TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches

Aashaka Shah[*]
University of Texas at Austin

Vijay Chidambaram
University of Texas at Austin and VMware Research

Meghan Cowan
Microsoft Research

Saeed Maleki
Microsoft Research

Madan Musuvathi
Microsoft Research

Todd Mytkowicz
Microsoft Research

Jacob Nelson
Microsoft Research

Olli Saarikivi
Microsoft Research

Rachee Singh
Microsoft and Cornell University

Abstract

Machine learning models are increasingly being trained across multiple GPUs and servers. In this setting, data is transferred between GPUs using communication collectives such as ALLTOALL and ALLREDUCE, which can become a significant bottleneck in training large models. Thus, it is important to use efficient algorithms for collective communication. We develop TACCL, a tool that enables algorithm designers to guide a synthesizer into automatically generating algorithms for a given hardware configuration and communication collective. TACCL uses a novel *communication sketch* abstraction to get crucial information from the designer to significantly reduce the search space and guide the synthesizer towards better algorithms. TACCL also uses a novel encoding of the problem that allows it to scale beyond single-node topologies. We use TACCL to synthesize algorithms for three collectives and two hardware topologies: DGX-2 and NDv2. We demonstrate that the algorithms synthesized by TACCL outperform the Nvidia Collective Communication Library (NCCL) by up to 6.7×. We also show that TACCL can speed up end-to-end training of Transformer-XL and BERT models by 11%–2.3× for different batch sizes.

1 Introduction

Machine-learning models have been dramatically increasing in size over the past few years. For example, the language model MT-NLG has 530 billion parameters [31] and the Switch-C mixture-of-experts model has 1.6 trillion parameters [18]. Model sizes are expected to further grow to increase model accuracy and perform more complex tasks. These models are too large for the resources of a single GPU and have to be distributed across multiple servers, each with several GPUs, using different parallelism strategies like data, model, pipeline, and expert parallelism [18, 27, 43] for training and inference. Intermediate data and parameters of the model at each GPU are accumulated, shuffled, and transferred over the network between other GPUs for distributed machine learning, depending on the type of parallelism strategy used.

[*]Work was partially done during an internship at Microsoft Research.

The inter-GPU communication bottleneck. Recent work has shown that GPU idle time spent waiting for network communication can be significant in practice [2, 19, 26, 28]. For instance, BERT [15] and DeepLight [14] spent 11% and 63% of time, respectively, with GPUs idle on a 100 Gbps Ethernet cluster of P100 GPUs [2]. Newer generations of faster GPUs will only make this problem worse. This inefficient use of GPUs shows that there is significant model performance to be gained by optimizing inter-GPU communication.

Collective communication primitives and algorithms. Efficient communication between GPUs is the key to enabling fast distributed ML training and inference. Modern GPU systems use message passing interface (MPI)-based *collective communication primitives*, such as ALLREDUCE, ALLGATHER, and ALLTOALL to perform inter-GPU communication (Figure 2 in §2). *Collective algorithms* implement collective communication primitives. They route data along various paths in the network and schedule the necessary computation (e.g., a sum in ALLREDUCE) while optimizing for latency and bandwidth characteristics of each link in the network. For example, a common collective algorithm for ALLGATHER (all GPUs gather data from all GPUs) is a Ring algorithm, in which all GPUs are logically arranged in a ring and each GPU receives data from its predecessor in the ring and sends a previously received data to its successor. Inefficiencies in collective communication algorithms can cause poor network utilization, causing GPUs to remain idle until inter-GPU transfers complete [53], and thus reducing the overall efficiency of distributed training and inference.

Challenges in designing GPU communication algorithms. Designing algorithms for efficient collective communication on GPU topologies is challenging. First, these algorithms have to strike the right balance between latency and bandwidth optimality. For instance, the commonly used Ring algorithm for ALLREDUCE is not efficient for small input sizes as it has a high latency. Second, GPU communication algorithms have to manage the heterogeneity of connectivity in the underlying topology. For instance, GPUs within a machine (also referred to as a node) are usually connected using fast NVLinks [38] (up to 300 GBps aggregate bidirectional bandwidth per GPU) while GPUs across nodes are connected using slow Infini-

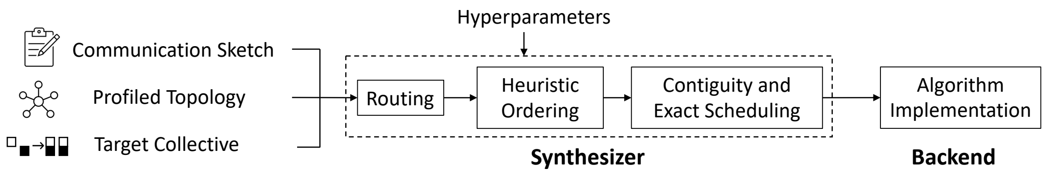


Figure 1: TACCL's novel synthesizer takes as input a communication sketch, profiled topology, and target collective along with synthesizer hyperparameters to generate an algorithm for the collective. The synthesized algorithm is implemented on the hardware cluster using TACCL's backend.

Band [36] links (12.5-25 GBps per NIC). Moreover, these topologies vary significantly between vendors. And finally, searching over the entire space of routing and scheduling algorithms to find optimal ones for communication collectives is computationally prohibitive. In fact, previous approaches that synthesize collective communication algorithms are limited to single-node topologies [9] or 8 GPUs at most [51].

