

Rethinking Machine Learning Collective Communication as a Multi-Commodity Flow Problem

Behnaz Arzani*
Microsoft Research
bearzani@microsoft.com

Siva Kesava Reddy Kakarla*
Microsoft Research
sivakakarla@microsoft.com

Miguel Castro
Microsoft
mcastro@microsoft.com

Srikanth Kandula
Microsoft Research
Srikanth@microsoft.com

Saeed Maleki
Microsoft Research
saemal@microsoft.com

Luke Marshall
Microsoft Research
luke.marshall@microsoft.com

Abstract

We show communication schedulers’ recent work proposed for ML collectives does not scale to the increasing problem sizes that arise from training larger models. These works also often produce suboptimal schedules. We make a connection with similar problems in traffic engineering and propose a new method, TE-CCL, that finds better quality schedules (e.g., finishes collectives faster and/or while sending fewer bytes) and does so more quickly on larger topologies. We present results on many different GPU topologies that show substantial improvement over the state-of-the-art.

1 Introduction

Near-optimal collective communication optimizers [5, 27, 29] — that optimize the communication routes and schedules of distributed training — cannot scale to what cloud operators need. This is because cloud operators run large multi-tenant GPU clusters where they schedule distributed training jobs over many GPUs. Tools that find optimum topologies, hardware architectures, or co-optimize various aspects of distributed training [19, 30, 31] also rely on these optimizers and call them multiple times during their search.

Without communication optimizers GPU clusters spend a significant amount of time with idle GPUs: prior work reports the GPUs in BERT [8] and DeepLight [7] spent 11% and 63% of the time idle respectively [27]. The problem becomes worse as we move to faster GPUs. Current communication optimizers leave significant room for improvement: for example, we show we can improve upon state-of-the-art solutions such as TACCL [27] by over 2× on its two chassis NDv2 topology [2] (Figure 11).

We scale near-optimal collective communication optimizers (e.g., SCCL [5]) — that model the problem imperfectly but optimally solve their model — to enable cloud operators to use them for today’s large GPU collectives and improve their runtime to make them more usable as part of other collective optimizers such as [19, 30, 31] — our goal is to

improve the solution quality of state-of-the-art *heuristics* (e.g., TACCL [27]) while maintaining the same ability to scale.

The input to a collective communication optimizer is a *demand* (e.g., ALLTOALL, ALLGATHER, ALLREDUCE): a set of interconnected GPUs where each GPU has a certain amount of data to send to other GPUs in the interconnect. The goal of the optimizer is to produce routes and schedules that either maximize bandwidth utilization [29] or minimize job completion time [5, 27] for the input demand or both.

Near-optimal optimizers (e.g., [5]) apply to a single chassis [5]. In contrast, operators require solutions that scale to topologies with 30-60 chassis (and project larger topologies) [6]. Heuristic scale but often produce highly sub-optimal solutions [27, 29]. This is becoming a problem as topologies grow and more users share the same underlying network.

SCCL cannot scale because it uses SMT solvers [28]. The heuristics avoid using SMT solvers and scale better but fail to account for one or more factors (e.g., identifying where traffic should be copied inside the network, enforcing synchronization barriers, and properly accounting for latency of transfers) and produce sub-optimal solutions as a result.

We propose an alternate solution: TE-CCL. Our insight is that we can model the problem of collective communication optimization through techniques from a class of problems known as multi-commodity flow.

Operators use multi-commodity flow problems in traffic engineering (TE) and use flow conservation constraints to model the flow of traffic — they assign paths to maximize a cost function [3]. They, too, take a set of demands as input and produce routes and schedules that optimize various objectives. But the collective problem has nuances that are not present in a traditional multi-commodity flow model:

Temporal variations. Multi-commodity flow problems assume “sustained demand”: such problems rely on a continuous flow of data between a source and destination (for several minutes), and this is why the demand in these problems is a bandwidth request (with units such as bits/sec). But GPUs in a collective have finite data to send — the demand in these problems is a transfer request (with units such as bits).

*both authors contributed equally to the paper

This means we can no longer minimize the delay on the longest path to minimize the transfer time as traditional flow problems do: we can no longer assume an uninterrupted flow of traffic to approximate the delay cost of transfers (see § 2).

Support for store and forward. Traditional flow problems [3] do not model caches. We show in § 6 that we can speed up the solver to find schedules faster if we use the available memory in GPUs.

Supporting copy. Unlike typical use-cases of the network flow formulation (*e.g.*, in the TE context [14, 16]), collective communication often multicasts the same data to multiple parties, which requires the model to appropriately copy data within the network (and adjust the traditional flow conservation constraints accordingly).

Some prior works do extend multi-commodity flow problems to incorporate these concerns: *e.g.*, Calendaring [17] supports deadlines on fixed-size transfers, NetStitcher [18] allows for store-and-forward, and several multicast TE works [10, 22] support copying (see § 7). But, it is non-trivial to combine these techniques to add support for all three dimensions *simultaneously* without affecting scalability.

We adapt multi-commodity flow problems to model all three behaviors and solve the general collective communication optimization problem. Our solution is a scalable mixed-integer linear program with optimality gap guarantees (based on the primal-dual theorem [3]). We show that this solution scales to much larger collectives than techniques such as TACCL [27] and SCCL [5] and improves the solution quality.

For certain collectives we can scale this solution even further by converting the MILP into an LP by removing all integer variables. In the general case, we improve scalability by partitioning the problem in time — a technique inspired by the A^* [12] from robotics.

TE-CCL’s solutions match the solution quality of SCCL and outperform the quality of state-of-the-art solutions such as TACCL [27] — we show a *minimum of 2×* performance improvement on the same 2 chassis NDv2 topology TACCL uses — and shortest path schedules [31] because the optimization models the end-to-end problem (whereas these works contain consecutive optimizations that only see a partial view of the problem at each stage), and adds support for copy and store-and-forward. As part of TE-CCL we are also able to account for multi-tenant, heterogeneous topologies where links have different latency, and bandwidth costs and tenants have different priorities to support cloud-scale GPU clusters better.

Our contributions are as follows:

- We present a novel, scalable, solution to the collective communication optimization problem. To the best of our knowledge, this is the first multi-commodity based solution to this problem. This new mode of thinking provides an opportunity to improve other aspects of

machine learning collectives such as topology design and adapting to failures.

- We show how to scale this solution to larger topologies through a linear program for ALLTOALL-like demands and a technique inspired by A^* in the general case.
- We evaluate TE-CCL both on popular topologies and on the proprietary, large-scale topologies from a large public cloud. We show our solution improves the solution quality of TACCL [27] by a minimum of 2× in many scenarios. We find TACCL’s heuristic is unreliable (produces different solutions in each run) and cannot find a feasible solution in many cases. In contrast, TE-CCL is reliable, produces the same solution in each run, and finds a feasible solution in instances where TACCL was infeasible. TE-CCL and TE-CCL have similar abilities to scale although TE-CCL was able to run on much larger topologies.

2 Background and Motivation

We present the necessary background on collective communication and motivate the need for scalable communication schedules for ML collectives. We then describe the multi-commodity flow formulation, how it relates to collective communication optimization, and show why we should modify them to model delay, store-and-forward, and copy.

2.1 The need for fast collective scheduling

ML collectives have pronounced communication patterns with flavors of multicast aggregation trees: *e.g.*, ALLGATHER, ALLTOALL, SCATTERGATHER (Figure 2 in TACCL [27] illustrates these communication patterns and how they differ).

These communication patterns constitute a *demand* on the network where each GPU wants to send data to other GPUs. For example, in an ALLGATHER demand, each source GPU intends to send all of its data to all other GPUs, and in an ALLTOALL demand, each GPU wants to send data to all other GPUs, but the data it sends to each GPU is different.

Collective communication optimizers take these demands as input and find solutions that route and schedule them efficiently to minimize transfer time. Operators use these optimizers in their multi-tenant GPU clusters and as part of solutions that help improve their offerings [19, 30, 31].

Most optimizers use the $\alpha - \beta$ cost model [13]. β is the transmission time of bytes on a link (how long it takes for the NIC to get the bytes on the wire): if we send \mathcal{B} bytes on a link with capacity C bytes per second, it takes $\frac{\mathcal{B}}{C}$ seconds for the bytes to cross that link and $\beta = \frac{1}{C}$. α is the constant delay of a link. In its simplest form, we can think of it as the propagation delay over a link, but it can also include other factors such as the fixed compute cost of consolidating the data and making the call to the network stack to transmit it. It takes $\alpha + \beta S$ seconds to send a chunk of size S over a link.

Most existing optimizers fail to scale to large topologies (e.g., SCCL [5]) or produce sub-optimal schedules (e.g., NCCL [5, 31], TACCL [27]). SCCL uses SMT solvers and does not scale. TACCL separates the routing and scheduling problems and fails to co-optimize the two. The shortest path first algorithm in [31] fails to leverage copy.

2.2 Background on network flow solutions

Many works find optimal routes for wide area traffic engineering (WAN-TE) and for multicast networks (e.g., [1, 9, 10, 14, 16–18, 21, 22]). These problems also take as input a set of *demands*: “rate requests” between a source-destination pair. The solutions aim to meet these demands and maximize the total flow the network carries, or the network utilization, or maintain fairness without violating capacity constraints.

Although these formulations take different forms (most notable of these is the path-formulation which takes a set of paths as input and only allows flows to go over the input paths [14, 16]) they share the following key components:

Capacity constraints. Ensure that the traffic the solution allocates on a link never exceeds its capacity.

Flow conservation constraints. Ensure that the solution does not create traffic “out of thin air” but that each non-source node forwards what it receives or consumes it.

An objective. The objective encodes what the optimization is trying to minimize or maximize: the cost model. The most common TE objectives include max-min fair rate allocations, total satisfied demand, or the link utilization.

We observe that the multi-commodity flow and the collective communication optimization problems have many commonalities: both take a set of demands and a topology as input and produce routes (and schedules) to optimize an objective.

But the two are different as the collective optimizer requires we account for: copy, store-and-forward, and temporal behavior (and the impact on the latency cost as a result). We next discuss each of these in detail:

Temporal behavior. In the collective problem, the source wants to transfer a fixed number of bits — once the network satisfies the demand from that source, the demand goes to zero and frees up capacity. The network can then re-assign this capacity to other demands. This happens in the traditional TE as well but at larger time-scales and most deployed TE solvers periodically re-solve the optimization to handle it. This is not a problem at face-value — after all we solve the problem offline — but it impacts the scalability of the solution in the collective setting. Calendaring [17] and Netstitcher [18] both model this, but they do not model propagation delay and hence fail to address an important side-effect:

Modeling delay (the α -cost). Most TE solutions (e.g., [10, 22]) compute the delay-cost as the maximum delay across all paths where the delay of a path is the sum of the delay on each of its

links. These models assume the total time needed to fulfill a demand is the transmission delay (or β -cost) + this delay-cost.

We show why this model breaks through an example (Figure 1a). Here, two sources (s_1 and s_2) want to send a unit of traffic (■ and □) to destination d . The links on the path from s_1 to h_3 have a propagation delay α_1 and those on s_2 to h_3 have a propagation delay of α_2 where $\alpha_2 = 2\beta + 3\alpha_1$. If we take the traditional TE approach to model the delay, the path with the maximum delay is the one between S_2 and d which has a propagation delay of α_2 . It also takes an additional 4β for the traffic to get from both S_1 and S_2 to d : the TE solutions estimate $\alpha_2 + 4\beta$ as the completion time.

But because of the higher propagation delay on the link s_2 - h_3 the data from s_1 and s_2 both arrive at h_3 at the same time ($t = \beta + \alpha_2$), since the propagation delay on the link h_3 - d is zero, the total time to complete the transfer is $\alpha_2 + 3\beta$.

