V_r^T \setminus \{o, d\}, e \in \delta^-(v) \quad (4.3)$$

The remaining additional constraints that we need ensure that no reductions occur at vertices that do not support them, that data begins as a reduction of a single piece of data (i.e., no reductions have occurred), and that enough data arrives at the destination.

Combining these gives the following formulation.

$$\min_{f} \sum_{e \in E^T} c_e \sum_{S \in 2^{V_o^T}} w_{e,S}$$

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

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

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

$$\sum_{S \in 2^{V_o^T} : o \in S} w_{e,S} \geq f_{e,o,d} \qquad \text{for } e \in E^T, o \in V_o^T, d \in V_d^T$$

$$\sum_{e' \in \delta^+(v)} \sum_{S' \subseteq S : o \in S'} w_{e',S'} \geq w_{e,S} \qquad \text{for } S \in 2^{V_o^T}, o \in S, v \in V_r^T \setminus \{o, d\}, e \in \delta^-(v)$$

$$\sum_{e \in \delta^-(v)} w_{e,S} = \sum_{e \in \delta^+(v)} w_{e,S} \qquad \text{for } S \in 2^{V_o^T}, v \in \overline{V_r}^T \setminus \{o, d\}$$

$$w_{e,\{o\}} = 1 \qquad \text{for } o \in V_o^T, e \in \delta^-(o)$$

$$w_{e,S} = 0 \qquad \text{for } o \in V_o^T, e \in \delta^-(o), S \in 2^{V_o^T} \setminus \{o\}$$

$$\sum_{e \in \delta^+(d)} \sum_{S \in 2^{V_o^T} : o \in S} w_{e,S} = 1 \qquad \text{for } o \in V_o^T, d \in V_d^T$$

$$\sum_{e \in L} \sum_{S \in 2^{V_o^T}} w_{\kappa(e,t),S} \leq sb(L) \qquad \text{for } L \in \mathcal{L}, t \in \{0, \ldots, T-1\}$$

$$w \in \{0, 1\} \qquad (4.4)$$

Even this formulation isn't quite exact. In particular, it constraint (4.3) allows for the reduction of two pieces of data which share some components, which does not work for general reduction operations. (In some cases, this isn't a problem,

for example if our reduction operation is a maximum function or a minimum function.)

Furthermore, we cannot relax this solution to allow for fractional w variables without allowing many bad scenarios. The main concern is that by allowing fractional values, fractional flows can have the same source of data so we're actually getting multiple copies of the same information instead of all the information. (That is, this formulation cannot easily be adapted form CM3.)

Expressing AllReduce as a combination of other collectives.

Since the previous formulations are not very useful, we will consider the approach frequently used to generate algorithms for AllReduce, where we perform ReduceScatter followed by AllGather on the result.

We begin with an argument about the structure of optimal solutions, with the claim that such solutions tend to have reductions take place before making copies. We present the following theorem without proof to provide some justification for why this should be true.

Theorem 5. *Given an AllReduce algorithm in which some piece of data is copied and later reduced, we can find an algorithm with identical latency that transmits less total data (where we sum the total amount of data sent over all edges) where the reduction happens first.*

Though this theorem does not guarantee that there is an optimal solution with reductions before copies for any arbitrary objective, it is still decent justification to restrict such solutions.

Even so, this does not necessarily mean that the best strategy is to do a ReduceScatter followed by an AllGather. There are two particular ways in which this may not be optimal. 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 criteria should be true, this does not mean that we need to wait for all reductions to finish before we can begin any copy operations. We may be able to benefit by parallelizing the beginning of the AllGather algorithm and the end of the ReduceScatter algorithm.

We briefly describe the modifications we make to our formulations to address these concerns. For the first concern, to be fully general, we introduce an additional variable that describes what portion of the total reductions is completed on each vertex. For the second, we lean on our fluid formulation (LP(4.2)), with which we can ensure that for each piece of data, the reduce before copy criteria holds true. Then we combine our formulations for AllGather and ReduceScatter, where the formulations are only linked by the shared bandwidth constraints and shared costs.

$$\min_{f,x,w} \sum_{e\in E}\sum_{t_1=0}^{T^{(1)}-1} c_e \sum_{d\in V_d^{T^{(1)}}} w^{(1)}_{\kappa(e,t_1),d} + \sum_{e\in E}\sum_{t_2=0}^{T^{(2)}-1} c_e \sum_{o\in V_o^{T^{(2)}}} w^{(2)}_{\kappa(e,t_2),o}$$

s.t.

$$\sum_{e\in\delta^-(v)} f^{(1)}_{e,o,d} - \sum_{e\in\delta^+(v)} f^{(1)}_{e,o,d} = 0 \qquad \text{for } o\in V_o^{T^{(1)}}, d\in V_d^{T^{(1)}}, v\in V^{T^{(1)}}\setminus\{o,d\}$$

$$\sum_{e\in\delta^-(o)} f^{(1)}_{e,o,d} - \sum_{e\in\delta^+(o)} f^{(1)}_{e,o,d} = 1 \qquad \text{for } o\in V_o^{T^{(1)}}, d\in V_d^{T^{(1)}}$$

$$\sum_{e\in\delta^-(d)} f^{(1)}_{e,o,d} - \sum_{e\in\delta^+(d)} f^{(1)}_{e,o,d} = -1 \qquad \text{for } o\in V_o^{T^{(1)}}, d\in V_d^{T^{(1)}}$$

$$\sum_{a=1}^{n} x^{(1)}_{a,e,d} = w^{(1)}_{e,d} \qquad \text{for } e\in E^{T^{(1)}}, d\in V_d^{T^{(1)}},$$

$$\sum_{a=1}^{n} a x^{(1)}_{a,e,d} = \sum_{o\in V_o^{T^{(2)}}} f^{(1)}_{e,o,d} \qquad \text{for } e\in E^{T^{(1)}}, d\in V_d^{T^{(1)}},$$

$$w^{(1)}_{e,d} \geq f^{(1)}_{e,o,d} \qquad \text{for } e\in E^{T^{(1)}}, o\in V_o^{T^{(1)}}, d\in V_d^{T^{(1)}}$$