Managing scale for automated algorithm design. Our goal is to automatically obtain efficient algorithms for a given hardware configuration and communication collective. We encode the problem of finding optimal algorithms for communication collectives into a mixed integer linear program (MILP) with the goal of minimizing the overall execution time. Unfortunately, this problem is NP-hard; state-of-the-art commercial solvers like Gurobi [20] can spend several days exploring the search space without finding an optimal algorithm. In this work, we propose a *human-in-the-loop* approach that incorporates high-level inputs of an algorithm designer to efficiently synthesize collective communication algorithms for heterogeneous GPU topologies. We argue that it is easy for algorithm designers to provide a few simple inputs that constrain the search space of algorithms which allows synthesis engines to scale to large GPU topologies.

Communication sketches as user input. It is crucial that the input required from algorithm designers is simple and intuitive. For this, we introduce a new abstraction: *communication sketches* (§3). Inspired by the technique of *program sketching* [47] from program synthesis, in which developers supply a partially specified program with holes that capture the high level structure of the desired program, communication sketches allow algorithm designers to provide high-level intuitions that constrain the search space of algorithms. A synthesis engine fills in the remaining details such as routing and scheduling of the final collective communication algorithm, analogous to how a constraint solver in program synthesis searches the reduced space to fill the holes.

Our solution. We develop TACCL (Topology Aware Collective Communication Library), a system that synthesizes communication algorithms for a given topology and a target collective communication primitive. Algorithm designers can use communication sketches to guide TACCL into synthesizing efficient algorithms for a large range of hardware topologies. We develop a novel encoding of the problem in TACCL's synthesizer to scale beyond single-node topologies. Figure 1 shows an overview of TACCL's design.

Synthesizing algorithms from communication sketches. TACCL's synthesis approach builds on the solver based synthesis approach in SCCL [9], where the space of possible algorithms is directly encoded in a *satisfiability modulo-theories* (SMT) solver. SCCL does not scale to the sizes of clusters used by modern machine learning workloads. We present a novel *mixed integer linear programming* (MILP) encoding of the collective algorithm synthesis problem that improves scalability by first solving a bandwidth-relaxed version of the problem to decide on *routing*, followed by ordering heuristics and a second bandwidth-constrained problem to find a valid *scheduling* (§5). In addition to improving scalability, TACCL's MILP formulation allows modeling of heterogeneous links with different per-message overhead characteristics. This overcomes the limitation in SCCL [9] that prevents it from faithfully targeting distributed GPU clusters.

Results. We use TACCL to synthesize efficient algorithms for a range of collectives like ALLGATHER, ALLTOALL, and ALLREDUCE, and for different hardware backends like Azure NDv2 [6] and Nvidia DGX-2 [35] (§7). We compare TACCL to the state-of-the-art Nvidia Collective Communication Library (NCCL). TACCL synthesized an ALLGATHER algorithm for two Nvidia DGX-2 nodes (32 GPUs). This algorithm is up-to 6.7× faster than NCCL for small-to-moderate input sizes. For large input sizes on the same hardware, TACCL synthesized a different ALLGATHER algorithm that nearly saturates the inter-node bandwidth and is up-to 25% faster than NCCL. TACCL synthesized an ALLTOALL algorithm for two Azure NDv2 nodes (16 GPUs) that is up-to 66% faster than NCCL. Finally, we replaced NCCL with TACCL using only a two-line code change in PyTorch and found that TACCL achieves a speed-up of 17% in end-to-end training of a mixture-of-experts model that uses ALLTOALL and ALLREDUCE, and a speed-up of 11% - 2× in end-to-end training of a Transformer-XL model distributed over 16 GPUs for varying batch sizes. TACCL's codebase is open-source and is actively in use by researchers at universities and practitioners at Microsoft for Azure's GPU virtual machines. [1]

[1] https://github.com/microsoft/taccl

2 Background and Motivation

Collective communication in distributed ML workloads. Multi-GPU ML workloads typically communicate using MPI-style collectives like ALLGATHER, ALLTOALL, and ALLREDUCE shown in Figure 2. These primitives capture the application's intent behind the communication, thus allowing collective communication libraries to optimize for specific hardware configurations. In ALLGATHER, every GPU receives the data buffers of all other GPUs (left diagram in Figure 2). In ALLTOALL, every GPU receives different parts, or chunks, of the data buffers present on all GPUs. This effectively transposes the data chunk from buffer index to GPU index as can be seen in center diagram in Figure 2. In ALLREDUCE, every GPU ends up with a data buffer that has the results of performing a point-wise computation (e.g., sum in right diagram in Figure 2) over the same data index of all GPUs.

The parallelism strategy for the distributed ML workload determines which collective communication primitive is used. Data parallelism and some tensor model parallelisms [43] make use of the ALLREDUCE collective to aggregate gradients and intermediate data respectively from multiple GPUs. Expert parallelism [18, 27] and common deep learning recommendation models (DLRM) [32] make use of the ALLTOALL collective to shuffle intermediate data between experts and embedding lookup data between GPUs respectively. DLRMs [32] also make use of the ALLGATHER collective and another REDUCESCATTER collective to perform embedding lookups from embedding tables sharded over multiple GPUs.

Existing approaches to collective algorithms. Collective algorithms must be designed considering the target input sizes and the heterogeneity of the target topology. However, most collective communication libraries used for distributed ML today, including the state-of-the-art NCCL, use pre-defined templates of collective algorithms superimposed onto a target topology. For example, for collectives like ALLGATHER and REDUCESCATTER, NCCL identifies rings in the target topology and uses the Ring algorithm. For n GPUs, this algorithm requires $n-1$ link transfer steps per data chunk and is not ideal for smaller data sizes where link transfer latencies dominate. Further, this algorithm treats the slow inter-node and fast intra-node links similarly, scheduling equal number of data transfers across both. The communication is thus bottlenecked on the slower inter-node links, when it could have benefitted by sending more node-local data (i.e. data of GPUs local to the node) over the faster intra-node links instead.