The impact of α is greater for smaller transfers (Figure 2): the error in our estimate of algorithm bandwidth for a schedule where we do not model α to one where we do goes up to $100\times$. **Store-and-forward.** Most nodes in a collective topology can buffer incoming traffic before sending it out. We can use this to improve solver time (Figure 1b) as the number (space) of optimal solutions increases. In Figure 1b, without store and forward, in the first second, any two nodes (3 schedules) can send their chunks to h . With store and forward, we can have three additional schedules where all three sources send to h in the first second, and we then choose in which order to send them to the destination in the next. The solution quality is the same in both cases (we satisfy the demand in 3s). We confirmed this in our experiments in § 6.3 across all the scenarios we considered. For some collective demands store-and-forward may also help with transfer time (though it did not in our experiments).

But traditional TE does not model buffering [14, 16]. Net-Stitcher [18] models store and forward but assumes flows do not compete for bandwidth and solves a separate optimization for each flow: it is sub-optimal and does not scale. Some multi-cast TE solutions model intermediate caches [10], but they fail to account for the delay, and it is difficult to modify them to do so.

Copy. Some collective demands (e.g., ALLGATHER) consist of sources that send the same data to multiple destinations (i.e., multicast). Traditional TE does not model copy (e.g., SWAN and B4 [14, 16]) and produces sub-optimal solutions (see Figure 1c). Multi-cast TE [10, 22] addresses this problem but fails to model delay (these works assume sustained demands) and, in some instances [22], store-and-forward.

We formulate the collective communication optimization problem as a TE problem that supports these elements. The challenge is to maintain scalability. We show our model, as-is, outperforms current state-of-the-art solutions such as

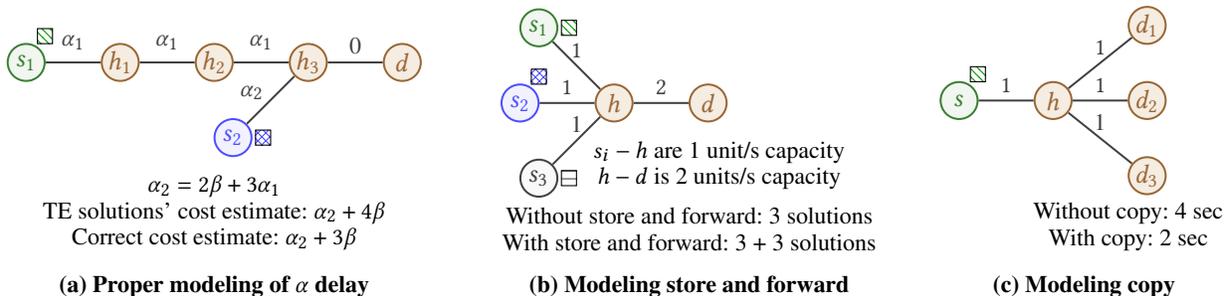


Figure 1: Examples that show why we should model properly: (a) α -delay: the maximum delay across all the paths is an incorrect estimate; (b) store-and-forward: buffers improve the solver time as there are more solutions (c) Copy: we can leverage copy to use the available bandwidth more efficiently.

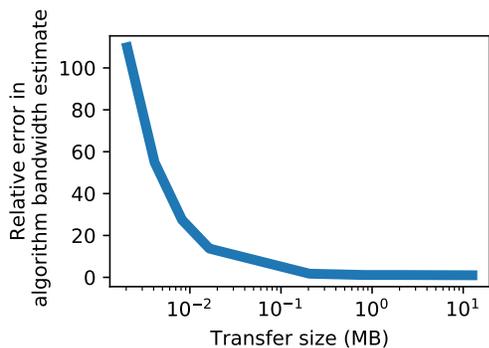


Figure 2: The relative error in the algorithm bandwidth estimate (the output buffer size / transmission time) of a collective schedule that does not model alpha compared to one that does. We use a proprietary topology from a public cloud with 2 chassis, 8 GPUs, and 40 edges where the α of inter-GPU and the GPU to the switch links is 0.6 and 0.75 microseconds respectively.

SCCL [5] in its ability to scale and TACCL [27] in its solution quality. We further improve its scalability through a technique inspired by A^* from robotics.

3 Solution

We next describe how we model the collective communication problem as a multi-commodity flow problem. We build on the ideas in Calendaring [17] and Netstitcher [18] to model delay, model store-and-forward, and copy.

But this solution does not scale to topologies with more than 64 GPUs. We scale it by changing our mixed integer program (MILP) into a linear program (LP) for demands such as ALLTOALL where sources send different data to each destination and do not benefit from copy (§ 4.1); and through a more general solution we call A^* (Appendix D).

3.1 The general model

We describe our notation in Table 1. Like any other multi-commodity flow problem we need to specify: capacity and flow conservation constraints, and an objective.

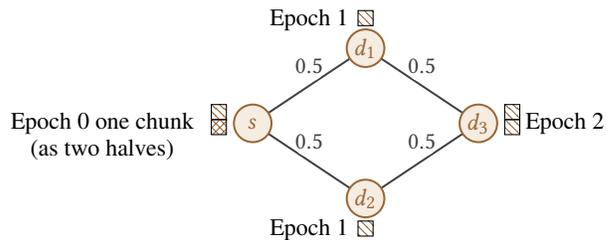


Figure 3: An example of why we need integer variables to track each chunk. If we allow partial chunks (⊠, ⊞) and copy at the same time, we run into a situation where the optimization can send the same copy of part of a chunk (⊞) to two neighboring nodes (in this case d_1 and d_2) and they can forward it along to the destination (d_3). Since the formulation has no way of knowing these two halves are the same, it thinks d_3 has received the full chunk.

But, to model delay, store-and-forward, and copy we need to introduce a few new concepts: chunks, epochs, and buffers.

Our notion of chunks is similar to prior work (e.g., SCCL): a chunk (like packets) is a block of bytes¹.

We use epochs (similar to how SCCL uses rounds) to make time discrete: epochs are fixed periods of time — our solution produces a schedule that tells the user in which epoch they should send a chunk and on which link.

We discuss chunk sizes and epoch durations in detail in § 5. For now, we assume τ is the epoch duration and T_{ij} is the capacity of a link (where the units are chunks per second), and that epoch is sufficient for at least one chunk to traverse any link.

We use buffers to model store-and-forward. To simplify the explanation we assume each node has enough buffer to store the entire network demand if it needs to (we show how to remove this assumption in Appendix B).

To model copy, we need to track each chunk: we use $F_{s,i,j,k,c}$ and $B_{s,i,k,c}$ to track whether chunk c from source s is going over link (i, j) or is in node i 's buffer at epoch k respectively.

¹We allow our solution to split chunks into smaller blocks when we move to the linear program form.

Variable	Description
----------	-------------

N	Set of nodes in the graph
S	Set of nodes in the graph that are switches ($S \subset N$)
E	Set of edges in the graph ($E \subseteq 2^{N \times N}$). Edges are unidirectional.
C	Chunk IDs ($C = \{0, 1, 2, \dots, C\}$). Each node has $\leq C + 1$ number of chunks.
D	Demand function ($N \times C \times N \rightarrow \{0, 1\}$) where $D_{s,c,d}$ is whether destination d wants chunk with id c from node s
τ	Epoch duration
K	The set of epochs ($K = \{0, 1, 2, \dots, K\}$)
$F_{s,i,j,k,(c)}$	Amount of source s chunks that are going over link $(i, j) \in E$ at epoch $k \in K$
$B_{s,i,k,(c)}$	Amount of source s chunks that are in node i 's buffer at the <i>start</i> of epoch k
T_{ij}	Capacity of link $(i, j) \in E$
α_{ij}	Fixed latency associated with link $(i, j) \in E$
δ_{ij}	Number of epochs contained within an α_{ij} for each link $(i, j) \in E$
$R_{s,d,k}$	Source s chunks that node d read off of the network in epoch k
$\mathcal{R}_{s,d,k,(c)}$	Source s chunks read off the network by d up to epoch k .

Table 1: Our notation. We put in parentheses the index (c) because we only use it when demands benefit from copy. When we model copy the values of F and B are integers. We show for some demands (where copy is not useful) we can use real variables instead in § 4.1.

We need to use integer variables for $F_{s,i,j,k,c}$ and $B_{s,i,k,c}$ to model copy — we cannot allow chunks to be split into smaller pieces. We use the example in Figure 3 to explain why. Source s sends the first half of a chunk (⊞) to both destinations d_1 and d_2 . These nodes then both forward it to d_3 : they have no way of knowing this is the same half. The optimization now thinks it has delivered the full chunk to d_3 while it has only delivered one half of it twice: it will send the second half of the chunk to both d_1 and d_2 but not to d_3 . Using integers for $F_{s,i,j,k,c}$ and $B_{s,i,k,c}$ allows us to avoid this problem (we do not need this for demands that do not benefit from copy § 4.1). We can increase the number of chunks to decrease the size of each individual chunk and support smaller transmission blocks (the optimization automatically consolidates them to bigger transmission units if needed) — but this increases the number of variables and slows down the optimization.

We now have everything we need:

Capacity constraints. Capacity constraints ensure we do not send more data than the link can carry in an epoch. We have:

$$\text{Capacity Constraint}(i, j, k) \triangleq \sum_{s \in N} \sum_{c \in C} F_{s,i,j,k,c} \leq T_{ij} \tau$$

Flow conservation constraints. The purpose of these constraints is to ensure the network does not create or lose traffic. The traditional form of these constraints specifies: a node should either consume or forward all of the traffic it receives. Here, we need to change these constraints to account for: (a) copy — nodes can create new traffic; (b) delay.

To model delay, we need to ensure a node does not forward a chunk if it has not received it. We first compute $\delta_{ij} = \frac{\alpha_{ij}}{\tau}$:

number of epochs it takes for a chunk to traverse a link. Traffic that node i sends to node j at the beginning of epoch k arrives at node j by the end of epoch $k + \lceil \delta_{ij} \rceil$. Node j can forward a chunk it receives from node i if node i sent it $\lceil \delta_{ij} \rceil$ ago.

Copy, by definition, violates traditional flow conservation constraints: it creates traffic where it didn't exist before. But, the node does not need to copy the chunk on the same link in the same epoch. We use this, along with δ_{ij} to rewrite the flow conservation constraints as follows:

Flow conservation constraints(s, n, k, c) \triangleq

$$B_{s,n,k,c} + \sum_{\forall j | (j,n) \in E} F_{s,j,n,k-\lceil \delta_{jn} \rceil, c} \geq \max_{\forall j | (n,j) \in E} F_{s,n,j,k+1,c}$$

This constraint encodes that what the node n has in its buffer along with what it receives in epoch k has to be larger than what it sends out in the next epoch on *each* of its outgoing links. We track the buffer contents as follows:

Buffer constraints(s, n, k, c) \triangleq

$$B_{s,n,k,c} = B_{s,n,k-1,c} + \sum_{\forall j | (j,n) \in E} F_{s,j,n,k-\lceil \delta_{jn} \rceil-1,c}$$

The buffers accumulate all traffic the GPU has received up to that point. Nodes have enough memory for this: for collective demands such as ALLGATHER each GPU needs all the chunks that are sent over the network and stores them anyway. But it is straight-forward to model limited buffers as well if we track what we should remove from the buffer in each epoch (see Appendix B). We evaluate the benefit of buffers using an ALLGATHER demand in § 6.

The first and last epoch’s flow conservation constraints are slightly different from the above: a node does not receive anything in the first epoch and doesn’t send anything in the last. We refer the reader to the appendix for details due to space constraints (see [Appendix A](#)).