$$\sum_{e\in\delta^-(o)} x^{(1)}_{1,e,d} = 1 \qquad \text{for } o\in V_o^{T^{(1)}}, d\in V_d^{T^{(1)}}$$

$$\sum_{e\in\delta^-(v)}\sum_{a'=a}^{n} x^{(1)}_{a',e,d} \geq \sum_{e\in\delta^+(v)}\sum_{a'=a}^{n} x^{(1)}_{a',e,d} \qquad \text{for } a\in[n], d\in V_d^{T^{(1)}}, v\in V_r^{T^{(1)}}\setminus(V_o^{T^{(1)}}\cup V_d^{T^{(1)}})$$

$$\sum_{e\in\delta^-(v)} x^{(1)}_{a,e,d} = \sum_{e\in\delta^+(v)} x^{(1)}_{a,e,d} \qquad \text{for } a\in[n], d\in V_d^{T^{(1)}}, v\in(V_m^{T^{(1)}}\cup V_s^{T^{(1)}})$$

$$\sum_{e\in\delta^-(v)} f^{(2)}_{e,o,d} - \sum_{e\in\delta^+(v)} f^{(2)}_{e,o,d} = 0 \qquad \text{for } o\in V_o^{T^{(2)}}, d\in V_d^{T^{(2)}}, v\in V^{T^{(2)}}\setminus\{o,d\}$$

$$\sum_{e\in\delta^-(o)} f^{(2)}_{e,o,d} - \sum_{e\in\delta^+(o)} f^{(2)}_{e,o,d} = 1 \qquad \text{for } o\in V_o^{T^{(2)}}, d\in V_d^{T^{(2)}}$$

$$\sum_{e\in\delta^-(d)} f^{(2)}_{e,o,d} - \sum_{e\in\delta^+(d)} f^{(2)}_{e,o,d} = -1 \qquad \text{for } o\in V_o^{T^{(2)}}, d\in V_d^{T^{(2)}}$$

$$\sum_{a=1}^{n} x^{(2)}_{a,e,o} = w^{(2)}_{e,o} \qquad \text{for } e\in E^{T^{(2)}}, o\in V_o^{T^{(2)}},$$

$$\sum_{a=1}^{n} a x^{(2)}_{a,e,o} = \sum_{d\in V_d^{T^{(2)}}} f^{(2)}_{e,o,d} \qquad \text{for } e\in E^{T^{(2)}}, o\in V_o^{T^{(2)}},$$

$$w^{(2)}_{e,o} \geq f^{(2)}_{e,o,d} \qquad \text{for } e\in E^{T^{(2)}}, o\in V_o^{T^{(2)}}, d\in V_d^{T^{(2)}}$$

$$\sum_{e\in\delta^+(d)} x^{(2)}_{1,e,o} = 1 \qquad \text{for } o\in V_o^{T^{(2)}}, d\in V_d^{T^{(2)}}$$

$$\sum_{e\in\delta^-(v)}\sum_{a'=a}^{n} x^{(2)}_{a',e,d} \leq \sum_{e\in\delta^+(v)}\sum_{a'=a}^{n} x^{(2)}_{a',e,d} \qquad \text{for } a\in[n], d\in V_d^{T^{(2)}}, v\in V^{T^{(2)}}\setminus(V_s^{T^{(2)}}\cup V_o^{T^{(2)}}\cup V_d^{T^{(2)}})$$

$$\sum_{e\in\delta^-(v)} x^{(2)}_{a,e,d} = \sum_{e\in\delta^+(v)} x^{(2)}_{a,e,d} \qquad \text{for } a\in[n], d\in V_d^{T^{(2)}}, v\in V_s^{T^{(2)}}$$

$$\sum_{e\in L}\sum_{d\in V_d^{T^{(1)}}} w^{(1)}_{\kappa(e,t_1)} + \sum_{e\in L}\sum_{o\in V_o^{T^{(2)}}} w^{(2)}_{\kappa(e,t_2)} \leq sb(L) \qquad \text{for } L\in\mathcal{L}, t_1\in\{0,\dots,T^{(1)}-1\}, t_2\in\{0,\dots,T^{(2)}-1\}$$

$$f^{(1)}, f^{(2)}, x^{(1)}, x^{(2)}, w^{(1)}, w^{(2)} \geq 0$$

CHAPTER 5

EXPLOITING SYMMETRY

Our linear programs are always very structured, so it should not be a surprise that we can use standard decomposition techniques here. We start with a standard Dantzig-Wolfe decomposition, where we describe our linear programs as many nearly-independent subproblems, each of which restricts the routing of data from a single origin. By iteratively solving the subproblems with varying objective functions, we converge to an optimal solution to the original large linear program by only ever solving the subproblems. We augment this procedure with a new method we call Mirrored Dantzig-Wolfe. We define symmetry in the context of our input, and observe that if the input is symmetric, we can always find optimal solutions that are also symmetric. And by constraining our solution to be symmetric, every subproblem in our previous Dantzig-Wolfe decomposition is isomorphic, that is they can all be mapped to solutions to every other subproblem while preserving feasibility. Using this idea, we are able to make equivalent progress solving only one subproblem (as opposed to n subproblems) each iteration.

The size of these subproblems is roughly a factor of n^2 smaller than the original problem, and the number of iterations required does not grow very fast, so we can solve the original linear programs much faster.

5.1 Dantzig-Wolfe decomposition

Although the sizes of all our linear programs so far are polynomial in the size of our underlying graph G, they still may become intractable as the size of the graph grows. For example, for AllGather, the number of variables needed in LP(4.2) is $\Omega(n^3)$. This motivates us to find ways to improve the computation time for our LP solvers.

Still restricting our attention to AllGather, (see LP(4.2)) we may notice that the variables for each choice of origin $o \in V_o^T$ are mostly isolated; 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.

These features make our problem a good candidate for Dantzig-Wolfe decomposition, a technique frequently used for multicommodity flow problems [1]. 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.

