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

Xuting Liu
University of Pennsylvania

Behnaz Arzani
Microsoft Research

Siva Kesava Reddy Kakarla
Microsoft Research

Liangyu Zhao
University of Washington

Vincent Liu
University of Pennsylvania

Miguel Castro*
OpenAI

Srikanth Kandula
Microsoft

Luke Marshall
Microsoft Research

ABSTRACT

Cloud operators utilize collective communication optimizers to enhance the efficiency of the single-tenant, centrally managed training clusters they manage. However, current optimizers struggle to scale for such use cases and often compromise solution quality for scalability. Our solution, TE-CCL, adopts a traffic-engineering-based approach to collective communication. Compared to a state-of-the-art optimizer, TACCL, TE-CCL produced schedules with 2× better performance on topologies TACCL supports (and its solver took a similar amount of time as TACCL's heuristic-based approach). TE-CCL additionally scales to larger topologies than TACCL. On our GPU testbed, TE-CCL outperformed TACCL by 2.14× and RCCL by 3.18× in terms of algorithm bandwidth.

CCS CONCEPTS

• **Networks** → **Network design and planning algorithms**; *Traffic engineering algorithms*; • **Computer systems organization** → *Interconnection architectures.*

KEYWORDS

GPU, Collective Communication, Traffic Engineering

ACM Reference Format:
Xuting Liu, Behnaz Arzani, Siva Kesava Reddy Kakarla, Liangyu Zhao, Vincent Liu, Miguel Castro, Srikanth Kandula, and Luke Marshall. 2024. Rethinking Machine Learning Collective Communication as a Multi-Commodity Flow Problem. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3651890.3672249

1 INTRODUCTION

Collective communication libraries (CCLs) like MSCCL [23], TACCL [33], and Blink [35] optimize data transfers in distributed ML training. They take a topology and a *demand* as input and output a set of routes and a schedule that either maximizes bandwidth utilization, minimizes job completion time, or both, as illustrated in Figure 1. Here, demand is the amount of data each GPU wants to send to other GPUs in the topology, common examples of which include AllToAll, AllGather, and AllReduce.

Operators want these CCLs to produce near-optimal schedules for the collective communication problem to maximize the efficiency of their single-tenant training clusters, as GPUs are expensive and scarce. They use CCLs to support parallelized training jobs (model or data) in data centers with heterogeneous server configurations. They also use CCLs to search for well-provisioned topologies, evaluate hardware architectures, or co-optimize various aspects of distributed training [22, 36, 37] — optimizers guide these explorations.

Traditional, manual collective communication strategies are often wasteful and inefficient. For example, prior work reports the GPUs in BERT [9] and DeepLight [8] spent 11% and 63% of the time idle respectively [33]. Faster GPUs and larger models will only emphasize inefficiencies in these communication schedules.

Recent CCLs attempt to automatically schedule communication to improve performance and utilization. While successful in many cases, they struggle to support the largest of today's training tasks in large cloud providers like Microsoft. These recent CCL solutions, in broad terms, take one of two approaches: precise modeling and heuristic-based modeling.

Optimizers like MSCCL [5] precisely model all aspects of the system and its compute, and they attempt to achieve a near-optimal solution through techniques like SAT solvers. These approaches do not scale beyond one or two chassis and become intractable for typical large distributed training tasks (often at least 30–60 chassis [6]).

In reaction to these scalability concerns, a second class of approaches falls back on increasingly rough heuristics to find the solution more quickly [33, 37]. Unfortunately, these heuristics sacrifice considerable performance to simplify the problem. For example, TACCL [33]'s heuristics separate routing (the path packets take) from scheduling (when each chunk is sent on each link) and underperform hand-crafted solutions by a factor of 2 or more. Similarly, the SPFF schedule in [37] explores only a small subset of the feasibility space, which causes performance gains to fall apart outside of a small set of well-defined topologies.

*This work was performed while at Microsoft.

ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0614-1/24/08
https://doi.org/10.1145/3651890.3672249

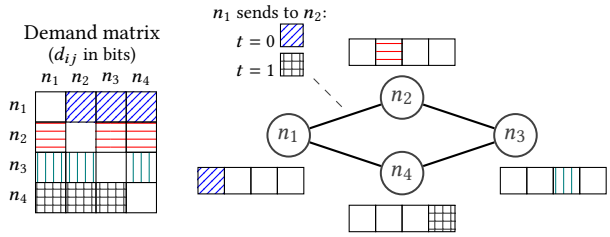


Figure 1: An illustration of the collective communication problem. The input is a demand matrix and capacitated network topology. Links are bi-directional with capacity C bits/s, and, for simplicity, chunks are C bits. We show the allocation on link $n_1 \rightarrow n_2$.

We argue that it was too early to give up on near-optimal solutions for CCLs. We observe that the adjacent field of Traffic Engineering shares striking similarities with the collective communication optimization problem in its inputs, constraints, and objective functions, all while producing near-optimal solutions. Crucially, TE algorithms avoid any sizable performance sacrifices to (already) scale beyond the sizes we require for current training tasks — most recent production WAN TE deployments [20] are approximately the same size as the topologies considered in recent work in collective communication [33]. Other production TE solutions demonstrate utility on networks of thousands of nodes [17, 19].

We acknowledge that there are concrete differences between the two problems, and we do not know which will scale faster in the future; however, our community has successfully deployed TE algorithms at massive production scales [20], which suggests, at a minimum, that today's distributed ML training jobs may benefit from similar lessons. Our system, TE-CCL, proves this is feasible. We show TE-CCL can improve upon state-of-the-art solutions by over 2× on a two-chassis NDv2 topology [3] (Figure 14) and can scale significantly further.

We credit TE-CCL's formulation, scalability, and performance to our ability to borrow ideas from scalable production TE systems. TE solutions give us a model of the *physical problem* (i.e., the flow-conservation constraints and the capacity constraints) and make it easier to reason about a formulation that is otherwise difficult [23, 33]. But the analogy is not exact:

- *Discretized sends:* TE problems mostly focus on traffic bundles with high rates and the problem of allocating a fixed fraction of link capacity to each demand. Instead, CCLs have to *schedule* small- to medium-sized demands, which introduces more structure and adds new and, in some cases, hard-to-model constraints and dependencies.
- *In-network copies:* TE problems often assume flow conservation as a fundamental constraint; in contrast, collectives benefit significantly from copying data at intermediate GPUs, *e.g.,* for tree broadcast/reduce patterns.
- *Latency and queuing:* TE problems get away with focusing on steady-state effects and are able to make fluid-flow assumptions about data delivery because they assume large traffic bundles. In contrast, we cannot ignore the effects of propagation and queuing delay for small transfers; modeling them is essential to CCL scheduling.
- *Support for storage and caching:* TE problems generally assume that data is received and sent as soon as possible [4]; in contrast, as we show in §6, we can speed up solvers substantially if we use the available GPU memory.

Others have explored a subset of these features in isolation in the TE domain (*e.g.,* [17] supports deadlines on fixed-size transfers, [21] allows for store-and-forward, and [11, 27] consider multicast), but we need to reconsider the problem formulation significantly to model the combination.

Our solution, TE-CCL, is a scalable mixed-integer linear program (MILP) with optimality gap guarantees [4]. Similar to other CCLs like TACCL [33] and MSCCL [5], we designed TE-CCL for the scenario where the operator has full control of the infrastructure. The scheduler only needs to run infrequently each time the operator provisions a new workload. However, unlike prior work, TE-CCL scales to much larger collectives than TACCL and MSCCL, and it substantially improves solution quality compared to this prior work. For certain collectives, we can further scale our solution 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, using a technique inspired by the A^* [12] algorithm from robotics.