We next need to account for demands: we need to make sure all demands are met at the end.

Destination constraints. These constraints ensure each node receives its full demand by the end:

$$\begin{aligned} \text{Destination constraints}(s, d, k, c) &\triangleq \\ \mathcal{R}_{s,d,k,c} &= \min(D_{s,d,c}, B_{s,d,k+1,c}) \quad \& \\ \mathcal{R}_{s,d,K,c} &= D_{s,d,c} \end{aligned}$$

where $\mathcal{R}_{s,d,k,c}$ is whether d has received chunk c of source s by epoch k . These destination constraints are different from their counterparts in traditional TE models. This is because of copy: d may want a chunk and also relay the chunk to others. Hence, we cannot assume d wants to consume everything in its buffers. This is why we take the minimum of $D_{s,d,c}$ and $B_{s,d,k+1,c}$. We ensure d eventually receives its full demand by the last epoch K by setting $\mathcal{R}_{s,d,K,c}$ to $D_{s,d,c}$.

Modeling switches. So far, we have only modeled the behavior of GPU nodes. While some topologies (*e.g.*, within a single DGX1 node [5]) only consist of GPUs, almost all larger topologies use switches to connect GPU blocks. We have to model switches differently because they have limited memory: we cannot buffer chunks at the switch. Hence, we set the buffer at each switch to zero.

Traffic needs to pay the α delay cost of two links to cross a switch: one from the node to the switch and one from the switch to the node.

Most of today’s switches support copy [9], and so we model switches with this assumption (switches have the same flow conservation constraint as other nodes). But we can also model switches without this capability to support legacy hardware. One way is to replace the flow conservation constraints at the switch with the traditional TE flow conservation constraints (what comes into the switch must go out).

Another option is to use the approach from TACCL [27]: replace switches with *hyper-edges* and allow the user to choose which hyper-edges to allow. For this second model we need to add additional constraints and due to limited space we refer the reader to [Appendix C](#) for the details.

The former two approaches are easier to use in practice: the user does not need to specify a sketch (which is a crucial in TACCL) or pick which GPU communicates with which other GPU — when we looked at the TACCL code we found the authors used their `uc-min` and `uc-max` strategy along with the user-specified sketch to automatically find which links to enable for switches within the node, but for cross-node links

they pre-identified which links perform best manually. We need to understand the topologies well to write such sketches and we found it difficult when we evaluated new topologies with TACCL. In contrast, our solution requires no human in the loop — the user only needs to specify the topology and the demand matrix — but the solver is slightly slower.

The objective. Our optimization objective is to finish the transfer as quickly as possible. We can encode this as follows:

$$\text{Objective function} \triangleq \sum_{\forall k \in K, \forall s, d \in N: s \neq d} \frac{1}{k+1} \mathcal{R}_{s,d,k}$$

Notice how the objective gives fewer rewards as k increases: the objective improves if the schedule satisfies the demand as soon as possible. If we combine the objectives with our constraints we arrive at an optimization that maximizes the objective subject to all of the above constraints.

One nuance here is that the optimization has multiple optima: the objective does not discourage solutions where we send flows that do not satisfy any demand (as long as the schedule satisfies all demands as quickly as possible the solution is optimal). Such solutions are clearly wasteful.

To avoid such *silly* cases, we can do one of two things: (a) we can either add a term to the objective to discourage unnecessary flows; or (b) we can zero out those flows in post-processing the solutions. The first results in higher solver runtimes as it becomes harder for the solver to prove optimality.

We use the latter approach where we run an algorithm similar to a reverse DFS. We start from each destination, and track the flows from that destination to the source until we account for its entire demand. We then remove (zero-out) all remaining flows as there is no demand corresponding to them. This takes $O(|N| + |E|)$ time where N is the number of nodes in the graph and E is the number of edges.

4 Scaling

Our formulation is general and pushes beyond the scale boundaries of SCCL and outperforms the solution quality of TACCL. But it is slow for topologies with more than 32 chassis. We next show two methods to scale this solution. The first works in situations where copy is not useful (*e.g.*, ALL-TOALL) and preserves optimality. The second is general (*i.e.*, supports copy): it solves the problem by partitioning it in time (its goal, in each time partition, is to make as much progress as it can towards finishing the transfer). This later model is sub-optimal, but outperforms the TACCL heuristic (see § 6) as it more accurately captures the optimization incentives and constraints. Its formulation allows users to trade-off optimality and speed by changing the number of partitions (smaller partitions increase sub-optimality but improve scalability).

4.1 Scaling by converting to a linear program

There is only one reason we needed integer variables for our model: copy! But some demands do not benefit from copy — this is when each destination wants a unique segment of information from each source. In these scenarios we can change our formulation into a linear program (LP). LPs are convex optimization programs which we can solve in polynomial time and scale much better than MILPs.

We remove support for copy and modify the flow conservation constraints back to their traditional form. The following constraint dictates: a node either buffers a chunk it received, forwards it in the next epoch, or consumes it. Notice a node can consume a chunk it received at the end of an epoch. We do not track individual chunks since we no longer need to worry about duplicates. This reduces the number of variables.

$$\begin{aligned} \text{Flow conservation constraints}(s, n, k) \triangleq \\ \sum_{\{j|(j,n) \in E\}} F_{s,j,n,k} - \lceil \delta_{jn} \rceil + B_{s,n,k} = \\ B_{s,n,k+1} + R_{s,n,k} + \sum_{\{j|(n,j) \in E\}} F_{s,n,j,k+1} \end{aligned}$$

The flow conservation constraints for switches are different: a switch does not consume chunks and does not buffer them — we remove those terms from the flow conservation equations.

Since destinations no longer need to both consume *and* forward chunks, we can modify the destination constraints:

$$\begin{aligned} \text{Destination constraint}(s, d, k) \triangleq \\ \mathcal{R}_{s,d,k} = \sum_{r=0}^k \mathcal{R}_{s,d,r} \quad \& \\ \mathcal{R}_{s,d,k} = \sum_{\forall c} D_{s,d,c} \end{aligned}$$

Our LP produces a *rate allocation* to demands that originate from each source on each link. From this we generate a schedule that we then execute in hardware (we translate these rates to paths for each chunk through the same DFS-like solution we described earlier). This is a straight-forward algorithm — TE solutions also use similar algorithms that we can adopt [17, 18] — and we omit it due to space constraints.

4.2 Scaling using the A^* technique

The LP form allows us to scale the solution to large topologies, but it does not permit copy. Copy is important for demands such as ALLGATHER (see § 2). We also provide a second scaling method inspired by the A^* technique in robotics [12].

We partition the problem into multiple rounds. In each round we no longer find a solution that satisfies all demands but instead motivate the solver to make as much progress towards this goal as it can. These optimizations have fewer

variables and are faster. We sequentially solve them one after the other until we reach a round where we meet all demands.

Here we need to address two new modeling challenges:

Encoding the right incentives. We need to remove the constraint that required the optimization to meet all demands by the last epoch — otherwise the optimization in each round may become infeasible. This means our objective function is no longer sufficient: it only says *if* it is feasible to satisfy a demand do so as fast as possible, but it does not reward incremental progress — we need to augment our objective with a term that rewards the optimization for moving data closer to the destinations in each round. But how to do this in a way that preserves the MILP format?

We augment our topology with logical links that allow us to compute this reward function: we add logical edges to the graph that connect each node to all the destinations and add weights to each of these logical edges that correspond to the minimum distance — we compute these weights using the Floyd Warshall algorithm [15] and the α -delay cost of each edge — from the node to each destination. We can now use these edges to encode a viable cost function which we can add to our original objective. Due to space constraints we refer the reader to the [Appendix D](#) for the details.

Modeling delay. Chunks that we send on any link (i, j) may not reach j by the end of the round (because of the α_{ij} -delay on that link) but instead arrive in a future round. We therefore need to maintain state from one round to the next and incorporate these late arrivals in our formulation.

We refer the reader to the appendix for the full formulation.

5 Important Considerations

Earlier we described how to formulate collective communication optimization using a TE approach. All three formulations (the general MILP form, the LP form, and A^*) find solutions for any input demand but only the general MILP form and the A^* model support copy. There are a number of parameters in these formulations we need to choose carefully:

Epoch durations and chunk sizes. A side-effect of using integer variables in the MILP formulation and the A^* -based technique is that the choice of chunk-size and epoch duration is important (the LP is not sensitive to these settings) — smaller epochs allow for finer-grained schedules that better leverage the available network capacity. To find the best chunk size we can sweep a range of values to find the best one quickly. We can also take this as an input — smaller chunks allow for finer grained schedules but can increase the resource usage on a node. Users can also utilize solutions such as [19] to pick what is optimum for their work-flow.

To set the epoch duration we can do one of two things: (a) to get the best schedule from the vanilla MILP formulation we can set the epoch duration to the time it takes the slowest link to transmit a chunk — the MILP cannot send anything

if we use smaller epochs because of the capacity constraints; (b) we can set the epoch duration based on the time it takes the *fastest* link to transmit a chunk. Option (b) enables the MILP to produce finer grained schedules but to use it we have to modify the capacity constraints and the flow conservation constraints: the capacity constraints ensure we don't exceed the capacity constraint on the slowest link and the flow conservation constraints ensure we do not forward a chunk before receiving it. Due to space constraints we refer the reader to the appendix for the details (see [Appendix F](#)). We compare the two approaches in § 6. Option (b) produces better schedules which is why we use it for most of our evaluations.

Number of epochs. We need to input an upper bound on the number of epochs which estimates how many epochs it may take to fully satisfy the demand: pick too small a number and the optimization will be infeasible, pick too large of a number and the MILP will be too large and too slow. To streamline finding the right number of epochs — and to not burden the user with having to identify what numbers to use — we develop a simple algorithm which finds a loose upper bound on how long we need to satisfy all the demands.

To find this number, we quickly sweep a range of transmission times: for each transmission time, we use coarser grain epoch durations (very large epochs) and run the optimization. Because we use large epoch sizes, we have fewer variables, which allows us to solve the optimization quickly. The solution of these runs is not optimal (because the epochs are too coarse), but it gives us an idea of how long we need when we switch to the optimal epoch duration. We describe the process in detail in [Algorithm 1](#) in the [Appendix E](#). We use the output to initialize the optimization which automatically identifies if a lower number of epochs is sufficient.

Number of epochs in a round in A^* . We solve round after round of A^* until we deliver all the demands. Users can choose how many epochs to use in each round. The smaller the number of epochs in a round, the faster the optimization and the higher the optimality gap. Picking a small number of epochs per round also impacts the state we need to maintain. In our experiments, we set the number of epochs such that chunks do not arrive later than one round in the future.

The topology, α , and β inputs. TE-CCL takes the topology and the values for α and β as input. We do not provide an independent method for computing these values.

Which switch model to use. We provide two switch models: one that allows the switch to copy chunks (to model networks with the SHArP protocol [9] enabled) and one which does not (the latter is similar to TACCL's hyper-edge model). It is up to the user to decide which model to use in the optimizer.

Modeling variable bandwidth. Our model supports networks with variable bandwidth. To add support for this, we have to assume bandwidth only changes from one epoch to

the next. We can then take the capacity matrix for each epoch and use that in our capacity constraints.

Use in multi-tenant clusters. TE-CCL supports multi-tenant communication optimization: all our models accept a network demand as input — to model a multi-tenant environment we have to change the demand matrix to the sum of the demands across all collectives. The capacity constraints will ensure we do not exceed network capacity and the objective ensures we minimize the total completion time across all tenants.