For the ALLTOALL collective, NCCL implements the collective algorithm as peer-to-peer data transfers between all pairs of GPUs. This algorithm is topology-agnostic and often inefficient. For the ALLREDUCE collective, NCCL chooses between two algorithms — Double-Binary-Tree [34] and Ring. This decision is made according to the communication input size and number of nodes, but might not be most accurate, as it is based on hardcoded latency and bandwidth profiling done previously by Nvidia on their machines.

Designing efficient collective algorithms requires careful analysis of the topology and its performance with different buffer sizes. Recent work [9, 51] has shown that synthesis is a promising approach for generating collective algorithms for different topologies and to achieve bandwidth and latency optimality. However, scaling these approaches to multi-node (i.e. multi-machine) distributed GPU topologies has been a challenge. We measured the synthesis time for ALLGATHER and ALLTOALL collectives on topologies of two Azure NDv2 nodes and two Nvidia DGX2 nodes (Figure 5) using SCCL [9, 30]. We modified the codebase to include both topologies and attempted to synthesize the collectives with a 24-hour time limit set for each synthesis query. Given a 24-hour time limit, SCCL's `pareto-optimal` solver strategy did not finish synthesis for any combination of collective and topology. The only algorithm that SCCL could synthesize within the time limit was a latency optimal algorithm for ALLGATHER on two NDv2 nodes.

Low-effort inputs from algorithm designers. The search space of possible algorithms to implement a collective is intractably large and cannot be explored via brute-force. Deciding whether or not to route data chunks from n GPUs over l links in a topology has $O(2^{n \times l})$ combinations. As we scale to multi-node topologies, n as well as l will also scale, increasing the exponent quadratically. The search space explodes further if we consider the problem of ordering data sends at each link along with deciding routing for the data. We argue that high-level inputs from a human algorithm designer help reduce the search space to make algorithm synthesis more tractable. In the most extreme case, the designer would hand-write the entire algorithm. However, handcrafting data routing and scheduling over links to implement a collective is complex and requires many design choices. Instead, designers only provide input in the form of a communication sketch around which TACCL synthesizes an algorithm. Our goal is to ensure that providing inputs is a low-effort activity, but can discard large parts of the search space to achieve improvements in running-time of the synthesis engine.

Synthesis technique. TACCL synthesizes a collective algorithm by deciding the route that each data chunk in the collective should take in the topology as well as the ordering of chunks at every link. Even with communication sketches which reduces the search space for the synthesizer, this decision problem is NP-hard and the complexity increases exponentially with number of GPUs. To make the problem more tractable, we first relax the synthesis problem to solve just the routing of all data chunks and then heuristically order chunks sent over the same links according to bandwidth constraints. TACCL's synthesizer design along with communication sketches help TACCL synthesize efficient collectives for multi-node topologies.

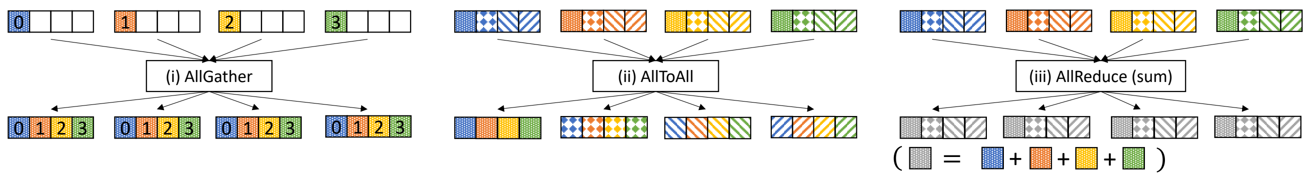


Figure 2: The initial and final data buffers on four GPUs participating in different collectives.

3 Communication Sketches

This paper proposes a new form of sketching [47] as an effective tool for users to communicate interesting aspects of collective communication algorithms to synthesis backends. Sketching approaches must strike a balance between allowing users to omit implementation details while still providing enough direction for the synthesis to scale. In our experience, *routing* is an aspect of collective communication that we often have intuitions about, while reasoning about *scheduling* tends to be tedious and better left to synthesis. Moreover, properties about scheduling are routing dependent since the order of operations is only relevant when routes intersect, which makes them harder to express. Meanwhile, interesting properties about routing are expressible globally, e.g., "never send over the InfiniBand NIC from this GPU". Therefore, we ask the algorithm designer (user) for four low-effort inputs as a part of the communication sketch:

- Specify the *logical topology* as a subset of the actual physical topology that the algorithm will operate on. This constrains the routes chosen by the communication algorithm and alleviates over-subscription of low-bandwidth links. For example, the outgoing links of all but one GPU can be removed in the logical topology to force all data going to remote GPUs to be relayed through one GPU.
- The logical topology abstracts away switches (e.g., NVSwitches, IBSwitches) in the GPU network. Users can annotate switches in the topology for the synthesizer to use certain *switch-hyperedge policies*, enabling it to apply synthesis policies that help algorithms avoid contention.
- Provide *algorithm symmetry* based on the symmetries in the topology and the collective.
- Specify the expected *input size* of the data, which is used as a part of the synthesis engine's built-in cost model.

We explain all parts of the communication sketch and provide an example sketch written for TACCL in Appendix A.