TE-CCL's solutions outperform the state-of-the-art scalable solution, TACCL [33]. We show *a minimum of* 2× performance improvement on the same 2-chassis NDv2 topology used by TACCL. As part of TE-CCL, we are also able to algorithmically account for multi-tenant and heterogeneous topologies, which are critical for cloud-scale GPU clusters[1].

- We re-examine recent claims that significant approximation is needed to scale CCLs to large distributed ML training tasks and construct TE-CCL to disprove those claims.
- We develop a novel formulation of the CCL problem that supports the full set of features supported by prior CCLs *and also* model network effects more completely.
- We evaluate TE-CCL on topologies from prior work and a large public cloud. We show it scales and improves solution quality by more than 2× in terms of algorithm bandwidth. On our AMD GPU testbed (Figure 15), TE-CCL outperformed TACCL by 2.14× and RCCL by 3.18×.

This work does not raise any ethical issues.

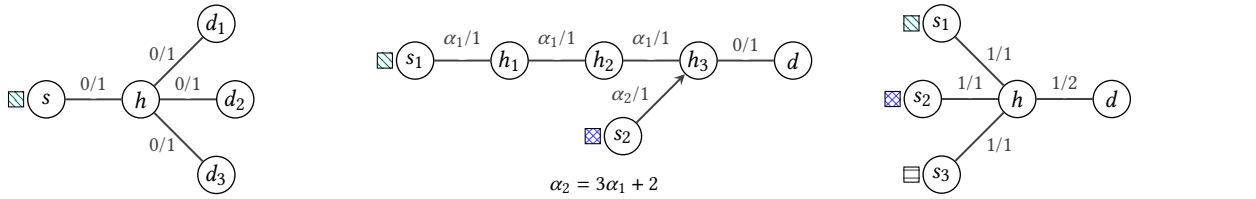
2 BACKGROUND AND MOTIVATION

We present background on collective communication and motivate the need for scalable communication schedules for ML collectives. We then describe TE formulations and how they relate to collective communication optimization.

2.1 The Need for Fast Collective Scheduling

Distributed training jobs that run on multiple GPUs use data parallelism (which aggregates gradients across GPUs) or model parallelism (which aggregates intermediate data across GPUs) to speed

[1]This work does not investigate how to implement the multi-tenant schedule in practice but only provides a formulation that can solve for a multi-tenant demand matrix.



(a) Modeling copies **(b) Proper modeling of delay** **(c) Modeling store and forward**

Figure 2: Examples of three aspects of the CCL problem that are essential for accurate modeling but whose combination is not well-handled by similar formulations. These types of inaccuracies lead solvers to make poor decisions. s_i are the starting location of chunks (shaded boxes), and each chunk is of size 1 MB. d_i are the destinations. Link labels show <*latency*> (in μs) / <*bandwidth*> (in MB/s).

up training. In these settings, Collective Communication Libraries (CCLs) optimize this cross-GPU communication by optimizing routes and schedules for the particular hardware configuration [23, 33].

Collective schedules were traditionally hand-optimized for each new collective and class of topologies, *e.g.,* in or using NCCL [16] — a process that was inaccurate, time-consuming, and error-prone, particularly for large and/or heterogenous deployments.

Recent collective communication optimizers seek automation. To use these libraries, applications specify (1) the "collective," *i.e.,* a relationship between a set of GPU's input and output buffers; (2) the "demand matrix," *i.e.,* the amount of data to be sent between each input buffer and output buffer; (3) the topology, *i.e.,* the connectivity, latency, and capacity of each link; and (4) an objective.

Collective optimizers typically represent their objectives in terms of the "$\alpha - \beta$" cost model where α is the fixed delay data experiences when it goes over a link (which includes propagation and processing delay), and β represents the transmission delay associated with sending 1 bit of data over a link (the inverse of the link capacity). A data chunk of size L thus incurs a cost of $\alpha + L\beta$ when sent over a link. Collectives implicitly model each link's capacity constraints through this notation — when a link has less capacity available, the data takes longer to traverse it.

The output of these optimizers specifies which and how much of the data each GPU should send on each link at each point in time in a way that optimizes the user's objective, respects the network capacity constraints, and results in the correct data in the output buffer at each GPU.

2.2 Relationship with TE Solutions

Traffic Engineering (TE) is a problem in computer networks that arises most frequently in the context of managed Wide Area Networks (WANs), and most recently, those of cloud networks [1, 13, 15, 20, 25].

A primer on TE formulations. Multi-commodity flow problems in TE route specific demands (with a given source and destination) in a way that meets the capacity constraints of the network. Experts usually model TE in one of two forms: a path [13, 15] or edge formulation [4]. We focus on the edge formulation here, but our discussion also applies to the path form. In its most basic format, the edge formulation takes a set of demands D (represented as a matrix matching source and destination nodes (s, d)) and the topology where each link (i, j) has capacity T_{ij} as input, and it outputs how much of each demand should go over each link $F_{s,d,i,j}$. It solves:

$$\text{OptMaxFlow}(N, E, D) \triangleq \arg\max_{\mathbf{F}} \text{Objective}(\mathbf{F})$$
$$\text{s.t.} \quad \mathbf{F} \in \text{FeasibleFlow}(N, E, D)$$

where N and E denote the nodes and the edges in the topology respectively. It adds feasibility constraints (FeasibleFlow) to ensure the network can physically route the traffic and not experience congestion, these are:

(1) *Capacity constraints:* TE formulations model capacity constraints explicitly. The TE capacity constraints ensure the flow the TE optimizer allocates to each link does not exceed the available capacity:

$$\sum_{(s,d)\in D} F_{s,d,i,j} \leq T_{ij}$$

(2) *Flow conservation constraints:* These constraints appear in the edge form of TE and ensure the network does not create traffic "out of thin air" but only routes traffic:

$$\sum_{j|(j,i)\in E} F_{s,d,j,i} = \sum_{j|(i,j)\in E} F_{s,d,i,j} \;\forall i \mid i \neq \text{dest}(s)$$
$$\sum_{j|(j,i)\in E} F_{s,d,j,i} = \sum_{j|(i,j)\in E} F_{s,d,i,j} + D_{s,i} \;\forall i \mid i = \text{dest}(s)$$

where $D_{s,i}$ is the data node i demands from source s.

It may appear that elements of this formulation match the model we laid out in Figure 1, but there are fundamental differences that make this difficult to apply directly.

Discretized sends. Demands in TE request a fixed bandwidth (where the units are in bits per second), which the TE solution assumes is sustained for the entire duration the operator uses the solution. This is true because TE solutions bundle multiple flows between the same nodes into a single demand — statistical averaging then ensures the demand is sustained for longer. Production systems periodically recompute the solution with an updated demand matrix to support shifts in the demand over longer time scales but only at course-grained regular intervals (minutes).

In collectives, on the other hand, demands ask the network to transfer a fixed amount of data (where the units are in bits): once

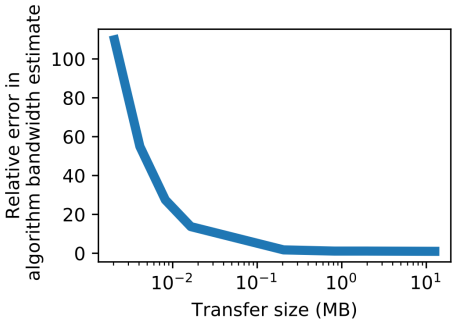


Figure 3: Relative error in the average throughput (output buffer size / total transmission time) of solutions that model latency and queuing compared to solutions that do not. Results are for a proprietary topology from a public cloud with 2 chassis, 8 GPUs, and 40 edges, where the α of intra-chassis and GPU-switch links are 0.6 and 0.75 µs, respectively.

the node sends a chunk of data, it moves on to a different chunk of data if available — capacity frees up over time. In the case of a sender, sender nodes will eventually act as relays and no longer send data of their own for the current instance of the collective.