We can also support priorities across tenants (*i.e.*, prioritizing one tenant's completion time over the others) if we add a separate buffer and read variable for each tenant: we can then add the priorities to the objective function. This change increases the number of variables in the MILP which slow it down — we may have to use A^* in this case but this does not impact the quality of the solution compared to when we solve a single tenant problem at the same scale.

Scaling through intermediate solutions. The solver we use, Gurobi [25], often finds an optimal solution and then spends a long time proving it is optimal — often the solution does not improve even after the solver runs for an additional 10 hours. We therefore apply a timeout and stop the solver after 2 hours and use the solution at that point. Gurobi reports its progress through the primal-dual gap [4].

6 Evaluation

We implement our solution in Python. We use Gurobi [25] to solve the optimizations. We convert our solution into MSCCL [5], which can then port it into a schedule that runs on the hardware. We plan to release our code.

The goal in this evaluation is to:

- Compare TE-CCL to state-of-the-art: both in scale and in terms of solution quality.
- Show TE-CCL scales to the large topologies.
- Show the impact of each of our different design choices.

Metrics. We use the following metrics to evaluate TE-CCL:

Solver time. The time it takes — which includes the time to setup the variables and constraints in the solver — to solve the collective optimization problem.

Transfer time. The time it takes for the transfer to complete: for all the nodes to receive their full demand.

Output buffer size. The data each GPU receives once we satisfy the demand (we borrow this from TACCL [27]).

Transfer size. The amount of data each GPU sends to others: for example, a GPU in an ALLGATHER demand with a transfer size of 1 GB sends 1 GB of data to *each* other GPU.

Algorithmic bandwidth. The output buffer size divided by the transfer time (this metric is from TACCL [27]).

Topologies and workloads. We evaluate TE-CCL using the topologies in [Table 2](#). We use common topologies such as

Topology	# of GPUs per chassis	# of edges per chassis
Internal 1	4	8
Internal 2	2	2
DGX1	8	32
NDv2	8	32
DGX2	17	32

Table 2: Our topologies. The internal topologies are from a large public cloud and are proprietary: α is $0.6\mu\text{s}$ and $0.75\mu\text{s}$ on their GPU to GPU and GPU to switch links.

DGX1, DGX2 [23], and NDv2 [2] as well as two proprietary topologies from a public cloud provider.

TE-CCL variants. We use three variants of TE-CCL in our evaluations: the optimal (where we use the vanilla MILP for ALLGATHER and LP for ALLTOALL), the early-stop version for ALLGATHER (where we use Gurobi’s ability to find a good solution – which is at most 30% away from optimal – quickly), and A^* for ALLGATHER.

Gurobi runs into numerical issues with ALLTOALL on large topologies (more than 64 nodes): we need to run it with a different configuration (`method = 2` [11]) which causes it to produce a feasible (but not optimal) solution. In those cases, we run the solver in a loop and do a binary search (on the number of epochs) to find the optimal solution.

We set the epoch duration based on the bandwidth of the fastest link. In the cases where $\alpha > 200 \times \tau$ we increase the epoch duration by $5\times$ to avoid large models (since α dominates this does not materially impact the solution).

TE-CCL solves optimization problems to produce a schedule, and the optimization is deterministic, outputting the same number of epochs to meet the demand every time we run it. The solver times also do not vary significantly for a given optimization across runs. **Baselines.** We compare our solution to two state-of-the-art solutions: TACCL [27] and SCCL [5].

TACCL. We obtained the TACCL code from the authors and track and report the solver time. TE-CCL takes an additional β compared to TACCL to route chunks through a switch: TACCL replaces the switch with direct edges between the nodes and only pays one transmission delay to cross that link whereas TE-CCL models the switch itself and pays two transmission delays — one from the node to the switch and one from the switch to the node. To compare fairly against TACCL we change our model of the switch to do the same when comparing with TACCL.

SCCL. We compare to SCCL using the public SCCL codebase [20] and also re-ran our experiments using the SCCL artifact from their submission (which the authors gave us). We verified and confirmed with the authors we used SCCL correctly and that our numbers are correct.

Platform. We use the solvers and the schedules they produce to compute the transfer times and algorithmic bandwidth for

Collective, # chunks	SCCL (μs)	TE-CCL (μs)
ALLGATHER, 1	3.4	4
ALLGATHER, 2	5.1	5
ALLGATHER, 3	8	6.1
ALLTOALL, 1	3.4	4

Table 3: Comparing the transfer time from SCCL least-steps with TE-CCL ($K = 10$ and chunk size = 25 KB). TE-CCL can better pipeline chunks and so pays less α cost with larger transfers.

SCCL, TACCL, and TE-CCL. We checked using a single 8 GPU DGX1 node that these estimates match what we get from running on hardware for both TE-CCL and TACCL.

We report the capacity and delay for the public topologies in the Appendix H.

Unexplored avenues. We show from testing on a DGX1 that TE-CCL’s estimates of collective latency match the actual runtimes on prototype hardware. We do not have access and the budget to run hardware experiments at scale on different kinds of GPUs. Thus, the effect of factors such as congestion, message batch sizes and other GPU implementation artefacts on the collective latency remains an unknown. But our results on all of the other metrics such as solver times and our ability to scale to large topologies hold regardless.

6.1 Comparison to SCCL and TACCL

SCCL. SCCL has two modes: one minimizes latency (`least-steps`) and one produces an instance solution (`instance`) with the number of chunks, rounds, and steps as input.

Our solution is equivalent to the former but the SCCL `least-steps` command took over a day to produce a solution for ALLGATHER demands with more than 3 chunks and ALLTOALL demands with more than 1 chunk on a DGX1 topology (the SCCL paper does not evaluate this mode). In contrast, we ran TE-CCL with $\max K = K = 10$ (the maximum number of epochs the optimization can use to satisfy the demand) and 25KB chunks, and it finished in $\leq 0.65\text{s}$ for all ALLGATHER demands and $\leq 0.97\text{s}$ for ALLTOALL demands with less than 5 chunks.

We used 25KB chunks to capture the impact of α ($\alpha = 0.7\mu\text{s}$) on the solutions (Table 3): for all > 1 chunk cases TE-CCL outperforms SCCL as it models the α delay better — it ensures a node receives a chunk before forwarding it but pipelines traffic; SCCL enforces a barrier instead. SCCL performs better in the 1 chunk case as TE-CCL cannot leverage its ability to pipeline.

We also compare with SCCL’s instance solution (due to space constraints, we show the results in the Appendix G). To create an apples-to-apples comparison, we use the number of rounds in SCCL for K in TE-CCL — since SCCL is no longer running an optimization — and use $\alpha = 0$ (this is necessary as

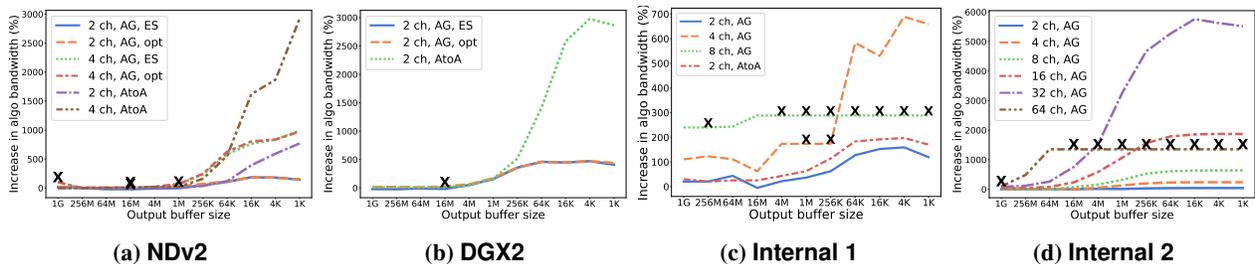


Figure 4: Compares the algorithmic bandwidth of TE-CCL and TACCL ($\frac{100(TECCL-TACCL)}{TACCL}$). We mark the scenarios where TACCL is infeasible — which cause dips in the graph — using an X.

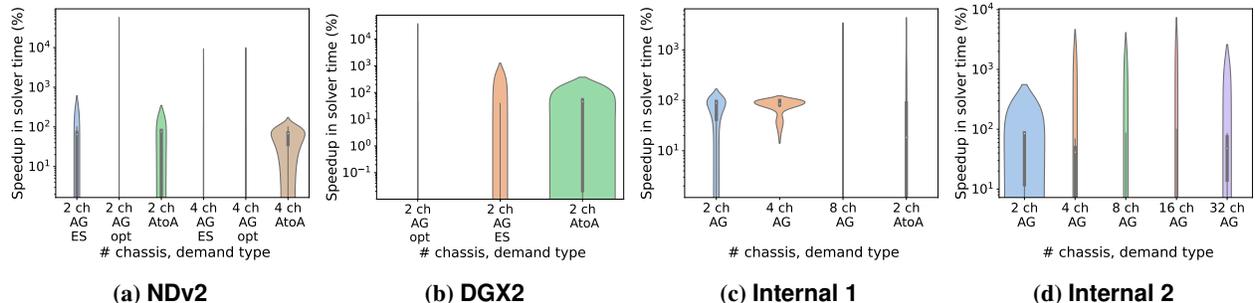


Figure 5: Compares the solver time of TE-CCL and TACCL ($\frac{100(TECCL-TACCL)}{TACCL}$). Ch stands for chassis, ES early stop, AG ALLGATHER, and AtoA ALLTOALL. We use log scale for the y-axis to improve resolution. TE-CCL is faster than TACCL on 45% of ALLTOALL scenarios and 40% of ALLGATHER scenarios (with early stop) on the NDv2 topology; 72% and 27% for DGX2; 72% and 83% for Internal 1; and 100% and 50% for Internal 2.

our model will need more epochs otherwise to account for α . We use the scenarios from Table 4 in SCCL [5] and run both solvers on a desktop with 6 cores and 32 GB RAM. SCCL failed to produce a solution for ALLGATHER workloads with more than 1 chunk even after 3 days. TE-CCL runs faster than SCCL in almost all cases and even improves SCCL’s solution quality by 33% in the ALLTOALL scenario. TE-CCL is slower than SCCL in one instance ((6, 7)): this is because in TE-CCL we solve for the optimal number of epochs, and we use a value for K that is too tight — we can reduce the solver time to 11 seconds by increasing K to 20 (the quality of the solution does not change). We can use the A^* technique to speed up the solution further.

To fully highlight our runtime advantage over SCCL, we ran an ALLTOALL demand with 8 chunks using both solvers: SCCL timed out after 10032.7s and did not produce a schedule, whereas ours finished in 1.88s with a valid schedule that finished the transfer in $21\mu s$ (for 25KB chunks).

TACCL. We compare the solver time and algorithmic bandwidth of TE-CCL and TACCL using ALLGATHER and ALLTOALL demands and on DGX2 and NDv2 based topologies with up to 34 nodes (a 2-chassis DGX2 topology has 34 nodes) and on both internal topologies with up to 128 nodes. We ran all experiments on a Linux Ubuntu 20.04 VM with two Intel Xeon(R) Platinum 8380 CPUs with a total of 80-cores/160-threads and 512 GB RAM and used Gurobi 9.5.2 version as

our solver. TACCL ALLTOALL does not terminate for large topologies (including the 2 chassis DGX2 ALLTOALL) — we use a timeout of 2 + 2 hrs or 4 + 4 hrs for their routing and scheduling phases depending on the topology size.

TACCL ran out of memory and did not produce a solution for large Internal 2 topologies (with over 64 chassis) and for almost all Internal 1 topologies (with over 4 chassis). Table 4 reports the numbers for TE-CCL on ≥ 64 nodes topologies.