3.1 Logical Topology

The core of TACCL's communication sketch is a *logical topology* consisting of a subset of the physical topology after excluding links that the user prefers TACCL to avoid. A logical topology has as many nodes as the physical topology and inherits the cost model produced by TACCL's profiler for the physical topology. Logical topologies omit NICs and switches and use switch-hyperedges (Section 3.2), abstracting them away into links between GPUs. The reason is two-fold: TACCL runtime is embedded in NCCL runtime and NCCL has no direct control over NIC or switch use, and it allows TACCL to reason over a smaller graph thus enhancing scalability. Section 3.2 discusses implications of this abstraction.

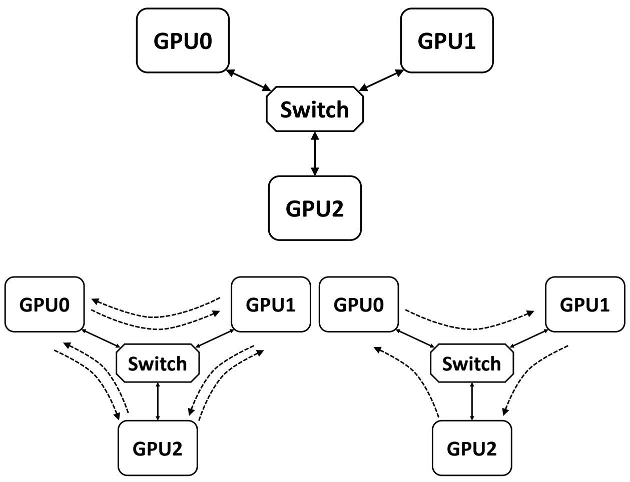
Example 3.1 (Sketching inside an NDv2)**.** Consider the physical topology of an Azure NDv2, given by the union of Figure 5a and Figure 5b. While NCCL is able to communicate over both the NVLink and PCIe connections, the bandwidth offered by the NVLinks is much higher than that of PCIe, and thus it is reasonable to set the logical topology to just the NVLink subgraph in Figure 5a.

Example 3.2 (Distributed sketching for NDv2 clusters)**.** It is essential to use PCIe connectivity for distributed collective communication with multiple NDv2 systems since the NIC is connected to GPUs over PCIe (Figure 5b). Due to lack of GPUDirect RDMA [1] on these systems, all communication over PCIe must pass through host memory. Therefore, care must be taken in choosing which links to use, as the PCIe links between PCIe switches and the CPU are oversubscribed. Obtaining maximum throughput communication requires a logical topology that avoids conflicting flows on the oversubscribed PCIe links. To build a logical topology for a cluster of NDv2 systems, a pair of receiver and sender GPUs is selected for each NDv2 such that the selected GPUs and the NIC are connected to the same PCIe switch.

3.2 Switch-Hyperedges

In a switched fabric with full bisectional-bandwidth, like the NVSwitch or IBSwitch fabrics in DGX-2 and NDv2 systems, nodes can simultaneously communicate at the full bandwidth of their ingress or egress links. However, as the number of connections through a switch, originating from a single GPU or NIC increases, the resulting queuing delays increase the latency. Figure 4 plots the accumulated ingress/egress bandwidth of exchanging varying volume of data (up-to 200-400 MB) for different number of connections over NVSwitches in a DGX2 node (left) and over IBSwitches among four DGX2 nodes (right). In both cases, the bandwidth drops as the number of connections increases despite the volume of data re-

(a) Physical topology with a switch.



(b) Max. connections strategy. (c) Min. connections strategy.

Figure 3: Effects of switch-hyperedge policies.

maining constant. However, for small input sizes, the difference for different number of connections is not significant. TACCL's logical topology does not model switches and does not capture the effect of number of connections.

TACCL incorporates the effect of multiple connections using *switch-hyperedges* in the synthesizer to control the number of connections between GPUs and switches. A switch-hyperedge replaces a switch with a set of direct links in the logical topology for the entire runtime of an algorithm. The synthesizer still has the freedom to select which direct links are imposed. To control the number of direct links for each switch-hyperedge, TACCL provides three policies for a user: (1) *maximize* the number of links, (2) *minimize* the number of links, and (3) freely choose any number of links. These policies are enforced by adding the number of Connections to the objective function (see Appendix B.1 for details).

Example 3.3 (Sketching for congestion)**.** Figure 3a shows a physical topology of three GPUs connected by a switch, where each GPU can communicate with any other GPU.

Figure 3b shows a logical topology with a switch-hyperedge that TACCL may choose with maximizing number of connections policy. This is desirable for small data sizes that result in low likelihood of congestion at the switch with large number of connections as shown in Figure 4.

In Figure 3c TACCL has minimized the number of connections, effectively resulting in a Ring topology. This is desirable for larger data sizes, as restricting the number of logical connections limits the congestion in the switch (Figure 4).

3.3 Algorithm Symmetry

Many collective communication algorithms are symmetric in a variety of ways. For example, ring algorithms follow a ring symmetry or in hierarchical algorithms, the local phases inside all machines are symmetric to each other. Inspired by this, TACCL offers a generic way to enforce algorithms to be symmetric.

The user may enforce a symmetry by supplying an *automorphism* of the logical topology and collective, i.e., a permutation of the ranks and chunks that maintains the structure of the topology and the collective pre- and post-conditions, and a *partition* of the ranks such that the automorphism maps each subset of ranks to some subset of the partition. TACCL will then restrict synthesis to algorithms with the same symmetry for all chunk transfers.

Example 3.4. Consider a cluster of two NDv2 systems and the task of synthesizing an ALLGATHER. A hierarchical symmetry may be specified with an automorphism composed of a the permutation $[8,\ldots,15,0,\ldots,7]$ for both chunks and ranks, and a partition $\{\{0,\ldots,7\},\{8,\ldots,15\}\}$. Now if an algorithm performs a send of chunk 0 from rank 0 to rank 1, then it must also include a send of chunk 8 from rank 8 to rank 9. However, sends between GPUs in different NDv2s, e.g., between 0 and 8, are not affected by the symmetry.