In-network copies. In TE, the flow conservation constraints ensure the incoming rate to a node is the same as the outgoing rate. These constraints enable traditional TE to avoid the scenario where more traffic comes into a node than what goes out — otherwise, the network will either have large buffers or buffer overflows. It also helps to avoid wasteful solutions, where traffic arrives at a node but is never used.

Many collective demands (*e.g.,* AllGather) consist of sources that send the same data to multiple destinations (multi-cast traffic) and benefit substantially from the ability to copy and send data at intermediate nodes. Figure 2a shows a simple example of the impact of not considering in-network copies, comparing the optimal schedule with and without the capability.

Latency and queuing. TE problems allocate flows to paths[2]. Traditional TE models can make this fluid-flow approximation because they model fixed (and sustained) demands and compute steady-state performance — they do not rely on queuing and do not have to reason about propagation delay.

But we have to model α and the role of interconnect timing in the CCL problem because of the small transfer sizes that are common in collectives [5, 33] (Figure 3 shows how α plays a bigger role in such transfers). If we model delay as the end-to-end transfer time, we introduce significant errors compared to when we model the interconnect behavior. For example, in Figure 2b, the error between these two models is introduced because the chunks from S_1 and S_2 arrive at h_3 at the same time, and, thus, one chunk needs to be queued.

Support for storage and caching. Most TE models do not leverage intermediate nodes' ability to buffer data and, in fact, usually avoid it. This makes sense in TE; switches and routers have shallow buffers compared to transfer sizes, and for stable demand, there is little reason to delay sends.

However, most nodes in a collective topology can buffer sizable amounts of data before sending it out. We show that we can use this to improve solver time as it increases the number (space) of (equivalent) optimal solutions. For example, in Figure 2c, any of the two nodes can begin the schedule and send their chunks to h, which creates 3 possible schedules for the first 1 s. 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 3 s). For some collective demands and topologies, store-and-forward may also improve the transfer time.

[2]This is also true of the edge-form of the problem.

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 ($C = \{0, 1, 2, \ldots, \mathtt{C}\}$). Each node has $\leq \mathtt{C} + 1$ 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 ($K = \{0, 1, 2, \ldots, \mathtt{K}\}$).
$F_{s,i,j,k,(c)}$	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 *start* 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 $(i, j) \in E$.
$\mathrm{R}_{s,d,k}$	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, F and B are integers. We show for some demands, we can use real variables instead in §4.1.

3 THE TE-CCL MODEL

Prior work assumed TE is incompatible with collective communication [33] in part because of the above differences. Our main insight is that the principles TE uses to model communication continue to be valid in CCLs, modulo a few crucial modifications. Drawing an analogy to TE provides a framework for us to retain a direct mapping of constraints between the two problems that (a) supports a principled construction of a solution to the CCL problem and (b) enables operators to later leverage the vast body of research on TE and methods to scale it, *e.g.,* PoP [25] and NCFlow [1].

To that end, in this section we show how we can model the collective communication problem based on ideas that have their roots in how operators solve TE problems [17, 21].

3.1 The General Model

Our notation is in Table 1. Like TE, we need to account for capacity and flow conservation constraints. But before we show how we model these behaviors, we need to introduce a few new concepts that allow us to address the differences we described in §2: chunks, epochs, and buffers.

Unlike TE, collective demands are finite — we need to keep track of where data is at each point in time. We divide the demand into *chunks* to facilitate this. Chunks are contiguous blocks of bytes[3]. We model time through discrete epochs and produce a schedule that tells the user, for each epoch, which chunk to send and where to send it. We assume that each source node has a maximum of $\mathtt{C}+1$ chunks, where we identify each chunk globally through a unique id (s, c), where s is a source node and c is the local id of the chunk at the node. The demand is represented by the function D, where $D_{s,c,d}$ indicates whether destination d wants the chunk with id c from node s. This notation is akin to the multi-commodity flow problem, where each commodity is tracked from its source to its destination to ensure all commodities reach their intended destinations.

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

We need to track each chunk to model copy: 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) in epoch k or is in node i's buffer at the beginning of epoch k respectively.

We also need to use integer variables — we cannot allow chunks to be split into smaller pieces. We use the example in Figure 4 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 sends the second half of the chunk to both d_1 and d_2 but not to d_3. If we use integers for $F_{s,i,j,k,c}$ and $B_{s,i,k,c}$ we can 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 units if needed) — but increasing the number of chunks introduces a trade-off because it 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.

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

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

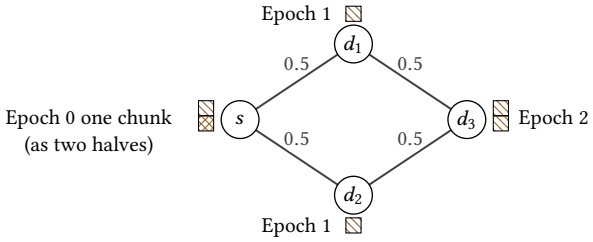


Figure 4: We need integer variables to track each chunk. If we allow partial chunks (▩ and ▨) and copy at the same time, we run into a situation where the optimization can send the same copy of part of a chunk (▨) to two neighboring nodes (in this case d_1 and d_2) and they can forward it along to the 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.

Flow conservation constraints. TE uses flow conservation constraints to ensure the network does not create traffic out of thin air: the sole purpose of these constraints is to ensure a non-source node only consumes or forwards all of the traffic it receives. But these nodes can create traffic in the collective case (they copy data and forward it along multiple outgoing links) and we also have to account for queueing and propagation delay.

To model delay, we need to ensure a node does not forward a chunk if it has not received it. We first compute $\delta_{ij} = \frac{\alpha_{ij}}{\tau}$, the number of epochs a chunk needs 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 $\lceil \delta_{ij} \rceil$ epochs ago.

In-network copies, by definition, violate traditional flow conservation constraints. At the same time, 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:

$$\text{Flow conservation constraints}(s, n, k, c) \triangleq$$
$$B_{s,n,k,c} + \sum_{\forall j \mid (j,n) \in E} F_{s,j,n,k-\lceil \delta_{jn} \rceil,c} \ge \max_{\forall j \mid (n,j) \in E} F_{s,n,j,k+1,c}$$

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:

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

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. Moreover, it is straightforward to model limited buffers 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 as a node does not receive anything in the first epoch and does not send anything in the last. We refer the reader to the Appendix A for details.

Destination constraints. We next need to account for demands to make sure that all demands are met at the end of execution. The constraints are as follows:

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

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 $\mathcal{R}_{s,d,\mathtt{K},c}$ 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 network switches differently because they have limited memory and cannot buffer chunks for appreciable durations. Hence, we set the buffer at each switch to zero.

Traffic pays 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[4].

Most of today's switches support copy through SHARP [10], and so we model switches with this assumption (switches have the same flow conservation constraint as other nodes). Note that we can also model switches without this capability to support legacy hardware. One way is to use traditional TE flow conservation constraints for the switch (what comes into the switch must go out). Another option is to use the approach from TACCL [33]: 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. Due to limited space, we refer the reader to Appendix C for details.

The former two approaches are easier to use in practice: the user does not need to specify a sketch (which is crucial to 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. 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.

[4]Prior work models this behavior incorrectly — they remove the switch and replace it with a direct link but set the α and β based on the uplink only [33].

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

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

The objective gives fewer rewards as k increases: the objective improves if the schedule satisfies the demand as soon as possible. We now have a complete optimization model.

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 run times 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 $O(|V| + |E|)$ time, where V are the vertices and E are the edges of the graph.

4 SCALING

Our model is general and pushes beyond the scale where operators use CCLs today [5, 33]. Extrapolating further, we acknowledge that quickly expanding training cluster topologies may render even the above solutions impractical in the future. We next show two methods to scale TE-CCL.