TACCL scales better on the NDv2 topology compared to internal topologies 1 and 2. In NDv2 only 2 nodes in a chassis connect to a switch but in internal topologies 1 and 2 many nodes in a chassis are connected to a switch — TACCL replaces the switch with direct edges; as we increase the size of internal topologies 1 and 2 the number of such edges increases exponentially. The TACCL authors recommended we use a sketch that only uses a subset of these edges. Doing so improved the runtime for smaller topologies but TACCL still failed to produce a solution after 8 hours for larger ones.

TE-CCL often produces higher quality solutions compared to TACCL (in some cases TACCL fails to produce a schedule and times out — we show those cases with an X): on DGX2 the improvement is at least 12% and 9% (maximum 471% and 2979%) for ALLGATHER and ALLTOALL respectively; on NDv2 0.36% and 0.18% (maximum 970% and 2919%); on Internal 1 –5% and 20% (maximum 689% and 197%), and on Internal 2, 0.33% and 0.48% (maximum 5759% and 12322%). We show these results in Figure 4 and Figure 6 (we report

ALLTOALL numbers for Internal 2 separately for clarity). We report the raw algorithmic bandwidths for TE-CCL variants in the appendix (see Table 8) for NDv2 2 chassis as a sample.

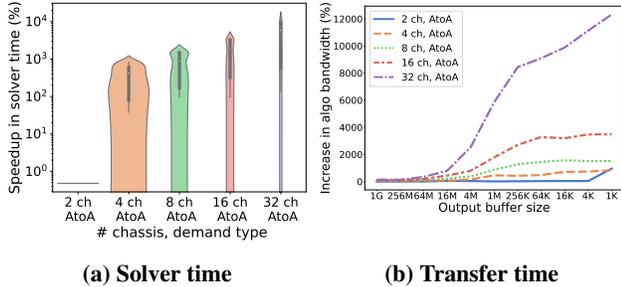


Figure 6: We compare TACCL and TE-CCL for ALLTOALL demands on Internal 2 with different number of chassis. TE-CCL is faster than TACCL in all cases and also produces higher quality solutions.

We use Gurobi’s early-stop for ALLGATHER demands to improve TE-CCL’s ability to scale: this does not materially impact the quality of TE-CCL’s solution — even with an aggressive optimality gap threshold of 30% — but allows TE-CCL to solve the problem faster in the ALLGATHER scenario (we found TACCL also uses this under the hood – our solver time matches TACCL even when TACCL uses this feature). TACCL uses this early stop mechanism in the ALLTOALL case as well but we run TE-CCL to completion: TE-CCL always produces schedules that match or beat those of TACCL and in many cases it produces these schedules more quickly. We compare the two solver times in Figure 5.

6.2 Scale

TACCL often crashes on large topologies, either due to requiring more than 400 GB RAM or memory leaks and segmentation faults. TE-CCL also requires a lot of memory in some cases (around 350 GB for ALLTOALL on large topologies), but we can control this by changing the epoch duration to trade off the quality of the solution with the amount of memory the solver needs. Table 4 summarizes our results on large topologies and reports the scale factor (EM). We use output buffer sizes larger than 16 MB — as the number of GPUs increases, chunks become too small beyond this point. We adjust the epoch size by a factor of, at most, 4 for these cases to limit memory usage.

6.3 Microbenchmarks

We next evaluate our design choices:

Copy. In-network copy is most helpful for large transfers where there is not enough capacity to transfer multiple copies directly from the source to each destination: we see in the largest transfer size (0.21 GB) copy reduces the transfer time by 50% for DGX1, the Internal 1 with $\alpha = 0$ and $\alpha > 0$, and

Topology	Collective	# GPUs	EM	Solver time
Internal 1	AG (A*)	64	1	3000 s
Internal 1	AG (A*)	128	1	7 h
Internal 2	AG (A*)	128	1	1300 s
Internal 2	AG (A*)	256	2	2.8 h
Internal 1	AtoA	16	1	66 s
Internal 1	AtoA	32	1	215 s
Internal 1	AtoA	64	1	500 s
Internal 1	AtoA	128	2	800 s
Internal 2	AtoA	128	1	2600 s
Internal 2	AtoA	256	4	1500 s

Table 4: Large Topologies for which TACCL can’t synthesize the schedule. The solver time is the average TE-CCL time to synthesize the schedule and EM is the epoch multiplier factor to change the epoch duration from the optimal duration for scalability.

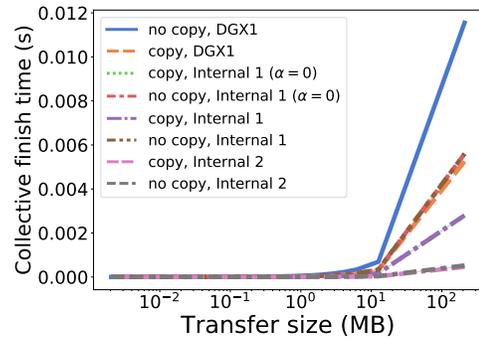


Figure 7: The benefit of copy: for large transfers, copy helps finish the transfer faster.

12.5% for Internal 2. In-network copy does not help with small transfers as there is enough capacity between the source and the destinations to send multiple copies of the data directly from the source. We use 4 chunks to complete these transfers.

Small vs large epochs. We investigate how the duration of epochs impacts the solver speed and the quality of the solution (Figure 8 where we use 2 chassis for each topology). In ALLGATHER we only allow chunks to traverse one link in a single epoch: the length of the longest path dominates the transfer time when we use large epochs because the length of the epoch is too large compared to how long it takes for the chunk actually to traverse the link (on faster links). We see this more predominantly in the NDv2 and DGX2 topology where the fast links have 4× higher bandwidth (large epoch duration is, therefore, 4× small epoch duration) compared to slower ones. In contrast, we do not see a difference on Internal 1, where the links are mostly homogeneous.

Store and forward. We find a somewhat surprising result. Buffers don’t impact the solution quality but only the solver time (Figure 9)! This is because of the nature of collective demands such as ALLGATHER and ALLTOALL: because each node needs the same amount of traffic as it has to forward it

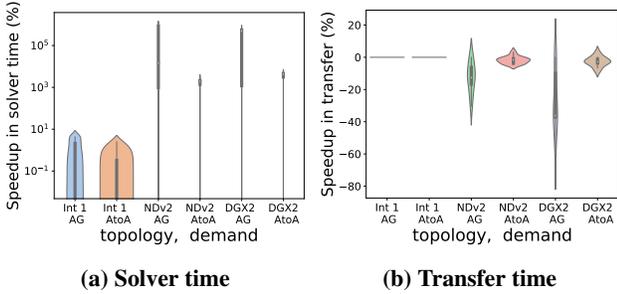


Figure 8: We compare the impact of small vs large epochs on the solver speed (a) and solution quality (b). We use 2 chassis for all topologies. Both graphs compute $\frac{100(\text{small}-\text{large})}{\text{large}}$. The solver finds a solution faster with large epochs but produces better quality solutions with small ones.

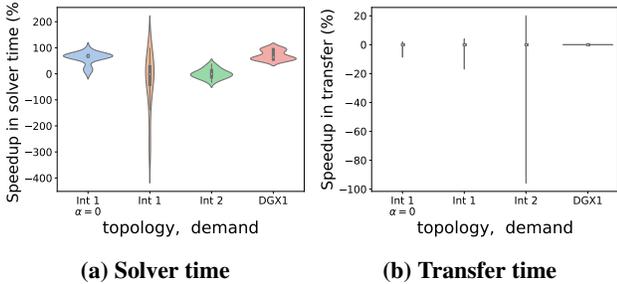


Figure 9: We evaluate the impact of buffers on (a) and solution quality (b) solver time. We use 2 chassis for all topologies. Both graphs compute $\frac{100(\text{without buffers}-\text{with buffers})}{\text{without buffers}}$. Buffers don't impact the solution quality in most cases, but only the solver times! The average speedups in solver time are: 61%, -28.46%, 0.23%, 71% for Internal 1 without α , Internal 1 with α , Internal 2, and DGX1 respectively.

can interleave consuming traffic with forwarding it to compensate for the lack of buffers. But in the presence of buffers the feasible space of solutions is larger which in many cases enables the solver to find the optimal solution more quickly (the speedup is 71% and 61% for Internal 1 and DGX1 respectively). We believe it is possible to formally prove this result but defer this proof to future work.

A* vs OPT. We compared the quality of the A* technique to the optimal on a 16-chassis Internal 2 topology with both $\alpha > 0$ and $\alpha = 0$. We used both single chunk and 2 chunk transfers.

When $\alpha = 0$, A* finished in 86.61s (263.29s for 2 chunk demands) whereas the optimal took 346s (4392s for two chunks). The optimal solution was 10% better than A* (6% in the 2 chunk case) — transfer times were 3.48s vs 3.89s.

The results are similar when $\alpha > 0$: A* finished in 137.02s (901.25s for the 2 chunk case) whereas the optimal took

363.40s (3047s). The optimal solution was 20% better (8% in the 2 chunk case).

7 Related work

TE-CCL provides a scalable method for collective communication optimization by using a network flow-based approach. Our solution supports unsustained demands, store-and-forward, and copy. Our work builds on prior work both in network traffic engineering and in collective optimization: **Multi-cast TE**. Prior works have looked at traffic engineering for multi-cast networks [10, 22]. Oliveira and Pardalos [24] provide a comprehensive summary of these works. Blink [29] used these techniques to optimize collective communication but does not model delay and store-and-forward.

WAN TE. Many prior works in networking use the network flow model to scalably route traffic in wide area networks [1, 14, 16, 21]. However, most of these works assume sustained demands, copy, and store-and-forward. Among these works, Calendaring [17] provides a solution that models unsustained demands. NetStitcher [18] adds to this the support for store and forward but assumes flows do not compete for bandwidth. Neither of these works simultaneously model copy, store-and-forward, and delay.

Prior work on collective communication optimization. Many prior work have tackled the collective communication optimization problem [5, 26, 27, 29, 31]. We find these solutions do not scale to the topologies and data sizes we have in production today and those we anticipate for the future. TACCL is the most scalable of these solutions, but it has trouble scaling when it sends more than 1-2 chunks, and is sub-optimal. Work such as [19, 30, 31] aims to co-optimize either topologies and parallelization strategies ([30]) or collective scheduling and execution planning [19]. These works rely on collective communication optimizers as part of their search but do not provide optimal solutions to the problem themselves — they can use TE-CCL as part of their search. Our work is complementary to these works.

8 Conclusion

We presented TE-CCL: a scalable collective communication optimizer that models the problem through a TE-based approach. We provide three algorithms to solve this problem: the MILP approach which optimally solves the general collective communication optimization problem and supports multi-cast; the LP form which is also optimal and much more scalable but removes support for multi-cast; and finally the A*-based approximation method which is much more scalable than the MILP technique and continues to support multi-cast but is no longer optimal. We show our solution outperforms prior, state-of-the-art, techniques such as SCCL and TACCL by over 2 \times .

This work does not raise any ethical concerns.