Since the internal topologies of NDv2 systems are identical, enforcing this symmetry is reasonable and helps TACCL scale to larger distributed topologies. Meanwhile, TACCL still has the freedom to synthesize the top-level algorithm and connect the systems to each other as it best can.

4 Physical Topologies of GPU systems

ML engineers use a variety of multi-GPU systems to meet the scaling challenges posed by growing ML models. Before users can effectively sketch algorithms for TACCL to synthesize, they must understand the *physical topology* of the target multi-GPU system. However, the performance characteristics of their heterogeneous links are sparsely documented and for some cloud offerings [5] even the topology is not given. To address this, TACCL includes a *physical topology profiler* to measure performance characteristics of links (§4.1) and to disambiguate the topology of some multi-GPU systems (§4.2). This section also serves as a concrete introduction into two target systems: Azure NDv2 and Nvidia DGX-2.

4.1 α-β Cost Model and Link Profiling

In the well-known α-β [21] cost model, α is the latency of a link and β is the inverse of its bandwidth. The cost of sending a chunk of size s along a link is $\alpha + \beta \cdot s$. TACCL's synthesizer adopts the α-β cost model for simplicity of encoding and tractability, but TACCL's communication sketches expose features that provide users additional control to avoid excessive concurrency and congestion (see Section 3), which are not modeled by the α-β cost model. α and β are affected by both the interconnect hardware and the software stack running

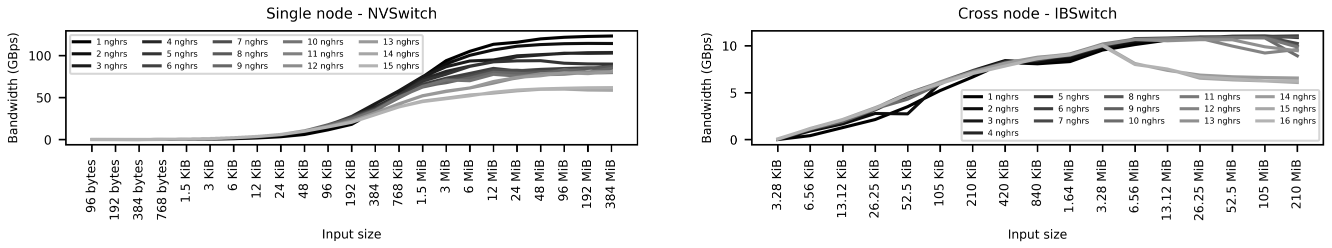
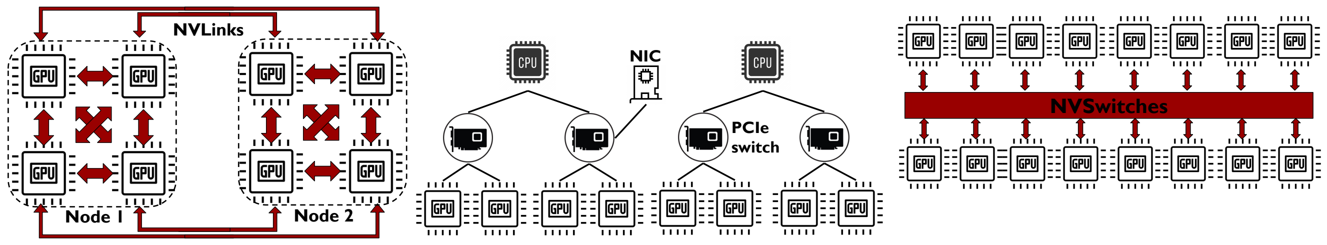


Figure 4: Multi-connection with varying number of GPU neighbors and data volume.



(a) NVLink connectivity of a NDv2. (b) PCIe connectivity of a NDv2. (c) NVLink connectivity of a DGX-2.

Figure 5: Aspects of physical topologies in various GPU systems.

the collective algorithm (for example software thread fences). TACCL's topology profiler measures and infers the α and β costs of different types of links in a GPU system.

Modern GPU systems, e.g., Azure NDv2 (Figure 5a) and Nvidia DGX-2 (Figure 5c), have the following types of interconnects: (1) **Peripheral Component Interconnect Express (PCIe)**, (2) **NVLink** [38], (3) **Infiniband** (IB) NICs [36]. A PCIe bus connects GPUs to CPUs with limited shared bandwidth (PCIe Gen3 offers $\approx$ 13 GBps). PCIe connections often form a hierarchy with PCIe switches (Figure 5b). NVLink [38], however, is a GPU to GPU *intra-node* connection with dedicated bandwidth. NVLinks are either directly connected to other GPUs (NDv2 in Figure 5a) or they are connected to other GPUs via NVSwitches [39] (DGX2 in Figure 5c). NVSwitches enable fully-connected GPU-GPU communication through NVLinks. IB is an inter-node interconnect which allows GPUs to communicate with GPUs in other nodes like in the Azure NDv2 (Figure 5b). IB NICs are usually connected to PCIe switches and GPUs may communicate directly with the NICs through Remote Direct Memory Access (RDMA) or indirectly via host memory.

The profiler empirically derives the α and β parameters of different links in the network by performing peer-to-peer data transfers between GPUs. We send n chunks one after another on a link and measure the time to transfer. As per the $\alpha-\beta$ cost model, the time to transfer is $n\cdot(\alpha+\beta\cdot s)$. We then send n chunks all at once on the link and attribute that time to be $\alpha+n\cdot\beta\cdot s$. Using several measurements of time to transfer, we solve for α and β. Table 1 shows the α and β values for

	Azure NDv2		Nvidia DGX-2	
Link	α (us)	β (us/MB)	α (us)	β (us/MB)
NVLink	0.7	46	0.7	8
InfiniBand	1.7	106	1.7	106

Table 1: Experimentally obtained α and β costs for Azure NDv2 and Nvidia DGX-2 nodes.