In our case, for AllGather, we have a subproblem (S_o) for each $o \in V_o^T$ that looks like:

$$
\begin{aligned}
\min_{f,w} \quad & \sum_{e \in E} \sum_{t=0}^{T-1} \hat{c}_e w_{\kappa(e,t),o} \\
\text{s.t.} \quad & \sum_{e \in \delta^-(v)} f_{e,o,d} - \sum_{e \in \delta^+(v)} f_{e,o,d} = 0 \quad \text{for } d \in V_d^T, v \in V^T \setminus \{o, d\} \\
& \sum_{e \in \delta^-(o)} f_{e,o,d} - \sum_{e \in \delta^+(o)} f_{e,o,d} = 1 \qquad \text{for } d \in V_d^T \\
& \sum_{e \in \delta^-(d)} f_{e,o,d} - \sum_{e \in \delta^+(d)} f_{e,o,d} = -1 \qquad \text{for } d \in V_d^T \\
& w_{e,o} \geq f_{e,o,d} \qquad \text{for } e \in E^T, d \in V_d^T \\
& f, w \geq 0, \qquad (5.1)
\end{aligned}
$$

where the $\hat{c}_e$ are costs dependent on the current master problem solution. To state the master problem, let J_o index the set of extreme points of the polyhedron defined by (S_o). And let $\bar{w}_{e,o}^j$ be the $w_{e,o}$ values at the jth extreme point of the poly-

hedron. Then our full master problem is:

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

We note that none of this reformulation strategy is specific to the AllGather formulation we have chosen. Virtually all of the formulations (excluding those that index variables by subsets) can be decomposed in this same manner.

5.2 Mirrored Dantzig-Wolfe

Even with our Dantzig-Wolfe decomposition, these LPs may take a long time to solve. However, in many cases, we may notice that our Dantzig-Wolfe decomposition is made up of subproblems that seem remarkably similar, especially when we our inputs are symmetric in some sense. In this section, we discuss how we can leverage this idea to greatly improve computation time.

Symmetry

We begin by defining symmetry. Before we state a rigorous definition, we provide some intuition for what we mean. Essentially, we want to capture the phenomenon where some pair of GPUs are indistinguishable: for a pair of GPUs g_1 and g_2, if we removed all labels from our input, there would be no way of telling the difference between g_1 and g_2.

More formally, we say that two vertices g_1 and g_2 are indistinguishable if there is a bijection $\chi : V \to V$ such that $\chi(g_1) = g_2$ and for all vertices $v_1, v_2 \in V$,

- $(v_1, v_2) \in E$ if and only if the edge; $(\chi(v_1), \chi(v_2)) \in E$;
- $v \in V_o$ if and only if $\chi(v) \in V_o$ (and analogously for V_b, V_c, V_d);
- $c_{(v_1,v_2)} = c_{(\chi(v_1),\chi(v_2))}$ for $(v_1, v_2) \in E$;
- $b(L) = b(\{(\chi(v_1), \chi(v_2)) : (v_1, v_2) \in L\})$ for $L \in \mathcal{L}$.

Now we can define symmetry. First, we will introduce a "Symmetry Partition" $\mathcal{P}$ of the vertices V_o into "indistinguishability classes" where for any $P \in \mathcal{P}$, if for $g_1, g_2 \in P$, g_1 and g_2 are indistinguishable. We begin by proving a useful lemma about Symmetry Partitions:

Lemma 6. *Given a Symmetry Partition $\mathcal{P}$, there are functions $\pi : V \times [|P|] \to V$ for each $P \in \mathcal{P}$ such that*

- *$\pi(\cdot, i)$ is a certificate of indistinguishability for $v, \pi(v, i)$ for $v \in P$ for all i,*

- $\{\pi(v,i) : i \in [|P|]\} = P$ *for all* $v \in P$.

Proof. We let H be the group of all bijections $\chi : V \to V$ that preserve the graph structure (that is, they satisfy the above four conditions). Then we may say that two vertices are indistinguishable if and only if they are in the same orbit under the H-action. And therefore, the parts in our Symmetry Partition $\mathcal{P}$ exactly correspond to orbits of H.

Now we construct π for each $P \in \mathcal{P}$. Fix some $v_0 \in P$. Then we know $P = \{h_1(v_0), \ldots, h_{|P|}(v_0)\}$ for some $h_1, \ldots, h_{|P|} \in H$. We let $\pi(v,i) = h_i(v)$.

First, we note that by the definition of our group action, $\pi(\cdot, i)$ acts as a certificate of indistinguishability. And for any $v \in P$, there is some $h' \in H$ such that $h'(v_0) = v$, so we find that

$$\begin{aligned}\{\pi(v,i) : i \in [|P|]\} &= \{g_i(v) : i \in [|P|]\} \\ &= \{g_i(h(v_0)) : i \in [|P|]\} \\ &= \{g_i(v_0) : i \in [|P|]\} \\ &= P.\end{aligned}$$

Therefore, our constructed π satisfies the requirements in the Lemma. □

In the following, we will extend the functions π in the natural way to also apply to edges: $\pi((v_1, v_2), i) = (\pi(v_1), \pi(v_2))$ and sets: $\pi(W, i) = \{\pi(v) : v \in W\}$. We further extend the notation to also work on vertices and edges our con-

structed time-expanded networks: $\pi(v^T, i) = (\pi(\mu(v), i), \tau(v^T))$ for $v^T \in V^T$, and $\pi((v_1^T, v_2^T), i) = (\pi(v_1^T), \pi(v_2^T))$ for $(v_1^T, v_2^T) \in E^T$.

Existence of symmetric solutions

Now, we establish that formulation LP(4.2), (and other similar LP formulations) have optimal symmetric solutions.

A solution is symmetric with respect to π if for all $i \in [|P|]$, $f_{\pi(e,i),\pi(o,i),\pi(d,i)} = f_{e,o,d}$ for all f variables and $w_{\pi(e,i),\pi(o,i)} = w_{e,o}$ for all w variables.