The first works in situations where in-network copies are not needed (*e.g.,* AllToAll) and preserves optimality. Such collectives only require us to account for the α delay cost but otherwise match the traditional TE formulation. 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 we empirically show it performs well (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 an LP

MILPs are time-consuming, non-convex problems, but there is only one reason we needed integer variables for our model: support for in-network copies. In cases where demands do not benefit from copy (*i.e.,* when each destination wants a unique segment of information from each source), we can change our formulation into a linear program (LP). LPs are convex optimization programs that we can solve in polynomial time and scale better than MILPs.

We remove support for copy and modify the flow conservation constraints back to their traditional form. The following constraint dictates that 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 track duplicates. This reduces the number of variables.

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

The constraints for switches are different: a switch does not consume chunks or buffer them, so we remove those terms.

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

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

Our LP produces a *rate allocation* to demands that originate from each source on each link. From this, we generate a schedule (we translate these rates to paths for each chunk through the same DFS-like solution in §3). This is a straightforward algorithm, 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 A^* [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. This solution is similar to the SPFF schedule in [37] but results in better performance because each optimization covers ("sees") more of the problem and can use more paths. 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, then do so as fast as possible.

We augment our topology with logical links that allow us to compute this reward function. More specifically, 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 [14] 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. 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. The full formulation is in Appendix D.

5 IMPORTANT CONSIDERATIONS

We described how we formulate the CCL problem in TE-CCL. All three formulations (the general MILP form, the LP form, and A^* model) 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 network capacity. To find the best chunk size, we can sweep a range of values to find the best one quickly, take it as an input, or users can also utilize solutions like [22] to pick the optimum for their workflow.

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); or (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. Due to space constraints, we refer the reader to Appendix F for the details. 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 that finds a loose upper bound on how long we need to satisfy all the demands.

We use a feature in optimization solvers (*e.g.,* Gurobi [30]) where they can quickly return some feasible solution. Most solvers quickly find a feasible solution that is also optimal and spend the majority of their time proving optimality [24] (in our experiments, the solver usually found a good solution in the first hour and did not improve it even when we ran for 10 hours). We use binary search with this feature to find the minimum number of epochs we need. We can also use this method to scale TE-CCL further if needed.

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 [10] enabled) and one that does not (the latter is similar to TACCL's hyper-edge model). It is up to the user to decide which is more appropriate for their infrastructure.

Modeling variable bandwidth. Our model supports networks with variable bandwidth. To add support for this, we assume minimal fluctuation within an epoch and change bandwidth only between epochs. We can then take the capacity matrix for each epoch and use that in our capacity constraints.

Use in multi-tenant clusters. TE-CCL 's formulation supports multi-tenancy: 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.

The formulation can be further updated to 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. For efficiency, we may have to use A^* in this case, but doing so would not impact the quality of the solution compared to when we solve a single tenant problem at the same scale.

The above formulation assumes that the schedule can be executed on hardware in a way that continues to fully utilize the bandwidth of a link when necessary. An efficient hardware implementation of a multi-tenant schedule likely introduces additional technical challenges, but we leave these for future work.

Handling stragglers. We note that our formulation also includes coarse-grained mechanisms that help account for stragglers. For instance, we can manage the latency variations that lead to stragglers using the $\alpha - \beta$ cost model. By increasing α, we can address latency variations that are not proportional to the chunk size, such as when computation lags and fails to produce a chunk in time for transmission. Similarly, we can adjust β to reflect latency variations proportional to the data transfer size, such as bandwidth fluctuations due to congestion control behaviors. It is important to note that the β cost applies to each transfer, while α impacts the end-to-end completion time, particularly when disruptions in pipelining occur, causing a link to become idle between transfers.

6 EVALUATION

We implement our solution in Python[5]. We use Gurobi [30] to solve the optimizations. We convert our solution into MSCCL [5], which can then port it into a schedule that runs on the hardware.

[5]Link to code: https://github.com/microsoft/TE-CCL

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
AMD	16	56

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

The goal of this evaluation is to (1) compare TE-CCL to state-of-the-art, both in scale and in terms of solution quality; (2) show TE-CCL scales to the large topologies; and (3) show the impact of each of our different design choices.

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

- *Solver time:* This includes the time to set up the variables and constraints in the solver.
- *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 [33]).
- *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.
- *Algorithm bandwidth:* The output buffer size divided by the transfer time, a metric from NCCL [26].

Topologies and workloads. We evaluate TE-CCL using the topologies in Table 2. We use common topologies such as DGX1, DGX2 [28], NDv2 [3], and AMD [2], as well as two next-generation (not in production yet) proprietary topologies from a public cloud provider. We evaluate with a range of data sizes and use profiled values for α and β, similar to TACCL [33]. Collective communication operations follow the standard sizes and groupings defined in NCCL [16].

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.

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× to avoid large models (since α dominates, this does not materially impact the solution).

TE-CCL solves optimization problems to produce a schedule. The optimization is deterministic and outputs the same solution 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 [33] and MSCCL[6] [5].

[6]SCCL was renamed to MSCCL.

Collective, #chunks	MSCCL (μs)	TE-CCL (μs)	Pipelining Possible
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 MSCCL `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.

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 cannot 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 relative to the finest granularity.

TACCL. We obtained the TACCL code from their public GitHub repository [34] 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.

MSCCL. We compare to MSCCL using the public MSCCL codebase [23] and also re-ran our experiments using the MSCCL artifact from their submission (which the authors gave us). We verified and confirmed with the authors that we used MSCCL correctly and that our numbers are correct.

Platform. We use the solvers and the schedules they produce to compute the transfer times and algorithm bandwidth for MSCCL, TACCL, and TE-CCL. We checked using AMD nodes that these estimates match what we get from running on hardware for both TE-CCL and TACCL. Most of the results we report in the paper are based on this approach (*i.e.,* we use the schedule the solver produces, the transfer times, and algorithm bandwidth to compute the total time it takes for the schedule to finish). We also report results on a small AMD test-bed that shows these results hold on real hardware.

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

6.1 Comparison to MSCCL and TACCL

MSCCL. MSCCL 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 MSCCL `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 MSCCL paper does not evaluate this mode). 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 $\leq 0.65s$ for all AllGather demands and $\leq 0.97s$ for AllToAll with less than 5 chunks.

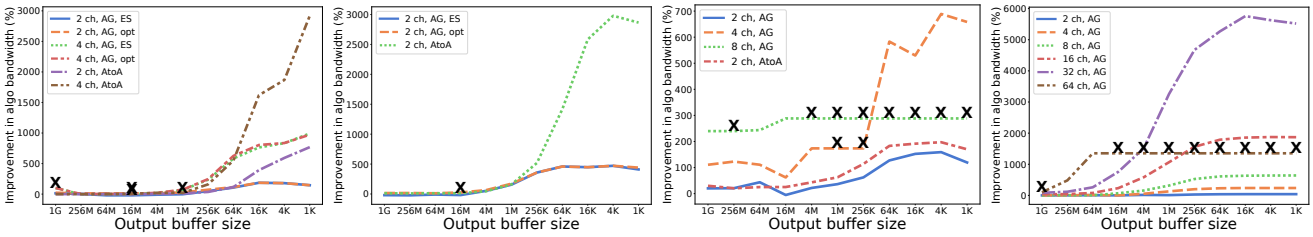
We used $25KB$ chunks to capture the impact of α ($\alpha = 0.7\mu$s) on the solutions (Table 3): for all > 1 chunk cases TE-CCL outperforms. This is because our TE-based formulation models pipelining explicitly and ensures a node receives a chunk before forwarding it; MSCCL enforces a barrier instead. MSCCL performs better in the 1 chunk case as TE-CCL cannot leverage its ability to pipeline.