NDv2 and DGX-2 systems. Using these values, we expect that for transfers between two Azure NDv2 nodes over InfiniBand (IB), a sending two 32 KB chunks together as a single 64 KB chunk will be 17% faster as compared to sending two 32 KB chunks one after the other. However, chunks sent together can only be forwarded once the last chunk is received. Based on the α-β values, TACCL's synthesizer determines if and when to send chunks together on a link.

The α-β cost model causes TACCL's synthesizer to formulate an MILP formulation as opposed to an LP since an algorithm has to be expressed in terms of discrete chunks.

4.2 Inferring Multi-GPU Topologies

For Azure NDv2 systems the physical topology was not fully documented: while the NVLink topology (Figure 5a) is known to match that of Nvidia DGX1, we did not know how GPUs and the one 12.5 GBps Infiniband NIC were connected with PCIe. PCIe peer-to-peer communication (and thus GPUDirect RDMA [1]) is not enabled on these machines, meaning that all communication happens through buffers in

CPU memory over potentially shared PCIe links. Further, virtualization obscures the true PCIe topology (all 8 GPUs and the NIC appear directly connected to one CPU) and NUMA node and GPU IDs are not assigned consistently from VM to VM. This means that, without additional information, software cannot avoid contention over shared PCIe links, creating interference and high variance in performance.

To determine the PCIe topology, TACCL's profiler sends bandwidth and latency probes between the two CPUs, between pairs of GPUs, and between CPUs and the NIC. It answers the following questions:

- Which CPU is nearest to the NIC? We answer this using the latency of loopback operations between the NIC and each CPU.
- Which GPUs share a PCIe switch? We find all pairs of GPUs that get low bandwidth in a simultaneous copy to the CPU, indicating contention.
- Which GPUs share a PCIe switch with the NIC? We find which GPUs get low GPU to CPU bandwidth while the CPU is doing a loopback with the NIC. The CPU in this case is the one that is closer to the NIC.

With this profiling information we were able to deduce the PCIe topology (Figure 5b). Each CPU has two PCIe switches connecting to two GPUs each, and the Infiniband NIC is connected to one of these switches. Additionally, by running the profiler on every new NDv2 VM TACCL is able to select one of the NVLink topology's four automorphisms and set the `CUDA_VISIBLE_DEVICES` environment variable such that the NIC is always placed close to GPU 0.

5 TACCL Synthesizer

Once the user has written a communication sketch, they are ready to call TACCL's synthesizer. This section describes the synthesis process TACCL uses, as well as additional hyperparameters available to the user.

5.1 Problem Formulation

GPUs participating in a communication collective partition their initial data into C equal chunks where C is a hyperparameter selected by the user. TACCL's synthesizer routes and schedules these chunks. Given a communication sketch and a collective, the synthesizer decides chunk transfer schedules across every link in the network, such that each chunk reaches its destination GPUs as specified by the collective.

TACCL encodes this problem as a mixed integer linear program (MILP) with binary and continuous decision variables. The encoding has a continuous variable called *start_time* for every chunk and GPU to indicate when a chunk is available at a GPU. A binary variable *is_sent* for all chunk and link pairs denotes if a chunk is sent over a link. Another continuous variable *send_time* indicates when a chunk is sent over a link. The encoding has bandwidth and correctness constraints to ensure the correctness of a chunk transfer schedule. The objective of the MILP is to minimize *time* which is a continuous variable indicating the maximum time among all chunks that must reach their destination GPUs. Details of these variables and constraints are in Appendix B.

Additionally, TACCL's synthesizer also decides if it should merge some chunks and transfer them contiguously as one large buffer over a link. Sending n chunks contiguously in one send instruction over a link requires paying only one α latency cost whereas sending n chunks one after the other requires paying $n \times \alpha$ latency costs. Note that this does not change the β bandwidth cost. However, sending n chunks separately over a link enables TACCL to order them such that subsequent dependent sends from the destination of the link could be scheduled earlier. TACCL's synthesizer navigates this trade-off to minimize the time. TACCL uses this feature only for IB transfers due to their high α cost and ignores it for NVLinks due to their lower latency.

MILP problems in general are NP-hard. Luckily, there are solvers such as Gurobi [20] that apply heuristics to solve MILPs in a feasible way. However, this requires careful consideration regarding the number of variables and constraints in the formulation. In TACCL's formulation, transferring chunks over a link cannot overlap and an ordering among them is required. Therefore, potentially a binary decision is needed for every pair of chunks that may traverse a link. If we assume there are C chunks for a collective problem, there are $O(C^2)$ such decisions per link. Moreover, as the number of nodes increase, the number of links increase linearly (larger topology) and the number of chunks for a collective increases linearly (ALLGATHER) or even quadratically (ALLTOALL). This large set of variables and constraints leads to infeasible solver time and memory requirements.

To solve this problem, we divide the synthesis into three parts. First, the synthesizer solves an optimization problem to determine the path used by every chunk without fixing any ordering among chunks, then it heuristically orders the chunks over every link, and finally, it solves another optimization problem to determine chunk contiguity. Complete formal descriptions of each step are in Appendix B.