Theorem 7. *Given a solution to LP(4.2), Symmetry Partition $\mathcal{P}$, and associated functions π for each $P \in \mathcal{P}$, we can find a solution to LP(4.2) of equal cost that is symmetric with respect to every π.*

Proof sketch. Take any solution f^0, w^0 to LP(4.2). We will describe a process to find a symmetric solution with equal cost. We will describe the argument for f, but the same argument holds for w.

Consider the all the sets $P_1, \ldots, P_r \in \mathcal{P}$ and their corresponding functions $\pi_1, \ldots, \pi_r$. For each choice of j from 1 to r, we compute $f^j_{e,o,d} = \sum_{i \in [|P_j|]} f^{j-1}_{\pi(e,i),\pi(o,i),\pi(d,i)}/|P_j|$.

Though we will not rigorously show this, it is not hard to check that f^j is symmetric w.r.t. π_j. We can also check that if f^{j-1} is symmetric w.r.t. some $\pi_{j'}$,

then f^j is also symmetric w.r.t. $\pi_{j'}$. These two results are enough to conclude that f^r is symmetric with respect to every π. $\square$

Now that we know that there is a symmetric optimal solution, we can take advantage of the symmetry and only search for a symmetric solution. In particular, we can constrain our solution to ensure that $f_{\pi(e,i),\pi(o,i),\pi(d,i)} = f_{e,o,d}$ for $i \in [|P|], P \in \mathcal{P}$ (and the same for w variables). Notably, now when we perform our Dantzig-Wolfe decomposition, we can 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. Furthermore, they 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.

To be concrete, given Symmetry Partition $\mathcal{P}$, for each $P_k \in \mathcal{P}$, choose $o_k = (v, 0) \in V_o^T$ for some $v \in P_k$, and also define π_k to be the function associated with

P_k. Then we create subproblems (S_k) for each indistiguishability class P_k:

$$
\begin{aligned}
\min_{f,w} \quad & \sum_{e\in E}\sum_{t=0}^{T-1} \hat{c}_e w_{\kappa(e,t),o_k} \\
\text{s.t.} \quad & \sum_{e\in\delta^-(v)} f_{e,o_k,d} - \sum_{e\in\delta^+(v)} f_{e,o_k,d} = 0 && \text{for } d\in V_d^T,\, v\in V^T\setminus\{o,d\} \\
& \sum_{e\in\delta^-(o_k)} f_{e,o_k,d} - \sum_{e\in\delta^+(o_k)} f_{e,o_k,d} = 1 && \text{for } d\in V_d^T \\
& \sum_{e\in\delta^-(d)} f_{e,o_k,d} - \sum_{e\in\delta^+(d)} f_{e,o_k,d} = -1 && \text{for } d\in V_d^T \\
& w_{e,o_k} \geq f_{e,o_k,d} && \text{for } e\in E^T,\, d\in V_d^T \\
& f, w \geq 0. && (5.2)
\end{aligned}
$$

And in our master problem, we replace the bandwidth constraint with the constraint

$$\sum_{t=0}^{T-1}\sum_{e\in L}\sum_{P_k\in\mathcal{P}}\sum_{j\in J_{o_k}}\sum_{i=1}^{|P_k|}\lambda^j_{o_k}\bar{w}^j_{\kappa(\pi_k(e,i),t),o_k} \leq Rb(L)$$

for each bandwidth constraint $L\in\mathcal{L}$. We are now considering each indistinguishability class P_k separately, then determining the contribution of an extreme point to the usage of an edge by finding all the usages that map to the edge via π_k.

Then the master problem becomes:

$$
\begin{aligned}
\min_{\lambda} \quad & \sum_{e \in E} \sum_{t=0}^{T-1} c_e \sum_{P_k \in \mathcal{P}} \sum_{j \in J_{o_k}} \lambda_{o_k}^j \bar{w}_{\kappa(e,t),o_k}^j \\
\text{s.t.} \quad & \sum_{t=0}^{T-1} \sum_{e \in L} \sum_{P_k \in \mathcal{P}} \sum_{j \in J_{o_k}} \sum_{i=1}^{|P_k|} \lambda_{o_k}^j \bar{w}_{\kappa(\pi_k(e,i),t),o_k}^j \leq Rb(L) && \text{for } L \in \mathcal{L} \\
& \sum_{j \in J_{o_k}} \lambda_{o_k}^j = 1 && \text{for } P_k \in \mathcal{P} \\
& \lambda_{o_k}^j \geq 0 && \text{for } P_k \in \mathcal{P}, j \in J_{o_k}.
\end{aligned}
$$

Additional thoughts

Though we will not elaborate on how we might do so, none of our Mirrored Dantzig-Wolfe method depends on the subproblems being linear programs. In fact, the subproblems are independent minimization problems without capacity constraints, regardless of how they are formulated. So we could even let our subproblems be integer or non-linear programs, where they solve a single component of a packing problem with the capacity constraints removed. For our specific instance, we can avoid many of the problems we encountered due to our linear programs being relaxations by instead solving the integer programming version of the minimization problem at each subproblem. For us, this minimization problem is just a directed Steiner tree problem, for which there are many existing techniques.

In essence, what this suggests is that we may be able to solve symmetric pack-

ing problems by instead solving many instances of a pure minimization problem, and it is not hard to imagine how the minimization version of the problem may be easier.

CHAPTER 6

EVALUATIONS

We have validated our results through extensive experimentation. We begin by highlighting some of our results on AllGather. In what follows, the experiments discussed are for AllGather unless otherwise noted. Here, we report results on the DGX A100 topology (see [4, Fig. 7] for a depiction), where we use a multi-rail structure to increase the size of the topology beyond a single server [16].

Comparison against TE-CCL

We evaluate the computational performance and solution quality of our algorithm against TE-CCL [10] across varying cluster sizes. Our experiments demonstrate significant improvements in both scalability and latency, with our algorithm successfully solving problems that TE-CCL cannot handle at scale.

We compared our algorithm against TE-CCL's publicly available code configured with 1 and 2 chunks. The performance metrics include total generation time (time to compute the communication schedule), latency (execution time of the resulting schedule), and objective function value. Notably, TE-CCL's formulation cannot directly model memory behavior without modification, so we conducted comparisons on the topology with memory vertices removed. Additionally, while TE-CCL's published work focuses on heuristic methods for AllGather operations, we compare against their optimal solution method as the closest comparison to

our methods.