We also compare with MSCCL'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 MSCCL for K in TE-CCL — since MSCCL is no longer running an optimization — and use $\alpha = 0$ (this is necessary as our model will need more epochs otherwise to account for α). We use the scenarios from Table 4 in MSCCL [5] and run both solvers on a desktop with 6 cores and 32 GB RAM. MSCCL failed to produce a solution for AllGather workloads with more than 1 chunk even after 3 days. TE-CCL runs faster than MSCCL in almost all cases and even improves MSCCL's solution quality by 33% in the AllToAll scenario. TE-CCL is slower than MSCCL 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).

To fully highlight our runtime advantage over MSCCL, we ran an AllToAll demand with 8 chunks using both solvers: MSCCL 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 21μs (for 25KB chunks).

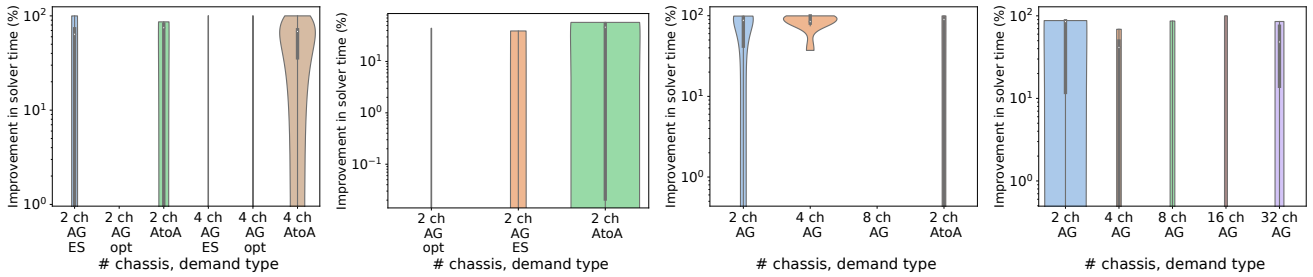
TACCL. We compare the solver time and algorithm bandwidth of TE-CCL and TACCL using AllGather and AllToAll demands and on DGX2 and NDv2 based topologies with up to 34 nodes (a 2-chassis DGX2 topology has 34 nodes) and on both internal topologies with up to 128 nodes. We ran all experiments on a Linux 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.



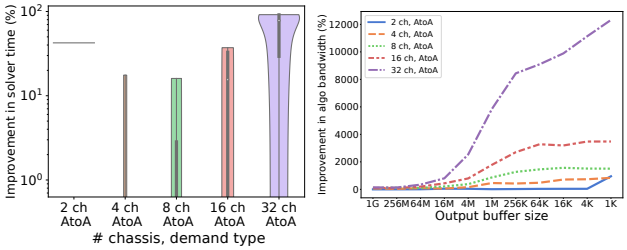
(a) NDv2 (b) DGX2 (c) Internal 1 (d) Internal 2

Figure 5: Compares the algorithm 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. AG stands for AllGather, AtoA AllToAll, and ES early stop. The X-axis is the output buffer size starting with 1GB and ending in 1KB.



(a) NDv2 (b) DGX2 (c) Internal 1 (d) Internal 2

Figure 6: Compares the solver time of TE-CCL and TACCL ($\frac{100(TACCL-TECCL)}{TACCL}$). Ch stands for chassis. 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 50% of AllGather for Internal 2. We observe that TE-CCL solver-time is on par with TACCL, but in general, we expect TACCL to be faster because it is a heuristic and gives up on quality for speed (in scenarios where we do not plot a line TE-CCL is slower than TACCL).



(a) Solver time (b) Transfer time

Figure 7: We compare TACCL and TE-CCL for AllToAll demands on the Internal 2 topology. TE-CCL produces equivalent or higher quality solutions in all cases with similar solver times.

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 5 and Figure 7 (we report AllToAll numbers for Internal 2 separately for clarity). We report the raw algorithm bandwidths for TE-CCL variants in the appendix (see Table 8) for NDv2 2 chassis as a sample.

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 6.

6.2 AllGather on AMD

We evaluate TE-CCL on a two chassis (32 GPU) AMD topology. TE-CCL outperforms RCCL's ring-based algorithm and TACCL (Figure 8) for most transfer sizes: its solutions are 3× faster than RCCL for 1MB transfers and 1.5-2× faster for larger transfers. Our

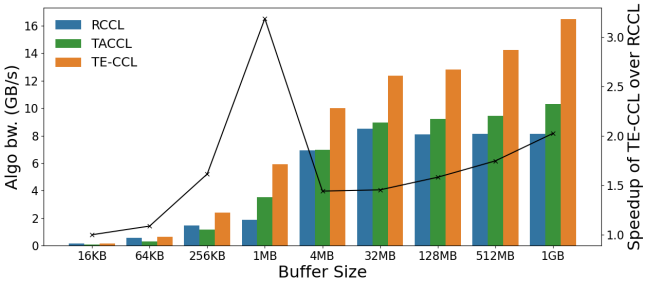


Figure 8: AllGather comparisons of RCCL, TACCL, and TE-CCL on two AMD nodes.

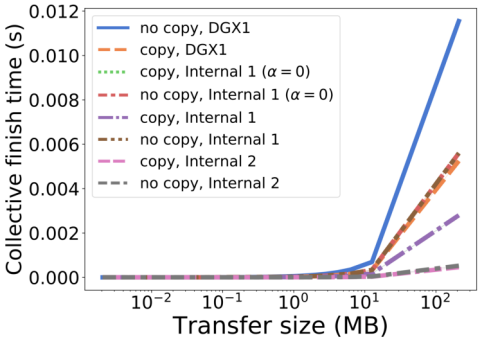


Figure 9: The benefit of copy. For large transfers, a copy helps finish the transfer faster.

results are based on ROCm6 [32]. We found TE-CCL outperformed RCCL on ROCm5.7 by a larger margin for small transfers, but RCCL improved their manually constructed schedules for small transfers in their recent update and reduced the gap.

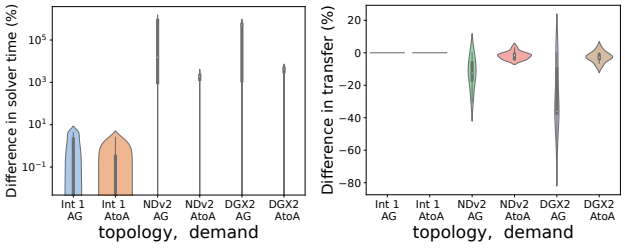
6.3 Scale

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 solver memory. 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.

6.4 Microbenchmarks

We next evaluate our design choices:

Copy. As shown in Figure 9, 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 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.



(a) Solver time **(b) Transfer time**

Figure 10: 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.

Small vs large epochs (Figure 10 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.