References

- [1] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. 2021. Contracting wide-area network topologies to solve flow problems quickly. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*. 175–200.
- [2] Azure NDv2-series 2021. (2021). <https://learn.microsoft.com/en-us/azure/virtual-machines/ndv2-series>
- [3] Dimitris Bertsimas and John N Tsitsiklis. 1997. *Introduction to linear optimization*. Vol. 6. Athena Scientific Belmont, MA.
- [4] Stephen Boyd and Lieven Vandenbergh. 2004. *Convex Optimization*. Cambridge University Press.
- [5] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2021. Synthesizing Optimal Collective Algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 62–75. <https://doi.org/10.1145/3437801.3441620>
- [6] ChatGPT runs 10K Nvidia training GPUs with potential for thousands more 2023. (2023). <https://www.fierceelectronics.com/sensors/chatgpt-runs-10k-nvidia-training-gpus-potential-thousands-more>
- [7] Wei Deng, Junwei Pan, Tian Zhou, Deguang Kong, Aaron Flores, and Guang Lin. 2021. DeepLight: Deep Lightweight Feature Interactions for Accelerating CTR Predictions in Ad Serving. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining (WSDM '21)*. Association for Computing Machinery, New York, NY, USA, 922–930. <https://doi.org/10.1145/3437963.3441727>
- [8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
- [9] John Matthew Simon Doar. 1993. *Multicast in the asynchronous transfer mode environment*. Technical Report. University of Cambridge, Computer Laboratory.
- [10] M. Doar and I. Leslie. 1993. How bad is naive multicast routing?. In *IEEE INFOCOM '93 The Conference on Computer Communications, Proceedings*. 82–89 vol.1. <https://doi.org/10.1109/INFCOM.1993.253246>
- [11] Gurobi Algorithm used to solve continuous models 2023. (2023). <https://www.gurobi.com/documentation/9.1/refman/method.html>
- [12] Peter Hart, Nils Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107. <https://doi.org/10.1109/tssc.1968.300136>
- [13] Roger W Hockney. 1994. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel computing* 20, 3 (1994), 389–398.
- [14] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving High Utilization with Software-Driven WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 15–26. <https://doi.org/10.1145/2486001.2486012>
- [15] Stefan Hougardy. 2010. The Floyd–Warshall algorithm on graphs with negative cycles. *Inform. Process. Lett.* 110, 8-9 (2010), 279–281.
- [16] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. 2013. B4: Experience with a globally-deployed software defined WAN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 3–14.
- [17] Srikanth Kandula, Ishai Menache, Roy Schwartz, and Spandana Raj Babbula. 2014. Calendaring for Wide Area Networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 515–526. <https://doi.org/10.1145/2619239.2626336>
- [18] Nikolaos Laoutaris, Michael Sirivianos, Xiaoyuan Yang, and Pablo Rodriguez. 2011. Inter-Datacenter Bulk Transfers with Netstitcher. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. Association for Computing Machinery, New York, NY, USA, 74–85. <https://doi.org/10.1145/2018436.2018446>
- [19] Kshiteej Mahajan, Ching-Hsiang Chu, Srinivas Sridharan, and Aditya Akella. [n. d.]. Better Together: Jointly Optimizing ML Collective Scheduling and Execution Planning using SYNDICATE. ([n. d.]).
- [20] MSCCL codebase [n. d.]. ([n. d.]). <https://github.com/microsoft/msccl>
- [21] Deepak Narayanan, Fiodar Kazhemiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. 2021. Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 521–537. <https://doi.org/10.1145/3477132.3483588>
- [22] C.A. Noronha and F.A. Tobagi. 1994. Optimum routing of multicast streams. In *Proceedings of INFOCOM '94 Conference on Computer Communications*. 865–873 vol.2. <https://doi.org/10.1109/INFCOM.1994.337651>
- [23] Nvidia DGX System 2021. (2021). <https://www.nvidia.com/en-us/data-center/dgx-systems/>
- [24] Carlos AS Oliveira and Panos M Pardalos. 2005. A survey of combinatorial optimization problems in multicast routing. *Computers & Operations Research* 32, 8 (2005), 1953–1981.
- [25] Joo Pedro Pedroso. 2011. Optimization with gurobi and python. *INESC Porto and Universidade do Porto., Porto, Portugal* 1 (2011).
- [26] Saeed Rashidi, William Won, Sudarshan Srinivasan, Srinivas Sridharan, and Tushar Krishna. 2022. Themis: A Network Bandwidth-Aware Collective Scheduling Policy for Distributed Training of DL Models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 581–596. <https://doi.org/10.1145/3470496.3527382>
- [27] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. 2021. TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches. (2021). <https://doi.org/10.48550/ARXIV.2111.04867>
- [28] Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir, and George Varghese. 2021. Campion: Debugging Router Configuration Differences. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 748–761. <https://doi.org/10.1145/3452296.3472925>
- [29] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. 2020. Blink: Fast and generic collectives for distributed ml. *Proceedings of Machine Learning and Systems* 2 (2020), 172–186.
- [30] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. 2022. TopoOpt: Co-optimizing Network Topology and Parallelization Strategy for Distributed Training Jobs. (2022). <https://doi.org/10.48550/ARXIV.2202.00433>
- [31] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. 2022. Optimal Direct-Connect Topologies for Collective Communications. (2022). <https://doi.org/10.48550/ARXIV.2202.03356>

A Initialization and termination constraints

We introduced the main constraints for the MILP and LP formulations in § 3 and § 4.1. However, we need to add a few additional constraints to initialize and terminate these problems.

The first epoch. We use buffers to indicate when the node has a specific chunk. In the first epoch of the MILP we initialize the source buffers as follows:

$$\begin{aligned} B_{n,n,0,c} &= \max_{d \in N} D_{n,d,c} \quad \forall n \in N, \forall c \in C \\ B_{s,n,0,c} &= 0 \quad \forall s, n \in N : s \neq n, \forall c \in C \end{aligned}$$

We no longer need to buffer chunks we have already sent out in the LP form and therefore these equations become:

$$B_{s,n,0} + \sum_{\forall j: (n,j) \in E} F_{s,n,j,0} = \sum_{\forall c \in C, \forall d \in N} D_{s,d,c} \quad \forall s, n \in N : s, n \notin S$$

The last epoch. In the LP we do not need to buffer chunks if they are not going to be forwarded. Nodes also don't need to send out any traffic after this epoch. Therefore, in the last epoch of the LP we have:

$$\forall s, n \in N : s \neq n, n \notin S \quad \sum_{\forall j: (j,n) \in E} F_{s,j,n,(K - \lceil \frac{\alpha_{j,n}}{\tau} \rceil)} = R_{s,n,K}$$

B Modeling limited buffers

In the MILP. To model limited buffers in the MILP we need to change the buffer constraints to track which chunks to remove from the buffer and in which epoch. Hence, we introduce a new variable $X_{s,n,k,c}$ which encodes whether we should remove chunk c from node s from the buffer at node n in epoch k . The buffer constraints become:

Buffer constraints $(s, n, k, c) \triangleq$

$$B_{s,n,k,c} = B_{s,n,k-1,c} - X_{s,n,k-1,c} + \sum_{\forall j: (j,n) \in E} F_{s,j,n,k - \lceil \delta_{jn} \rceil - 1, c}$$

To enforce the limit on the buffer size, we add the constraint:

$$\sum_{s,c} B_{s,n,k,c} \leq \mathcal{L} \quad \forall n \in N, \forall k \in K,$$

where \mathcal{L} is the limit on the buffer size. We impose no limit on the auxiliary variable $X_{s,n,k-1,c}$ as the algorithm can choose to re-buffer a chunk at a node at any point in time and again remove it later.

In the LP. The LP removes from the buffer what it sends out on a link. Hence to use limited buffers we only have to impose an upper bound on the sum of the buffer variables at a node:

$$\sum_s B_{s,n,k} \leq \mathcal{L} \quad \forall n \in N, \forall k \in K$$

C Modeling legacy switches

For switches that don't support copy, we use an approach similar to TACCL's hyper-edges. We remove the switch from the topology and replace it with direct links between all pairs of GPUs that were connected through the switch. We now need to account for the capacity to and from the switch: this translates to an upper bound on the number of hyper-edges we can use simultaneously in each epoch.

We augment our notation with the variables in Table 5. We need to add a constraint to the problem that enforces we can only use a subset of the hyper-edges: the minimum of the number of edges that come into the switch and go out of it. This constraint is as follows:

$$\sum_{\forall n \in N, \forall c \in C, \forall (i,j) \in \Omega(s)} F_{n,i,j,k,c} \leq \min(|\{(s,x) \in E\}|, |\{(y,s) \in E\}|) \quad \forall k \in K, \forall s \in S$$

Each node i can only send (receive) traffic on one of its outgoing (incoming) hyper-edges:

$$\begin{aligned} \forall k \in K, \forall i \in N, \forall s \in S \quad \sum_{\forall n \in N, \forall c \in C, \forall (i,j) \in \Omega(s)} F_{n,i,j,k,c} &\leq 1 \\ \forall k \in K, \forall i \in N, \forall s \in S \quad \sum_{\forall n \in N, \forall c \in C, \forall (j,i) \in \Omega(s)} F_{n,j,i,k,c} &\leq 1. \end{aligned}$$

We only need to use this model in the general MILP form to ensure the solution can scale — the LP model already assumes none of the nodes copy traffic.

D The A^* technique

In the A^* based approach we split the problem into multiple time partitions (or rounds). Our goal in each round is to get the chunks closer to the destination. We solve each of these rounds sequentially until we satisfy all the demands.

The delay on each link (*i.e.*, α_{ij}) means some chunks we send on link (i, j) in a particular round may arrive at node j in a subsequent round. We use the set K' to denote all subsequent rounds and $Q_{s,c,i,k',r}$ to denote the chunks that arrive in these rounds to account for this (Figure 10). To keep things simple, we choose to set the number of epochs in a round in a way that ensures chunks are only delayed by a single round at most. This means the total duration of the round is greater than the largest link delay. However, users can choose to use shorter chunks — they will have to maintain more state between rounds in that case.

To encode A^* we maintain most constraints from the MILP formulation but need to modify the objective function and the buffer constraints to account for chunks arriving in future

Notation Description

Γ	The function to get non-switch set of edges from the set of edges ($\Gamma : E \rightarrow E'$). Therefore, $E' \subseteq 2^{N-S \times N-S}$ and $(i, j) \in E' \implies (i, j) \in E \wedge i, j \notin S$.
Ω	The function from a switch node to the set of direct-connect edges ($\Omega : S \rightarrow 2^{N-S \times N-S}$). $\Omega(s) = \{(i, j) (i, s) \in E \wedge (s, j) \in E \wedge (i, j) \notin E\}$
L	The set of edges in the transformed graph ($L = \Gamma(E) \cup \bigcup_{s \in S} \Omega(s)$).

Table 5: Additional notation we need to model legacy switches.

Variable Description

R	The set of rounds ($R = \{0, 1, 2, \dots, R\}$)
K	The set of epochs in a round ($K = \{0, 1, 2, \dots, K\}$). The number of epochs in a round is constant and does not change with the round.
K'	The set of future epochs relevant for a round ($K' = \{0, 1, 2, \dots, \max_{\forall (i,j) \in E} \lceil \frac{\alpha_{ij}}{\tau} \rceil\}$)
D	The demand function ($N \times N \times C \rightarrow \{0, 1\}$) where $D_{s,d,c,r}$ represents whether destination d wants chunk with id c from node s at the start of round r
$F_{s,c,i,j,k,r}$	(boolean) whether chunk c of source s is going over link $(i, j) \in E$ at epoch k in round r
$B_{s,c,i,k,r}$	(boolean) whether chunk c of source s is in node i 's buffer at the start of epoch k in round r
$Q_{s,c,i,k,r}$	(boolean) whether chunk c of source s is in node i 's buffer at the start of future epoch k' in round r .
$\mathcal{R}_{s,c,d,k,r}$	whether chunk c of source s is delivered to node d by the end of epoch k in round r

Table 6: New variables for the A^* technique.