	Our Techniques			TE-CCL (1 ch.)			TE-CCL (2 ch.)		
GPUs	Time	Lat.	Obj.	Time	Lat.	Obj.	Time	Lat.	Obj.
8	2.95	0.021	109.7	3.15	0.031	112	6.23	0.031	112
16	27.6	0.045	504.9	22.7	0.24	530	91.1	0.24	530
32	82.2	0.12	2092	-	-	-	-	-	-
64	634	0.28	8600	-	-	-	-	-	-
128	9870	0.60	33640	-	-	-	-	-	-

Table 6.1: A comparison of time (s), latency (s/GB), and objective values for our techniques against TE-CCL on 1 and 2 chunks. Missing entries did not finish to optimality within a 5 hour time limit.

Table 6.1 presents the performance comparison between our techniques and TE-CCL across cluster sizes ranging from 8 to 128 GPUs. Our algorithm demonstrates superior scalability, successfully generating solutions for all tested configurations up to 128 GPUs, while TE-CCL fails to complete computation for any configuration larger than 16 GPUs. Furthermore, TE-CCL fails to generate solutions using more than 2 chunks for all but the smallest experiment size. Naturally, these results lose out on most of the benefits of pipelining, and cannot hope to compare to the quality of our solutions in terms of latency. The experimental results show exactly this phenomenon.

Our evaluation results demonstrate the power of our techniques: even for state-of-the-art servers consisting of heterogeneous processors (CPUs and GPUs), memory (classical DRAM as well as GPU memory), and peripheral interconnects (classical PCIe as well as inter-GPU NVLink interconnects), and even with prominently used large-scale network topologies, our techniques enable us to compute

optimal network communication algorithms within a couple of hours. As mentioned earlier, this has powerful implications beyond designing network communication algorithms for existing distributed ML/AI clusters. For example, the ability to quickly synthesize optimal algorithms for a given network topology enables relative comparison of different network topologies, an open problem in networking for ML/AI clusters [16]. As another example, the ability to quickly synthesize optimal algorithms for a given server hardware enables principled cost-benefit analysis of various hardware design points from a single vendor and across vendors (e.g., from NVIDIA, AMD, and many other startups building ML/AI compute and networking hardware).

Evaluating the benefit of symmetry.

To isolate the contribution of our novel Mirrored Dantzig-Wolfe technique, we compared our approach against a standard Dantzig-Wolfe implementation. This analysis demonstrates that our symmetry-exploiting modifications are not merely beneficial but essential for achieving computational tractability at scale.

Table 6.2 shows the generation time comparison between standard Dantzig-Wolfe decomposition and our Mirrored Dantzig-Wolfe approach across increasing cluster sizes. We note that the relative performance gain increases with the size of the cluster, demonstrating that the computational advantages of Mirrored Dantzig-Wolfe become more pronounced as the size of the system grows.

GPUs	Standard Dantzig-Wolfe	Mirrored Dantzig-Wolfe
8	60.5	2.95
16	2610	27.6
32	34700	82.2
64	-	634
128	-	9870

Table 6.2: Algorithm generation runtime (s) of standard Dantzig-Wolfe vs Mirrored Dantzig-Wolfe. Missing entries did not finish to optimality within a 10 hour time limit.

6.1 Additional results

We ran many more experiments than just those mentioned above. In this section, we present the results for many of our raw experiments before discussing some of the insights they may provide us.

We assume every cluster is made up of many identical servers (which we will refer to as nodes). These nodes are connected together in a multirail structure, which means that the first GPU from every node is connected to a common switch, the second GPU from every node is connected to a second common switch and so on.

We ran experiments for these topologies in three scenarios:

- Only using NVLink to communicate between GPUs
- Completely ignoring NVLink

- Using all links, including NVLink

On all of these configurations, we ran All-to-all and AllGather. For techniques other than our own, we did our best to adapt their methods for these topologies, though some liberties had to be made, and in some circumstances, we were unable to reconcile our requirements with their methods.

Also note that time and latency values are rounded to three significant figures, while objective values are rounded to four.

6.1.1 Our results

We will present the results as a series of tables, where the description of each table contains the information about them. Any entry left blank indicates that we were unable to get a result for the indicated parameters due to some physical limitation, usually either time or memory.

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	0.247	0.0233	112
2	16	19.3	0.0462	516.9
4	32	42.8	0.12	2176
8	64	254	0.28	8960
16	128	865	0.60	36350
32	256	5360	1.24	146400

Table 6.3: Our methods, AllGather, DGX A100 Topology, using only NVLink

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	0.257	0.0219	92
2	16	3.9	0.469	424
4	32	22	0.969	1808
8	64	360	1.97	7456
16	128	4200	3.97	30270
32	256	-	-	-

Table 6.4: Our methods, AllGather, DGX A100 Topology, using no NVLink

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	2.95	0.0211	109.7
2	16	27.7	0.0452	504.9
4	32	82.1	0.12	2092
8	64	634	0.28	8600
16	128	9870	0.60	33640
32	256	-	-	-

Table 6.5: Our methods, AllGather, DGX A100 Topology, using all links

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	0.854	0.0233	112
2	16	1.04	0.32	960
4	32	4.36	0.96	4864
8	64	21.6	2.24	21500
16	128	106	4.80	90110
32	256	633	9.92	368600

Table 6.6: Our methods, All-to-all, DGX A100 Topology, using only NVLink

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	0.198	0.375	112
2	16	0.902	0.75	736
4	32	4.58	1.5	3520
8	64	15.4	3.0	15230
16	128	97.1	6.0	63230
32	256	467	12.0	257500

Table 6.7: Our methods, All-to-all, DGX A100 Topology, using no NVLink

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	1.02	0.021	112
2	16	2.84	0.32	764.2
4	32	15.1	0.96	3689
8	64	75.1	2.24	16020
16	128	330	4.8	66610
32	256	1550	9.92	271500

Table 6.8: Our methods, All-to-all, DGX A100 Topology, using all links

6.1.2 ForestColl results

We ran an implementation of ForestColl [20] we received from the authors of the paper that is more efficient than the publicly available code and was specially written to take advantage of parallelization. In order to make it work, we needed to assume memory components behaved as switches, and ignored memory bandwidths.