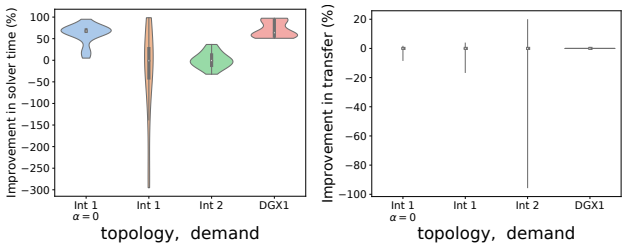
Store and forward. We find a somewhat surprising result: buffers do not impact the solution quality but only the solver time (Figure 11)! This is because of the nature of collective demands such as AllGather and AllToAll. Because each node needs the same amount of traffic as it has to forward, it can 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 improvement 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.** 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 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 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 [11, 27]. Oliveira and Pardalos [29] provide a



(a) Solver time **(b) Transfer time**

Figure 11: We evaluate the impact of buffers on (a) solver time and (b) solution quality. We use 2 chassis for all topologies. Both graphs compute $\frac{100(\text{without buffers} - \text{with buffers})}{\text{without buffers}}$. Buffers do not impact the solution quality but only the solver times. The average improvements in solver time are: 61%, −28.46%, 0.23%, 71% for Internal 1 without α, Internal 1 with α, Internal 2, and DGX1 respectively.

comprehensive summary of these works. Blink [35] 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, 13, 15, 25]. Most of these works assume sustained demands. Among these works, Calendaring [17] provides a solution that models unsustained demands. NetStitcher [21] 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, 18, 31, 33, 35, 37]. 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 [22, 36, 37] aims to co-optimize either topologies and parallelization strategies ([36]) or collective scheduling and execution planning [22]. 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 LIMITATIONS AND FUTURE WORK

Handling failures. In our setting, we employ a Clos-based topology with Equal Cost Multipath (ECMP) path redundancy. This allows the network to naturally adapt to failures of inter-chassis links and switches. We simplify this in our model by replacing the detailed topology with a single big-switch abstraction, ignoring the internal topology of the Clos. Handling failures within the chassis is more challenging. Fortunately, these are less common. We defer robust handing of such failures to future work.

Extension to public clouds. Our solution is designed to enable operators who have full control over their training clusters and are knowledgeable about the training workloads to optimize their infrastructure. There are several challenges that arise when attempting to export our schedule to scenarios where the tenant does not control the infrastructure, such as cloud customers. In such multi-tenant environments, the user may not be aware of the topology or the α and β values of each link. Prior work [5, 33, 37] proposes profilers to capture these values. Although our solution can produce a schedule based on the numbers generated by these profilers, aligning with the same abstractions as these prior works, we anticipate that the true values of α and β will be unstable in a shared network, thereby preventing these works from producing a good schedule. We defer extending our solution to such scenarios to future work.

AllReduce implementation. TE-CCL supports AllReduce implicitly through the combination of AllToAll and AllGather operations. We can also directly solve for the AllReduce workload by utilizing multiple demand matrices, each representing an intermediate stage of the operation. However, we acknowledge that our model does not account for the compute cost in this case, and we plan to address this in future work.

Lowering to hardware. In TE-CCL, we match the chunk abstraction and the cost-model (α-β) from prior work and rely on their observations to ensure the schedule we produce can run on hardware. We do not account for any additional hardware constraints, such as the number of channels or the number of thread blocks required to run these schedules on hardware. MSCCLang [7] covers many of the nuances involved in deploying such custom schedules on hardware. In our hardware experiments, we hand-optimized the implementation of the TE-CCL schedule (through the XML we provided to MSCCL) and ran it with the same number of channels as the schedules in RCCL and TACCL to produce fair comparisons.

9 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 MSCCL and TACCL by over 2×.

ACKNOWLEDGMENTS

We gratefully acknowledge our shepherd Praveen Kumar and the anonymous SIGCOMM reviewers for all of their thoughtful reviews, comments, and time. Thanks to Saeed Maleki, Aashaka Shah, Madanlal Musuvathi, Ilias Marinos, and Sadjad Fouladi for their feedback and help with running TACCL and understanding RCCL/NCCL nuances. This work was funded in part by NSF grant CNS-2107147.

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] AMD 2023. AMD Instinct MI250 GPUs (slide 18). https://hc34.hotchips.org/assets/program/conference/day1/GPU%20HPC/HC2022.AMD.AlanSmith.v14.Final.20220820.pdf

[3] Azure NDv2-series 2022. Azure NDv2-series. https://learn.microsoft.com/en-us/azure/virtual-machines/ndv2-series

[4] Dimitris Bertsimas and John N Tsitsiklis. 1997. *Introduction to linear optimization*. Vol. 6. Athena Scientific Belmont, MA.

[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* (Virtual Event, Republic of Korea) *(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. ChatGPT runs 10K Nvidia training GPUs with potential for thousands more. https://www.fierceelectronics.com/sensors/chatgpt-runs-10k-nvidia-training-gpus-potential-thousands-more

[7] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. 2023. MSCCLang: Microsoft Collective Communication Language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 502–514. https://doi.org/10.1145/3575693.3575724

[8] 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* (Virtual Event, Israel) *(WSDM '21)*. Association for Computing Machinery, New York, NY, USA, 922–930. https://doi.org/10.1145/3437963.3441727

[9] 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).

[10] John Matthew Simon Doar. 1993. *Multicast in the asynchronous transfer mode environment*. Technical Report. University of Cambridge, Computer Laboratory.

[11] 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

[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] 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* (Hong Kong, China) *(SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2486001.2486012

[14] Stefan Hougardy. 2010. The Floyd–Warshall algorithm on graphs with negative cycles. *Inform. Process. Lett.* 110, 8-9 (2010), 279–281.

[15] 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.

[16] Sylvain Jeaugey. 2017. NCCL 2.0. In *GPU Technology Conference (GTC)*, Vol. 2.

[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* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 515–526. https://doi.org/10.1145/2619239.2626336

[18] Heehoon Kim, Junyeol Ryu, and Jaejin Lee. 2024. 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 (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 999–1015. https://doi.org/10.1145/3620666.3651362

[19] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. 2022. Decentralized cloud wide-area network traffic engineering with BLASTSHIELD. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 325–338.

[20] Umesh Krishnaswamy, Rachee Singh, Paul Mattes, Paul-Andre C Bissonnette, Nikolaj Bjørner, Zahira Nasrin, Sonal Kothari, Prabhakar Reddy, John Abeln, Srikanth Kandula, et al. 2023. OneWAN is better than two: Unifying a split WAN architecture. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 515–529.

[21] Nikolaos Laoutaris, Michael Sirivianos, Xiaoyuan Yang, and Pablo Rodriguez. 2011. Inter-Datacenter Bulk Transfers with Netstitcher. In *Proceedings of the ACM SIGCOMM 2011 Conference* (Toronto, Ontario, Canada) *(SIGCOMM '11)*. Association for Computing Machinery, New York, NY, USA, 74–85. https://doi.org/10.1145/2018436.2018446

[22] Kshiteej Mahajan, Ching-Hsiang Chu, Srinivas Sridharan, and Aditya Akella. 2023. Better Together: Jointly Optimizing ML Collective Scheduling and Execution Planning using SYNDICATE. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 809–824. https://www.usenix.org/conference/nsdi23/presentation/mahajan

[23] MSCCL codebase 2023. MSCCL codebase. https://github.com/microsoft/msccl

[24] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, and Srikanth Kandula. 2022. Minding the gap between fast heuristics and their optimal counterparts. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. 138–144.

[25] Deepak Narayanan, Fiodar Kazhamiaka, 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* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 521–537. https://doi.org/10.1145/3477132.3483588

[26] NCCL Tests 2021. NCCL Tests. https://github.com/NVIDIA/nccl-tests/blob/d028efcf35101c6663ae8c5f33ad41bad00efb4d/doc/PERFORMANCE.md#algorithm-bandwidth

[27] 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

[28] Nvidia DGX System 2023. Nvidia DGX System. https://www.nvidia.com/en-us/data-center/dgx-systems/

[29] 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.

[30] Joo Pedro Pedroso. 2011. Optimization with gurobi and python. *INESC Porto and Universidade do Porto,, Porto, Portugal* 1 (2011).

[31] 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* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 581–596. https://doi.org/10.1145/3470496.3527382

[32] ROCm 2024. AMD ROCm Software. https://www.amd.com/en/products/software/rocm.html

[33] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. 2023. TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 593–612. https://www.usenix.org/conference/nsdi23/presentation/shah

[34] TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches 2023. TACCL codebase. https://github.com/microsoft/taccl

[35] 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.