Step 1: Routing solves a MILP for finding the path of each chunk independent of other chunks, allowing chunks sent over a link to overlap. The objective of this MILP is to minimize the time, which we constrain to be the maximum of two sets of variables. (1) for each link, the number of chunks that traverse that link multiplied by the transfer time of a chunk over that link. (2) for the path of each chunk, the summation of transfer times of the chunk along every link in the path. Note that this is only a lower bound on the time since we do not consider link contention or chunk ordering. TACCL also constrains each chunk's path to be via GPU ranks that are on the shortest paths from their sources to their destinations using the links the user decided to include in the logical topology. If the communication sketch specifies an algorithm symmetry, TACCL adds the constraints for the symmetric sends. Replacing switches with switch-hyperedges is also applied in this step. For each switch-hyperedge, a user-provided policy on the number of unique connections to/form a switch is applied (see Section 5.2).

TACCL uses Gurobi [20] to solve this MILP and the solution gives every chunk a start_time for each GPU along its path. Clearly this step solves chunk routing, but only partially solves the chunk scheduling and contiguity problem and requires follow-up steps (explained next) to account for ordering the chunks sent over a link as well as minimizing α costs of sends. However, by using this technique, TACCL's synthesizer is able to reduce binary variables needed from $O(C^2)$ to $O(C)$ per link.

Step 2: Heuristic Ordering decides the chunk ordering sent on each link based on a heuristic. Note that this step is not an MILP and solely solved by a greedy algorithm. Regardless of when each chunk becomes available at a GPU, this step assigns a total order on the chunks sent over a link $l = (src, dst)$. This is decided by two heuristic functions. (1) chunks which need to traverse the longest path from *src* to their final GPU, have higher priority. (2) In case there is tie in (1), chunks which have traversed the shortest path from their initial GPU to *src*, have higher priority. This ordering will be used in Step 3 to assign the final schedules.

Step 3: Contiguity and Exact Scheduling solves an MILP problem to decide which chunks to send contiguously and gives the exact schedule. The path to be taken by chunks and their ordering over links have already been determined by the previous steps which are added as constraints to this MILP. The start_time and send_time variables are reassigned in this step by considering both the α and β costs for each transfer. In this step, the synthesizer allows either sending one chunk at a time or sending multiple chunks contiguously. This offers a trade-off between (1) sending the chunks that are available at the same time for a link according to the ordering in step 2 so that the subsequent sends can be scheduled earlier or (2) sending the chunks contiguously in one send instruction to save the latency cost. The objective of this MILP is to minimize the total time by enforcing all constraints which in TACCL solved by Gurobi [20]. The solution gives the exact schedule for each chunk. The details of these constraints and their formulation are in Appendix B.

5.2 Synthesizer Hyperparameters

TACCL's synthesizer has some additional parameters that control the synthesis process. These are provided by the user to the synthesizer (see Figure 1) through the communication sketch. Details of each parameter is described in Appendix A.

Buffer Size. TACCL needs the size of input/output buffers of a collective for the α-β cost model. In ML workloads the input/output buffer size is a known fixed value.

Chunk Partitioning. The data buffer at each GPU at the start of the collective can be partitioned into multiple equal chunks. Each chunk is considered as an atomic scheduling unit by the synthesizer and different chunks of the same data buffer can be routed over different links. The semantics of a collective forces a minimum number of chunks such as ALLTOALL which needs at least as many chunks as the number of GPU for each buffer. On one hand, using the minimum number of chunks is often times ideal for finding latency-optimal algorithms. On the other hand, providing a higher number of chunks allows the synthesizer to better utilize the links that might be idle otherwise which is better for finding bandwidth-optimal algorithms.

Switch-Hyperedge Policy. TACCL can enforce policies for the number of connections established over a set of links in a switch-hyperedge by counting links utilized for data transfer and setting this count as a part of the MILP objective. The `uc-max` policy will maximize the number of connections, which performs best for small data sizes, while `uc-min` will minimize the number of connections, which works well when the data size is large and congestion is a concern.

5.3 Synthesizing combining collectives

TACCL synthesizes combining collectives (i.e., collectives that combine chunks like REDUCESCATTER and ALLREDUCE) by utilizing synthesis of non-combining collectives, similar to the technique used by SCCL [9]. REDUCESCATTER can be implemented as an "inverse" of ALLGATHER— a send from a source GPU in ALLGATHER is instead received and reduced on the source GPU. However, simply inverting the sends does not work — a GPU may simultaneous send on different links in an ALLGATHER, but it cannot reduce all receives together in the inverse case. We thus order the inverse sends using heuristic ordering followed by contiguity encoding in order to synthesize REDUCESCATTER. ALLREDUCE is synthesized directly by concatenating REDUCESCATTER with an ALLGATHER algorithm.

6 Backend

The synthesizer described above generates an abstract algorithm that specifies the order in which the nodes communicate the various chunks. The goal of the backend is to implement this abstract algorithm. To do so, we extend NCCL [37] with an *interpreter* which we call TACCL runtime. While any communication algorithm can be trivially implemented using NCCL's point-to-point sends and receives, TACCL runtime enables us to execute the entire algorithm in a single kernel launch, eliminating multiple launch overheads. In addition, by reusing NCCL transport mechanisms, TACCL runtime is able to support all of NCCL's communication backends such as IB, Ethernet, NVLink, and PCIe.

6.1 TACCL runtime

The input to TACCL runtime[2] is a TACCL-EF program, which is an XML format for representing collective algorithms. TACCL-EF programs operate on three buffers: input, output and scratch. For each buffer, the program specifies the number of chunks it will be sliced into such that all chunks are equal size. Every step of the algorithm is expressed in terms of these chunks.