Notably, ForestColl only produces results for three collectives: AllGather, ReduceScatter, and AllReduce. So here, we only present the results from AllGather.

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	0.294	0.0233	112
2	16	0.965	0.0462	512
4	32	2.10	0.12	2176
8	64	7.90	0.28	8960
16	128	39.2	0.60	36350
32	256	315	1.24	146400

Table 6.9: ForestColl, AllGather, DGX A100 Topology, using only NVLink

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	0.189	0.0219	112
2	16	0.670	0.469	512
4	32	1.69	0.969	2177
8	64	4.17	1.97	8960
16	128	19.4	3.97	36350
32	256	135	7.87	146400

Table 6.10: ForestColl, AllGather, DGX A100 Topology, using no NVLink

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	0.308	0.0211	112
2	16	1.31	0.0452	512
4	32	2.20	0.12	2176
8	64	10.2	0.28	8960
16	128	55.2	0.60	36350
32	256	419	1.24	146400

Table 6.11: ForestColl, AllGather, DGX A100 Topology, using all links

6.1.3 TE-CCL results

We ran our experiments for TE-CCL using the publicly available code on GitHub. As before with ForestColl, we needed to ignore memory bandwidth limitations,

and to treat memory like switches. Furthermore, as part of the input to TE-CCL for solving AllGather, we need to provide a number of chunks (as in CM2). Given how the performance worked out, we only ran experiments for one and two chunks. All-to-all doesn't rely on specifying a number of chunks.

TE-CCL offers multiple options for solving which need to be specified at the beginning. Since we wished to compare against other synthesizers that purported to produce optimal solutions, we used the setting to solve to optimality, and counted it as a failure if the integer program didn't solve. It is worth noting that TE-CCL also provides various heuristic methods based on their exact method that solve slightly faster (though in our experiments, the difference was minimal) to produce worse solutions, so in someways, we are being uncharitable towards them.

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	3.15	0.0313	112
2	16	22.74	0.24	530
4	32	-	-	-
8	64	-	-	-
16	128	-	-	-
32	256	-	-	-

Table 6.12: TE-CCL, 1-chunk, AllGather, DGX A100 Topology, using only NVLink

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	17.9	0.313	112
2	16	-	-	-
4	32	-	-	-
8	64	-	-	-
16	128	-	-	-
32	256	-	-	-

Table 6.13: TE-CCL, 1-chunk, AllGather, DGX A100 Topology, using no NVLink

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	4.89	0.0313	112
2	16	32.76	0.24	534
4	32	-	-	-
8	64	-	-	-
16	128	-	-	-
32	256	-	-	-

Table 6.14: TE-CCL, 1-chunk, AllGather, DGX A100 Topology, using all links

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	6.23	0.0313	112
2	16	91.1	0.16	522
4	32	-	-	-
8	64	-	-	-
16	128	-	-	-
32	256	-	-	-

Table 6.15: TE-CCL, 2-chunks, AllGather, DGX A100 Topology, using only NVLink

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	83.9	0.313	112
2	16	-	-	-
4	32	-	-	-
8	64	-	-	-
16	128	-	-	-
32	256	-	-	-

Table 6.16: TE-CCL, 2-chunks, AllGather, DGX A100 Topology, using no NVLink

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	9.16	0.0313	112
2	16	107.7	0.16	522
4	32	-	-	-
8	64	-	-	-
16	128	-	-	-
32	256	-	-	-

Table 6.17: TE-CCL, 2-chunks, AllGather, DGX A100 Topology, using all links

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	20.0	0.0625	112
2	16	302	0.44	912.6
4	32	3170	1.08	4463
8	64	-	-	-
16	128	-	-	-
32	256	-	-	-

Table 6.18: TE-CCL, All-to-all, DGX A100 Topology, using only NVLink

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	30.1	0.469	112
2	16	470	0.880	696.3
4	32	-	-	-
8	64	-	-	-
16	128	-	-	-
32	256	-	-	-

Table 6.19: TE-CCL, All-to-all, DGX A100 Topology, using no NVLink

Nodes	GPUs	Time (s)	Latency (s/GB)	Objective
1	8	55.9	0.0625	112
2	16	810	0.52	800.9
4	32	8760	1.16	4313
8	64	-	-	-
16	128	-	-	-
32	256	-	-	-

Table 6.20: TE-CCL, All-to-all, DGX A100 Topology, using all links

6.1.4 Discussion

We will now summarize some points we should take away from the mass of tables presented above. Compared to our techniques, ForestColl nearly matches the quality of our solutions, and typically generates them an order of magnitude more quickly. Some of this speed difference can be attributed to the extra effort spent optimizing their code to allow more efficient computation, but in the instances where it works, their techniques are very impressive. However, their methods are not universal, and still require making heavy assumptions about the input topology. They were able to get away with it in this instance because memory band-

width was not a large consideration, but it is not so difficult to construct instances (as we did in Chapter 2) where this could become problematic.

In comparison, TE-CCL does much worse. Their integer programs just solve too slowly to scale anywhere near as well as our methods, and the quality of their results is greatly hindered by their limitation on chunks.

We also tried to run experiments on TACCL [13], but due to the way they encode switches, it was impossible to run on an input that explicitly represented memory, even if we wished to treat it as a switch. TACCL also suffers from the same problem as TE-CCL, in that we must specify some number of chunks ahead of time. So even in the one configuration without memory (using only NVLink), we would not expect to generate competitive solutions.

CHAPTER 7

CONCLUSIONS

This dissertation has focused on optimizing collective communication patterns in large-scale machine learning clusters, addressing an important concern for increasingly distributed training procedures. Using standard linear programming concepts combined with novel algorithmic techniques, we developed a framework that achieves provably optimal communication algorithms while maintaining computational tractability at large scales.