[36] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. 2023. TopoOpt: Co-optimizing Network Topology and Parallelization Strategy for Distributed Training Jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 739–767. https://www.usenix.org/conference/nsdi23/presentation/wang-weiyang

[37] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. 2022. Optimal Direct-Connect Topologies for Collective Communications. https://doi.org/10.48550/ARXIV.2202.03356

APPENDIX

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

A INITIALIZATION AND TERMINATION CONSTRAINTS

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

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:

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

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

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

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:

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

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:

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

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 a 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:

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

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

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

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.

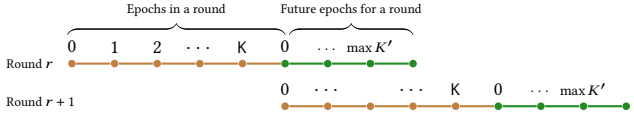


Figure 12: A^* time progression between rounds

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 12). 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 rounds. For

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 ($\Omega : S \rightarrow 2^{N-S \times N-S}$). $\Omega(s) = \{(i,j) \mid (i,s) \in E \wedge (s,j) \in E \wedge (i,j) \notin E\}$
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 ($R = \{0, 1, 2, \ldots \mathsf{R}\}$)
K	The set of epochs in a round ($K = \{0, 1, 2, \ldots, \mathsf{K}\}$). 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 ($K' = \{0, 1, 2, \ldots, \max_{\forall (i,j) \in E} \lceil \frac{\alpha_{i,j}}{\tau} \rceil\}$)
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.

switches, we need to modify the flow conservation constraints because they do not buffer chunks.

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:

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

and for all later arrivals we have:

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

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 $B_{s,n,c,\mathsf{K},r}$ 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:

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

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:

$$\forall s, n \in N : n \notin S, \forall c \in C, \forall k \in K : k > 0$$
$$B_{s,n,c,k,r} = B_{s,n,c,k-1,r} + \sum_{\forall j:(j,n) \in E} F_{s,j,n,c,k-\lceil \frac{\alpha_{j,n}}{\tau} \rceil-1,r} + Q_{s,n,c,k,r-1}$$

otherwise:

$$\forall s, n \in N : n \notin S, \forall c \in C, \forall k \in K : k > 0$$
$$B_{s,n,c,k,r} = B_{s,n,c,k-1} + \sum_{\forall j:(j,n) \in E} F_{s,j,n,c,k-\lceil \frac{\alpha_{j,n}}{\tau} \rceil-1}$$

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:

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

but since switches do not buffer chunks we incorporate them into the flow conservation constraints.

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.

```
for total_time ∈ C_τ do
    for n_e ∈ {4, 8, 12} do
        τ ← total_time / n_e
        Opt, status ← general_form(D, τ, α, C, n_e, G(N, E))
        if status = feasible then
            feasible_time ← total_time
            break
n_e ← feasible_time / τ_opt
return n_e
```

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 [14] 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.

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 constraints 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 exceeds its capacity. We first 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 13, Figure 14, and Figure 15 shows the topologies, capacity and α we used for DGX2, NDv2 and AMD 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.

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).

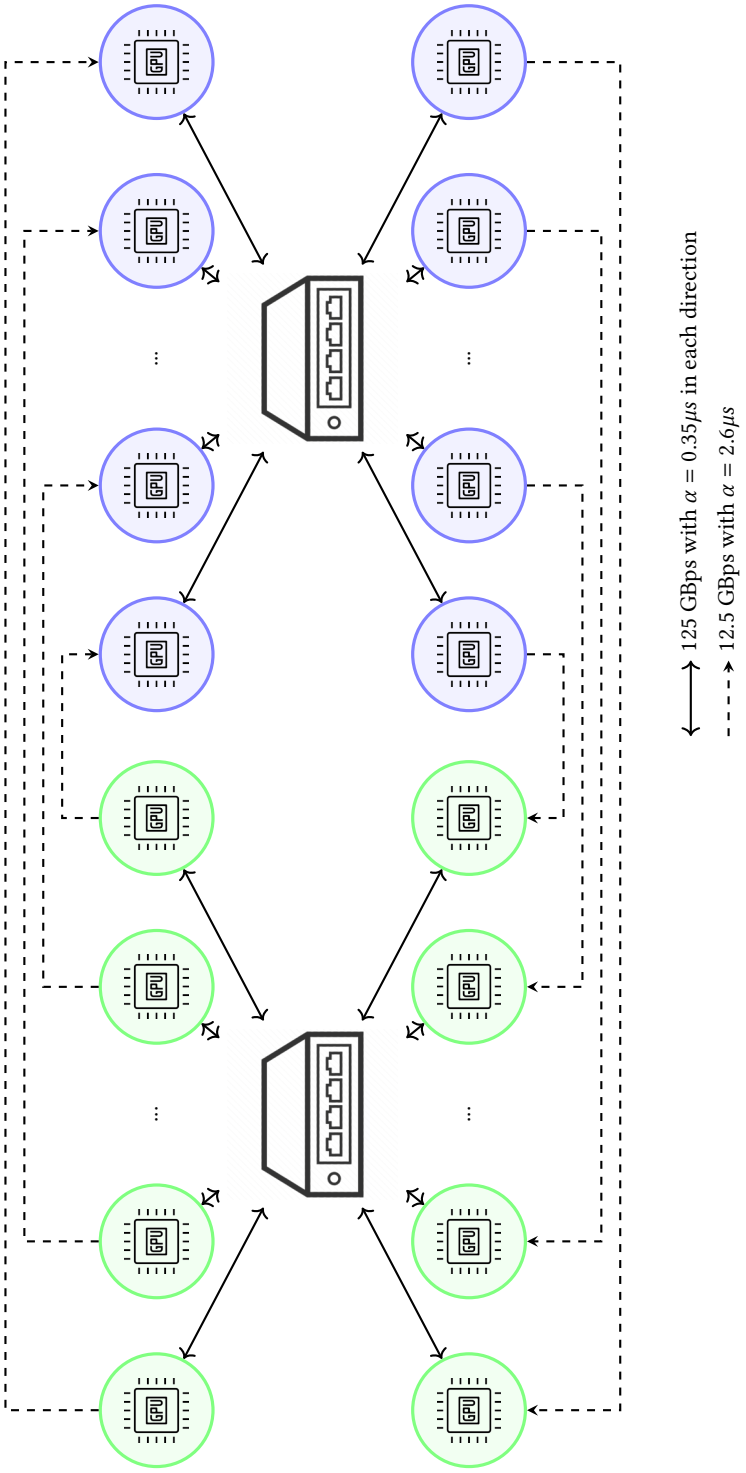


Figure 13: Two chassis DGX2 topology used by TE-CCL. Each chassis has 16 GPUs (8 GPUs are used for sending chunks to another chassis, and 8 GPUs are used for receiving chunks from the other chassis). Each dashed link is 12.5 GBps with $\alpha = 2.6\mu s$, and each thick straight link is 125 GBps with $\alpha = 0.35\mu s$ in each direction. TACCL 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.