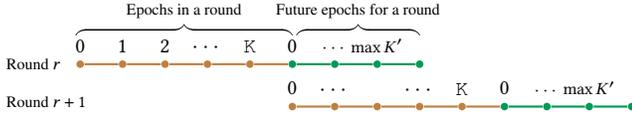


Figure 10: A^* time progression between rounds

rounds. For switches, we need to modify the flow conservation constraints as they do not have enough memory for buffering. **Look ahead constraints.** To account for chunks that will arrive in the subsequent epoch we need to maintain additional state. For none switch nodes, if the chunk arrives in the first epoch of the next round ($k' = 0$) we have:

$$Q_{s,n,c,0,r} = B_{s,n,c,K,r} + \sum_{\forall j:(j,n) \in E} F_{s,j,n,c,K - \lceil \frac{\alpha_{jn}}{\tau} \rceil, r}$$

$$\forall s, n \in N : n \notin S, \forall c \in C$$

and for all later arrivals we have:

$$Q_{s,n,c,k',r} = Q_{s,n,c,k'-1,r} + \sum_{\forall j:(j,n) \in E \wedge (k' - \lceil \frac{\alpha_{jn}}{\tau} \rceil) <= 0} F_{s,j,n,c,K+k' - \lceil \frac{\alpha_{jn}}{\tau} \rceil, r}$$

$$\forall s, n \in N : n \notin S, \forall c \in C, \forall k' \in K' : k' > 0.$$

These equations allow us to store in the variables Q what chunks are arriving in the next round. Notice how we also account for buffers by $B_{s,n,c,k,r}$ in $k' = 0$ and by $Q_{s,n,c,k'-1,r}$ for the $k' > 0$ case. Since the switches do not have large enough buffers we use the following:

$$Q_{s,n,c,k,r} = \sum_{\forall j:(j,n) \in E \wedge (k' - \lceil \frac{\alpha_{jn}}{\tau} \rceil) <= 0} F_{s,j,n,c,K+k' - \lceil \frac{\alpha_{jn}}{\tau} \rceil, r}$$

$$\forall s, n \in N : n \in S, \forall c \in C, \forall k' \in K'$$

All that we have to do now is to set the buffers at the beginning of each round $r > 0$ to Q (we exclude $r = 0$ since there is no prior round, and we can use the same initialization that we had earlier):

$$B_{s,n,c,0,r} = Q_{s,n,c,0,r-1}$$

$$\forall s, n \in N : s \neq n \wedge n \notin S, \forall c \in C, r > 0$$

For $k > 0$, if $Q_{s,n,c,k-1,r-1} = 0$ and $r > 0, k <= \max K'$ we have:

$$\forall s, n \in N : n \notin S, \forall c \in C, \forall k \in K : k > 0$$

$$B_{s,n,c,k,r} = B_{s,n,c,k-1,r} + \sum_{\forall j:(j,n) \in E} F_{s,j,n,c,k - \lceil \frac{\alpha_{jn}}{\tau} \rceil - 1, r} + Q_{s,n,c,k,r-1}$$

otherwise:

$$\forall s, n \in N : n \notin S, \forall c \in C, \forall k \in K : k > 0$$

$$B_{s,n,c,k,r} = B_{s,n,c,k-1} + \sum_{\forall j:(j,n) \in E} F_{s,j,n,c,k - \lceil \frac{\alpha_{jn}}{\tau} \rceil - 1}$$

Specifically, we are adding to the buffer what is arriving from the previous round. The two cases are there to ensure we account for each arrival only once for non-switch nodes. The equations are similar for switches:

$$\forall s, n \in N : n \in S, \forall k \in K : k > 0, \forall c \in C$$

$$\max_{\forall j: (n,j) \in E} F_{s,n,j,c,k,r} \leq \begin{cases} \sum_{\forall j: (j,n) \in E} F_{s,j,n,c,k-\lceil \frac{\alpha_{j,n}}{\tau} \rceil - 1} + Q_{s,n,c,k,r-1} & r > 0, k \leq \max K' \\ \sum_{\forall j: (j,n) \in E} F_{s,j,n,c,k-\lceil \frac{\alpha_{j,n}}{\tau} \rceil - 1} & \text{otherwise} \end{cases}$$

but since switches don't buffer chunks we incorporate them into the flow conservation constraints.

The objective. We now need to motivate the optimization in each round to get the chunks closer to the destination (while making it even more profitable to satisfy the demand fully). So first, we need to automatically compute this additional payoff. To do this, we add logical edges to the graph that allow nodes to form a clique. We assign a weight to each of these edges which we calculate using the Floyd Warshall algorithm [15] and the values for α_{ij} . The chunks we send in this epoch that don't contribute to satisfying a demand are stored in our Q variables. We now introduce a new variable: $P_{s,d,k',r}$ — the total number of chunks coming from source s and going towards destination d that are currently on their way towards the destination. We have:

$$P_{s,d,k',r} \leq \sum_{\forall n \in N, \forall c \in C: D_{n,d,c,r}=1} Q_{n,s,c,k',r} \quad \forall k' \in K', \forall s, d \in N$$

$$\sum_{\forall s \in N} P_{s,d,k',r} = \sum_{\forall s \in N, \forall c \in C} D_{s,d,c,r} \quad \forall k' \in K', \forall d \in N$$

we also modify the demands from round to round to remove the demands we have already satisfied. For $r > 0$ we have:

$$\forall s, d \in N, \forall c \in C$$

$$D_{s,d,c,r} = \begin{cases} 0 & D_{s,d,c,r-1} = 1, Q_{s,d,c,\max K',r-1} = 1 \\ D_{s,d,c,r-1} & \text{otherwise} \end{cases}$$

Given these new values of D and P we can now add the following to our objective:

$$\text{Distance Objective}(r) = \sum_{\forall k' \in K', \forall s, d \in N: s \neq d} \frac{\gamma}{(k' + 1)(1 + FW_{s,d})} P_{s,d,k',r} + \sum_{\forall k' \in K', \forall s, d \in N: s = d} \frac{1}{(k' + 1)} P_{s,d,k',r}$$

where the second term ensures having the chunk at the destination gives more payoff to the optimization ($\gamma < 1$).

E Number of epochs

We provide a simple algorithm for finding the number of epochs to run the optimization with. This algorithm has no bearing on the optimality of the solution as the optimization automatically identifies if less epochs are sufficient.

Algorithm 1: This algorithm identifies the number of epochs we need to run the optimization with. We use the resulting n_e to instantiate the general optimization — this is an upper bound on the number of epochs we need, and the optimization can automatically discover if a smaller number of epochs is sufficient.

Input: \mathcal{D} . The demand matrix.

Input: $G(N, E)$. The topology.

Input: τ_{opt}

Input: α_{ij} . The latency cost of each link $(i, j) \in E$.

Input: C_{ij} . The capacity of each link $(i, j) \in E$.

Input: C_τ . A set of candidate completion times.

Output: n_e . The upper bound on the number of epochs we need.

```

1 for total_time ∈ C_τ do
2   for n_e ∈ {4, 8, 12} do
3     τ ← total_time / n_e
4     Opt, status ←
5       general_form(ℳ, τ, α, C, n_e, G(N, E))
6     if status = feasible then
7       feasible_time ← total_time
8       break
9   end
10 end
11 n_e ← feasible_time / τ_opt
12 return n_e

```

F Epoch duration set based on the fastest link

To set the epoch duration based on the speed of the fastest link in the LP we do not need to change anything: the LP supports fractional chunks and handles this automatically. The MILP only allows us to send whole chunks — if we set the epoch duration to be lower than the transmission time of the chunk on the slowest link we can never use that link: we need to modify both the flow conservation constraint and the capacity constraints to address this issue.

We can model the flow conservation constraints similar to how we model α : we account for how many epochs it takes a chunk to traverse the slowest link and change the value of δ_{ij} accordingly.

To model the capacity constraint, we need to ensure the number of chunks on a link never exceed its capacity. We first

Collective	(# chunks, #epochs)	SCCL solver time (s)	TE-CCL solver time (s)	Diff in transfer time (%)
ALLGATHER	(1, 2)	0.3	0.09	0
	(2, 3)	0.7	0.07	0
	(3, 4)	1.8	0.19	0
	(4, 5)	4.1	1.45	0
	(5, 6)	11.2	8.96	0
	(6, 7)	27.7	50.57 (11s)	0
ALLTOALL	(1, 3)	8.8	0.11	33%
	(3, 8)	NA	0.18	NA
	(8, 30)	NA	1.88	NA

Table 7: Comparing TE-CCL’s runtime to SCCL. We use 25 KB chunks for these experiments and $\alpha = 0$. The difference in transfer time is $\frac{100(SCCL-TECCL)}{SCCL}$. For all-to-all we use our notation — the number of chunks represents the number of chunks the sender wants to send to each destination (SCCL’s notation uses the number of chunks to mean the total number of chunks the source needs to send).

calculate how many epochs we need to transmit the chunk over a link (κ) and modify the capacity constraints to:

$$\text{Capacity Constraint}(i, j, k) \triangleq \sum_{k-\kappa \leq k' \leq k} \sum_{s \in N} \sum_{c \in C} F_{s,i,j,k',c} \leq \kappa T_{ij} \tau$$

Notice this capacity constraint ensures the same behavior we had when we used the larger epoch duration.

G Comparing to SCCL instance

SCCL has two modes: the `least-steps` and `instance`. We compare TE-CCL to SCCL `instance` in [Table 7](#).

H Details of each topology

We use DGX1, DGX2, NDv2, and internal topologies 1 and 2 for our evaluation. [Figure 11](#) and [Figure 12](#) shows the topologies, capacity and α we used for NDv2 and DGX2 respectively. DGX1 has 8 GPUs and is similar to a single chassis NDv2. Internal topologies 1 and 2 are proprietary, and we cannot report numbers for those.