Our theoretical contributions were presented in three chapters: Chapter 3 provided linear programming formulations that capture the unique properties of collective communication, Chapter 4 presented techniques for converting between steady-state solutions and implementable discrete algorithms, and Chapter 5 introduced a technique we called Mirrored Dantzig-Wolfe, which exploits network symmetry for considerable empirical speedups. We validated our results with a wide range of experiments in Chapter 6.

We now conclude by including some speculation about potential implications of this work.

Design of distributed architectures

One significant implication beyond optimizing existing systems is the possibility to use these methods to assist in designing future distributed architectures. Since we can easily determine the performance of various collective algorithms on any proposed topology, we can determine the benefits of a topology without relying on empirical benchmarks.

Beyond the design of new topologies, our formulations naturally allow us to explore sensitivity to bandwidth constraints on different link sets. By solving with varying bandwidth parameters, system designers can identify the links that most affect communication performance. This could allow bandwidth tuning and assist with answering questions like whether investing in higher GPU-to-memory bandwidth would be more beneficial compared to improving inter-GPU connectivity.

Multiple collectives

A key strength of our approach lies in its natural extension to multiple simultaneous collective operations. The secondary objective optimization we developed proves particularly useful in this situation. By minimizing total data upload and download volumes across GPUs, our secondary objective reduces contention for shared resources, enabling multiple collectives to execute concurrently without performance degradation, especially compared to other techniques that remain

oblivious to bandwidth utilization on links not used at full capacity.

Not only can we extend to the setting of multiple collectives by layering solutions on top of each other, the flexibility of our model means that we are also able to fully encode multiple collectives within a single formulation and use our techniques to find an optimal solution for the combination of collectives.

Concluding Remarks

This dissertation has developed a comprehensive framework for synthesizing collective communication algorithms that addresses important performance challenges in distributed systems. The techniques we introduced have enabled practical optimization at a better combination of flexibility and scalability than other existing methods. As distributed machine learning systems continue to grow in scale and complexity, systematic approaches to communication optimization will become increasingly important, and the mathematical foundations established here contribute to this ongoing effort by providing both theoretical insights and practical algorithms for efficiently utilizing computational resources.

APPENDIX A

SIMPLE BOUNDS ON COLLECTIVES

In this section, we will argue some simple lower bounds that may be useful in establishing the optimality of solutions without relying on solving any linear programs. For this section, let b_{up} be a universal GPU upload bandwidth limit, which restricts the total data transmitted from any GPU to any other component, and similarly, b_{down} limiting the amount of data downloaded onto (received by) any GPU from any other component. Note arbitrary topologies need not have identical GPU upload bandwidths, but every GPU should have *some* bound on data uploaded or downloaded, and we can take the most permissive bounds to be the universal limits.

We now argue lower bounds for collective operations in terms of b_{up} and b_{down}. Recall that we assume the quantity of data we need to transmit is 1 unit.

Broadcast

Theorem 8. *Any Broadcast algorithm requires 1 unit of data uploaded from the root GPU and 1 unit of data downloaded to all other GPUs.*

Proof. In any Broadcast algorithm, at least 1 unit of data must leave the root GPU. All other GPUs must receive at least 1 unit of data. □

Theorem 9. *Any Broadcast algorithm will have latency at least* $1/\min\{b_{up}, b_{down}\}$

Proof. By Theorem 8, at least 1 unit of data is uploaded from the root GPU, which requires time at least $1/b_{up}$. Similarly, other GPUs download 1 unit of data, which requires time at least $1/b_{down}$. Combining these gives our bound. □

AllGather

Theorem 10. *Any AllGather algorithm requires at least 1 unit of data uploaded per GPU and at least* $(n-1)$ *data downloaded per GPU.*

Proof. In any AllGather algorithm, each GPU must receive D data from every other GPU. Which is $(n-1)$ data downloaded. And at least 1 unit of data from each GPU needs to reach the other GPUs, which requires 1 unit of data uploaded. □

Theorem 11. *Any AllGather algorithm will have latency at least* $(n-1)/b_{down}$

Proof. By Theorem 10, each GPU must download $(n-1)$ data. Due to the bandwidth limit, this requires latency at least $(n-1)/b_{down}$. □

Reduce

Theorem 12. *Any Reduce algorithm requires at least 1 unit of data downloaded to the root GPU and at least 1 unit of data uploaded from all other GPUs.*

Proof. In any Reduce algorithm, the data from each GPU must leave it somehow, which is 1 unit of data that must be uploaded. And by definition, at least 1 unit of data must arrive at the root GPU. □

Theorem 13. *Any Reduce algorithm will have latency at least* $1/\min\{b_{up}, b_{down}\}$.

Proof. By Theorem 12, all GPUs other than the root must upload at least 1 unit data, which requires time at least $1/b_{up}$. Similarly, the root GPU must download at least 1 unit of data, which requires time at least $1/b_{down}$. Combining these gives our bound. □

ReduceScatter

Note that we do not require all the data to be uploaded, because some pieces of data only need to stay on the same GPU. We use *input* to refer to the data that begins at a GPU. And we will say a piece of data is an h-reduction if it is the result of the reduction operation on h inputs. So if no reductions have happened to a piece of data, it is a 1-reduction, and if the data is the result of reducing all of the inputs, it is an n-reduction.

We say that a GPU g has *acquired* an input if either that data is on g, or if the result of reduction operations involving the input is on g.

Lemma 14. *Any algorithm in which a GPU* g *acquires* n *indivisible inputs requires at least* $n - 1$ *downloads to GPUs.*

Proof. We prove this by strong induction. Naturally, for the base case, if g has acquired one input, it requires no downloads. (The one piece of data can be the one beginning on g.)

For an inductive hypothesis, suppose that an algorithm in which g acquires ℓ indivisible inputs requires at least $\ell - 1$ downloads for $1 \leq \ell \leq h$.