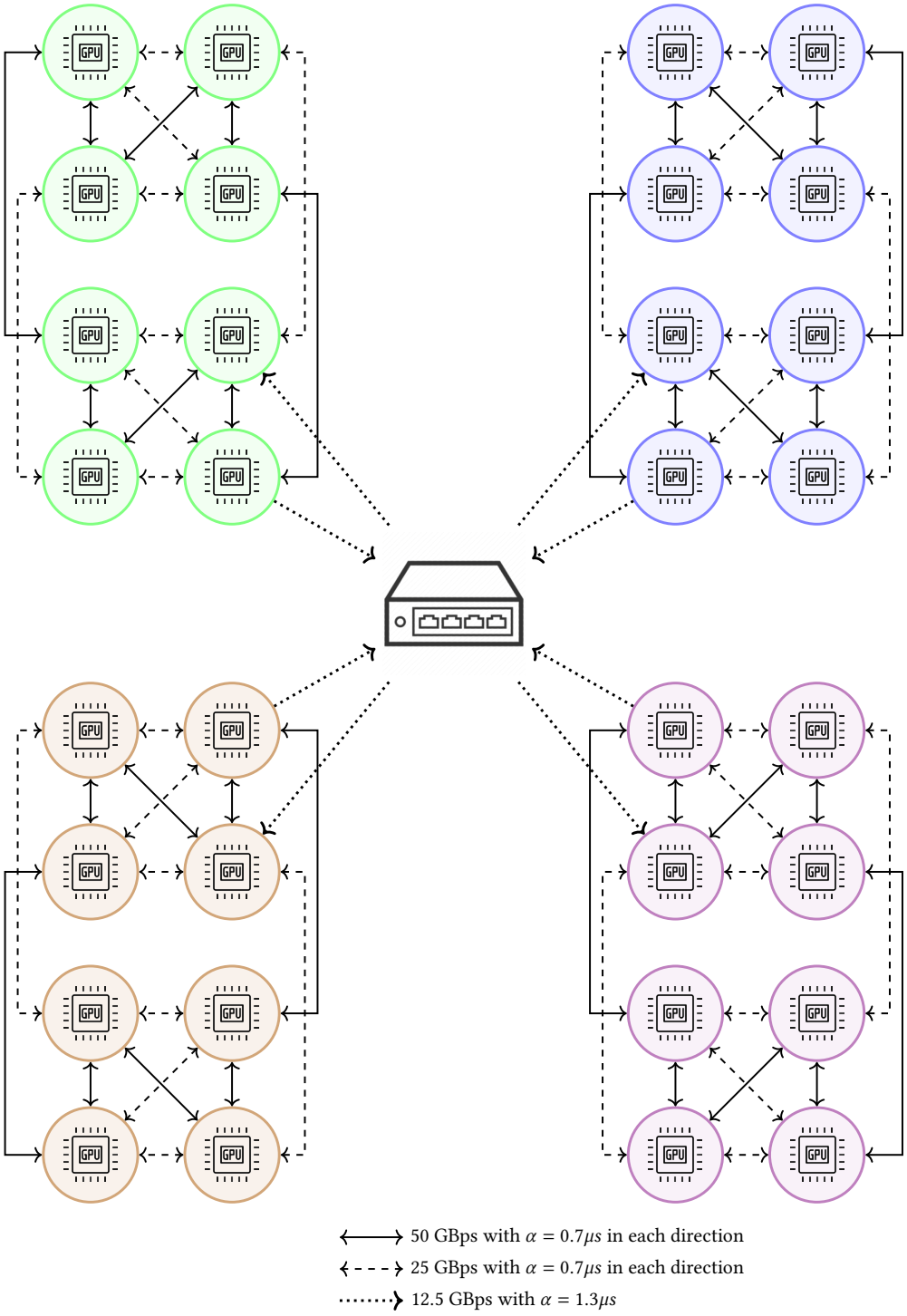


Figure 14: Four chassis NDv2 topology used by TE-CCL. Each chassis has 8 GPUs connected with 50 GBps and 25 GBps links. TACCL replaces the switch by connecting GPU 0 of a chassis to GPU 1 of all other chassis and constraints that only one of the three links can be used at a given time.

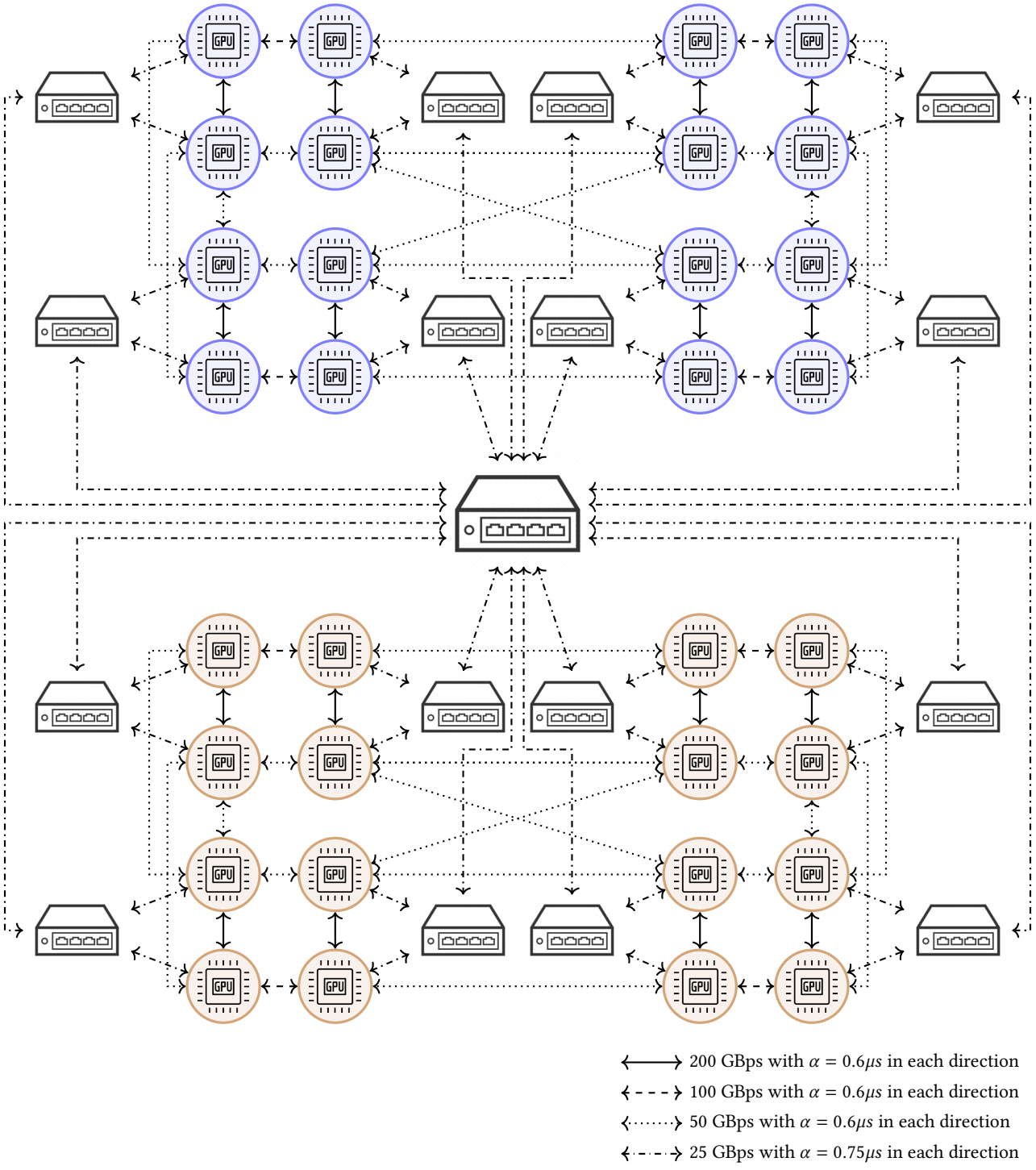


Figure 15: Two chassis AMD topology used by TE-CCL. Each chassis has 16 GPUs connected with 200 GBps, 100 GBps and 50 GBps links. GPU pairs are connected to small switches in each chassis, which are all connected to a bigger switch. All switch links are 25 GBps.

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

Output Buffer Size	ED (μs)	CT (μs)	ST (s)	**AB (GB/s)**	TACCL CT (μs)	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**

Table 8 continued from previous page

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**
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**