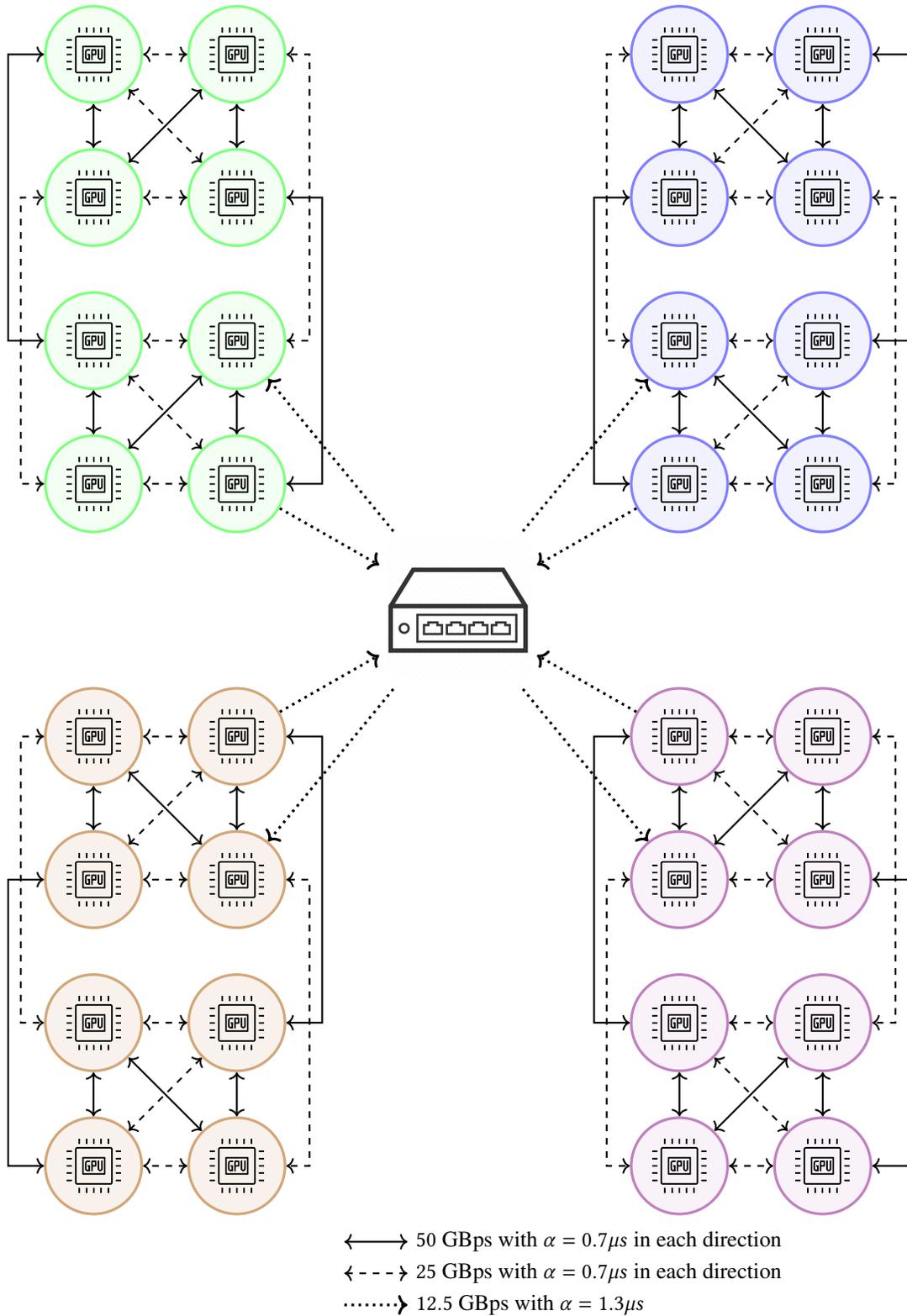


Figure 11: Four chassis NDv2 topology used by TE-CCL. Each chassis has 8 GPUs connected with 50 GBps and 25 GBps links. TACCL replaces the switch by connecting GPU 0 of a chassis to GPU 1 of all other chassis and constraints that only one of the three links can be used at a given time.

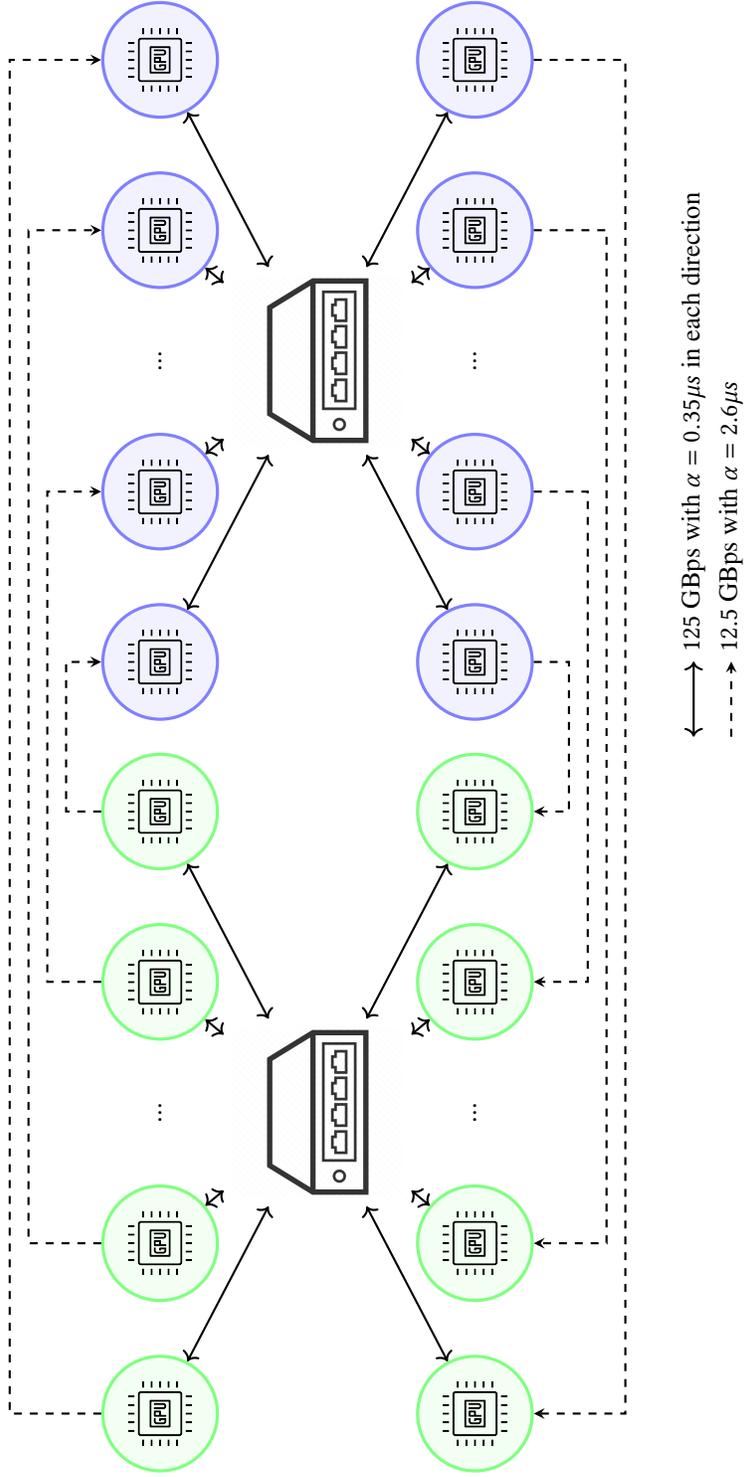


Figure 12: Two chassis DGX2 topology used by TE-CCL. Each chassis has 16 GPUs (8 GPUs are used for sending chunks to another chassis, and 8 GPUs are used for receiving chunks from the other chassis). Each dashed link is 12.5 GBps with $\alpha = 2.6\mu s$, and each thick straight link is 125 GBps with $\alpha = 0.35\mu s$ in each direction. TACCCL replaces the switch in each chassis and connects each GPU in a chassis to every other GPU, effectively forming a clique and uses its uc-min strategy to minimize the number of edges used.

Table 8: Experimental results for TE-CCL and comparison to TACCL on NDv2 2 chassis topology.

Output Buffer Size	ED (μ s)	CT (μ s)	ST (s)	AB (GB/s)	TACCL CT (μ s)	TACCL ST (s)	TACCL AB (GB/s)	Improvement %
ED - Epoch Duration		CT - Collective finish Time			ST- Solver Time			
AB - Algorithmic Bandwidth = output buffer size / collective time								
NDv2 2 chassis ALLTOALL optimal epoch duration								
1 GB	1250	320235.81	336.50	3.123	320049.4	1214.69	3.125	-0.058
256 MB	320	82000.00	307.33	3.122	81964.2	1217.56	3.123	-0.044
64 MB	80	20495.09	339.92	3.123	20532	1220.6	3.117	0.180
16 MB	20	5123.77	280.82	3.123	5164.4	1213.9	3.098	0.793
4 MB	5	1296.25	165.63	3.086	1324.2	1214.51	3.021	2.156
1 MB	1.25	325.28	189.47	3.074	359	1213.52	2.786	10.366
256 KB	0.32	85.52	218.50	2.993	115.72	1221.78	2.212	35.313
64 KB	0.08	23.30	161.99	2.747	50.34	860.88	1.271	116.052
16 KB	0.02	7.27	182.08	2.202	35.76	86.03	0.447	392.223
4 KB	0.02	4.64	69.58	0.862	32.16	31.14	0.125	592.134
1 KB	0.005	4.24	196.72	0.236	36.8	27.66	0.027	768.920
NDv2 2 chassis ALLTOALL max epoch duration								
1 GB	5000	325000	14.82	3.077	320049.400	1214.692	3.125	-1.52
256 MB	1280.41	83226.63	14.36	3.076	81964.200	1217.557	3.123	-1.52
64 MB	320.10	20806.66	11.01	3.076	20532.000	1220.602	3.117	-1.32
16 MB	80.01	5200.42	9.96	3.077	5164.400	1213.903	3.098	-0.69
4 MB	20	1300.03	11.81	3.077	1324.200	1214.507	3.021	1.86
1 MB	5	340	10.85	2.941	359.000	1213.521	2.786	5.59
256 KB	1.28	88.32	9.97	2.899	115.720	1221.779	2.212	31.02
64 KB	0.32	24.32	10.46	2.632	50.340	860.875	1.271	106.99
16 KB	0.08	7.6	8.83	2.105	35.760	86.034	0.447	370.53
4 KB	0.02	4.5	20.90	0.889	32.115	31.139	0.125	613.67
1 KB	0.01	4.235	276.47	0.236	36.799	27.660	0.027	768.92
NDv2 2 chassis ALLGATHER optimal epoch duration								
1 GB	1250	43750	7201.05	22.86	53766.70	7.01	18.60	22.90
256 MB	320	11200	7214.16	22.86	12494.60	6.56	20.49	11.56
64 MB	80	2800	7209.46	22.86	3133.20	8.27	20.43	11.90
16 MB	20	700	7208.70	22.86	-	-	-	-
4 MB	5	190	152.60	21.05	216.50	8.37	18.48	13.95
1 MB	1.25	48.75	160.10	20.51	62.15	62.65	16.09	27.49
256 KB	0.32	14.72	59.55	17.39	25.26	11.17	10.13	71.60
64 KB	0.08	6.08	27.61	10.53	13.08	3.66	4.89	115.13

Table 8 continued from previous page

16 KB	0.02	4.44	18.80	3.60	12.68	6.34	1.26	185.59
4 KB	0.02	4.24	12.26	0.94	11.85	4.30	0.34	179.48
1 KB	0.005	4.135	50.28	0.24	10.16	3.02	0.1	145.68
NDv2 2 chassis ALLGATHER early stop at 30% using optimal epoch duration								
1 GB	1250	47500	2.66	21.05	53766.70	7.01	18.60	13.19
256 MB	320	12163.89	2.37	21.05	12494.60	6.56	20.49	2.72
64 MB	80	3920.31	2.45	16.33	3133.20	8.27	20.43	-20.08
16 MB	20	980.02	2.42	16.33	-	-	-	-
4 MB	5	240	2.40	16.67	216.50	8.37	18.48	-9.79
1 MB	1.25	63.75	4.32	15.69	62.15	62.65	16.09	-2.51
256 KB	0.32	16.96	2.83	15.09	25.26	11.17	10.13	48.94
64 KB	0.08	6.32	3.94	10.13	13.08	3.66	4.89	106.96
16 KB	0.02	4.44	12.98	3.60	12.68	6.34	1.26	185.59
4 KB	0.02	4.24	10.17	0.94	11.85	4.30	0.34	179.48
1 KB	0.005	4.135	42.94	0.24	10.16	3.02	0.1	145.68
NDv2 2 Chassis ALLGATHER max epoch duration								
1 GB	5000	50000	0.94	20	53766.70	7.01	18.60	7.53
256 MB	1280.41	12804.10	0.77	19.99	12494.60	6.56	20.49	-2.42
64 MB	320.10	3201.02	0.78	19.99	3133.20	8.27	20.43	-2.12
16 MB	80.01	800.06	0.77	20	-	-	-	-
4 MB	20	200	0.77	20	216.50	8.37	18.48	8.25
1 MB	5	70	1.04	14.29	62.15	62.65	16.09	-11.21
256 KB	1.28	19.20	1.09	13.33	25.26	11.17	10.13	31.56
64 KB	0.32	7.68	1.74	8.33	13.08	3.66	4.89	70.31
16 KB	0.08	4.80	3.35	3.33	12.68	6.34	1.26	164.17
4 KB	0.02	4.24	21.56	0.94	11.85	4.30	0.34	179.48
1 KB	0.01	4.14	89.07	0.24	10.16	3.02	0.1	145.68