Then consider an algorithm in which g acquires $(h+1)$ inputs. Say the last piece of data downloaded to g is a s-reduction. Then we have $h + 1 - s$ data already on g. By our inductive hypothesis we require at least $h - s$ downloads for all the data to arrive at g. And s-reduction in the last download must be the result of a reduction operation at some GPU g', so g' must have acquired s inputs. Again by the inductive hypothesis, this data required at least $s - 1$ downloads. So in total, the two pieces, along with the final download to g is at least $h - s + s - 1 + 1 = h$ downloads. □

Lemma 15. *Any algorithm that produces a unit-sized n-reduction requires at least $(n-1)$ data downloaded to GPUs.*

Proof. If an algorithm produces a n-reduction of indivisible inputs on GPU g, then g will have acquired n indivisible inputs. So by Lemma 14, this requires $n - 1$ downloads.

So consider any algorithm that produces a unit-sized n-reduction where the inputs are divisible into k subdivisions of size $1/k$. Then each subdivision requires $(n - 1)/k$ data downloaded, which totals to $(n - 1)$ data downloaded to

GPUs. By making k arbitrarily large, we can conclude that this is true even with no divisibility restrictions. □

Theorem 16. *Any ReduceScatter algorithm requires at least $(n-1)$ total data downloaded summed over all GPUs and at least $(n-1)/n$ total data uploaded per GPU.*

Proof. For a ReduceScatter algorithm, every final piece of data is a n-reduction, each of size $1/n$. So by Lemma 15, each n-reduction requires $(n-1)/n$ total data downloaded by some GPU. So summing over all GPUS, the total amount of data downloaded is at least $(n-1)$. And each GPU has $(n-1)/n$ data that must leave and end up on another GPU. So this must all be uploaded. □

Theorem 17. *Any ReduceScatter algorithm will have latency at least $\frac{(n-1)}{n \min\{b_{up}, b_{down}\}}$.*

Proof. By Theorem 16, at least $(n-1)/n$ data is uploaded at one of the GPUs, and at least $(n-1)/n$ data is downloaded at one of the GPUs.

Using the limiting bandwidth for each gives bounds of $\frac{(n-1)}{nb_{up}}$ and $\frac{(n-1)}{nb_{down}}$. Taking the better of the two yields the claimed bound. □

AllReduce

Theorem 18. *To create n copies (across n GPUs) of a piece of data of unit size that is the reduction of n inputs requires at least $(2n-2)$ data downloaded (across all GPUs).*

Proof. In particular, we will argue that any algorithm that creates ℓ identical n-reductions of indivisible inputs requires $n + \ell - 2$ downloads to GPUs. Again, we will prove this by induction on the number of identical copies.

To begin with, when $\ell = 1$, the claim follows from Lemma 14.

Say that for $\ell - 1$ copies of a n-reduction, we require $n + \ell - 3$ downloads to GPUs. And suppose there is an algorithm that produces ℓ copies of a n-reduction, with no more than $n + \ell - 3$ downloads. Consider such an algorithm with minimum total downloads and take a last download (say to GPU g) in the sense that no download or upload from g occurs after this download. This download can only benefit the resulting data on g. So if we removed this download from the algorithm, $\ell - 1$ GPUs should still end with a copy of the n-reduction. But that would mean that this algorithm could produce $\ell - 1$ n-reductions on $\ell - 1$ GPUs with $n + \ell - 4$ downloads, which contradicts our inductive hypothesis. So indeed, we need $n + \ell - 2$ downloads for ℓ copies of a n-reduction.

Therefore, we require $2n - 2$ downloads for n copies of a n-reduction of indivisible data.

So consider any algorithm that produces n copies of a n-reduction of unit size where the inputs are divisible into k subdivisions of size $1/k$. Then the n copies of the n-reduction of each subdivision requires $(2n - 2)/k$ data downloaded, which totals to $(2n - 2)$ data downloaded to GPUs. As before, since we can make k arbitrarily large, we can conclude that this is true even with no divisibility restric-

tions. □

Theorem 19. *Any AllReduce algorithm will have latency at least* $\frac{(2n-2)}{nb_{down}}$.

Proof. By Theorem 18, there is some GPU where at least $\frac{(2n-2)}{n}$ data is downloaded. Divide by the bandwidth to get the limit on latency. □

BIBLIOGRAPHY

[1] Ravindra A. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[2] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 62–75, 2021.

[3] Jiamin Cao, Yu Guan, Kun Qian, Jiaqi Gao, Wencong Xiao, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. Crux: Gpu-efficient communication scheduling for deep learning training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 1–15, 2024.

[4] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[5] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[6] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 57–70, 2024.

[7] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[8] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. {MegaScale}:

Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

[9] Heehoon Kim, Junyeol Ryu, and Jaejin Lee. Tccl: Discovering better communication paths for pcie gpu clusters. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 999–1015, 2024.

[10] 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*, pages 16–37, 2024.

[11] NVIDIA. NCCL. https://github.com/NVIDIA/nccl, 2025.

[12] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. Scaling distributed machine learning with {In-Network} aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808, 2021.

[13] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. {TACCL}: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 593–612, 2023.

[14] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[15] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. *Proceedings of Machine Learning and Systems*, 2:172–186, 2020.

[16] Weiyang Wang, Manya Ghobadi, Kayvon Shakeri, Ying Zhang, and Naader Hasani. Rail-only: A low-cost high-performance network for training llms with trillion parameters. In *2024 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 1–10. IEEE, 2024.

[17] Herbert S Wilf. *generatingfunctionology*. CRC press, 2005.

[18] William Won, Midhilesh Elavazhagan, Sudarshan Srinivasan, Swati Gupta, and Tushar Krishna. Tacos: Topology-aware collective algorithm synthesizer for distributed machine learning. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 856–870. IEEE, 2024.

[19] Yongji Wu, Yechen Xu, Jingrong Chen, Zhaodong Wang, Ying Zhang, Matthew Lentz, and Danyang Zhuo. Mccs: A service-based approach to collective communication for multi-tenant cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 679–690, 2024.

[20] Liangyu Zhao, Saeed Maleki, Aashaka Shah, Ziyue Yang, Hossein Pourreza, and Arvind Krishnamurthy. Forestcoll: Efficient collective communications on heterogeneous network fabrics. *arXiv preprint arXiv:2402.06787*, 2024.