The program is divided into a set of GPU programs made up of threadblocks. Each threadblock is made up of a series of steps that are executed sequentially, with each step specifying an instruction and operands as indices into the input/output/scratch buffers. The current instruction set includes sends, receives (with optional reduction), and local copies. To simplify the implementation of TACCL runtime, each threadblock can send to and receive from at most one GPU. Additionally, threadblocks within a GPU can synchronize by indicating that one step depends on another step, which will cause the interpreter to wait until the dependency has completed before executing the dependent step.

The TACCL runtime extends NCCL and it is backward compatible with its API. Therefore, integrating TACCL runtime into machine learning frameworks such as PyTorch is a single line change wherein that change swaps the third-party NCCL library for TACCL runtime. This allows TACCL to dynamically swap in collective algorithms generated for any training/inference workload using `torch.distributed`.

6.2 Lowering to TACCL runtime

To target TACCL-EF, abstract algorithms are lowered to the executable format. The sets of sends operating on abstract chunks that comprise the steps of the algorithm are transformed into pairs of send and receive operations operating on concrete buffer indices. Furthermore, these operations are placed sequentially into threadblocks and any necessary dependencies recorded between them.

Buffer allocation. Input and output buffers are preallocated by the user and passed to the collective. Scratch buffers are allocated by the TACCL runtime per TACCL-EF. Chunks are indices in the input, output and scratch buffers. For chunks that are common for both the input and the output buffers (e.g. as in ALLGATHER) a local copy from input to the output buffer is performed at the end.

Instruction generation. The operations of the abstract algorithm are split into two instructions for the sender and receiver GPU, and chunks are translated into buffer references and indices according to the buffer allocation.

Dependency insertion. TACCL transforms a synthesized algorithm into the asynchronous execution model of TACCL-EF and dependencies for each buffer index are inserted to ensure that the data dependencies present in the abstract algorithm are honored.

[2]Link to code: https://github.com/microsoft/msccl

Threadblock allocation. Instructions are grouped such that all of them are either sending to at most one GPU and/or receiving from at most another GPU (possibly different). Order of the instructions inside a group should follow the order of the abstract algorithm. TACCL allocates a threadblock for each group of instructions.

Instances. NCCL and consequently TACCL runtime cannot saturate the bandwidth of a link in a topology using a single threadblock. Thus, TACCL generates multiple instances of the algorithm to maximize the performance. This is done by subdividing each chunk into n subchunks that follow the same path as the parent chunk. All groups of instructions and their threadblocks are duplicated n times and executed in parallel. §7.2 explores the performance implications of choices of n.

7 Evaluation

We evaluate algorithms obtained with TACCL for ALLGATHER, ALLTOALL, and ALLREDUCE collectives on a cluster of 32 GPUs comprised of two Nvidia DGX-2 nodes or up-to four Azure NDv2 nodes. To compare performances, algorithm bandwidth [33] measurement is used which is calculated by input buffer size divided by execution time. We synthesize TACCL algorithms by exploring different communication sketches and compare them against the popular Nvidia Collective Communication Library (NCCL) (v.2.8.4-1). This section analyzes how different communication sketches impact the performance of the algorithms synthesized by TACCL. In particular, we perform ablation studies by varying the inter-node connections in the logical topology, changing synthesizer hyperparameters, and changing the number of instances used when lowering to TACCL-EF. To evaluate how TACCL's speedups translate to end-to-end performance, we use algorithms generated by TACCL in two large language models, Transformer-XL and BERT. Finally, we discuss the synthesis time required by TACCL to generate these algorithms.

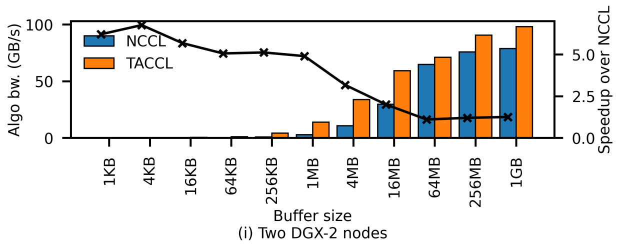
We believe our focus on up to 32 GPUs covers a large section of important use cases: in an internal cluster of DGX-2 nodes at Microsoft, the sum of GPUs in jobs of at most 32 was 93.7% of all jobs in the second half of 2021.

7.1 Standalone Experiments

All our communication sketches for DGX-2 and NDv2 use a hierarchical symmetry like the one in Example 3.4.

7.1.1 ALLGATHER

ALLGATHER on DGX-2. Figure 6(i) shows the algorithm bandwidth for TACCL's synthesized algorithms on two DGX-2 nodes for each output buffer size and plots it against that of



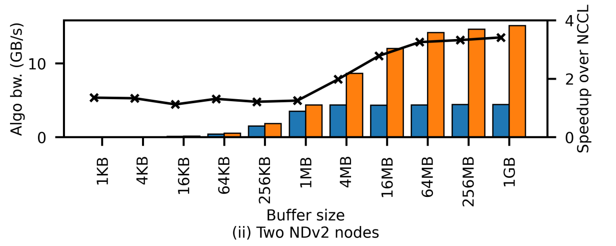
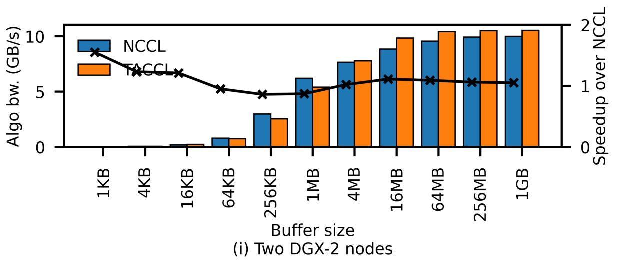


Figure 6: ALLGATHER comparisons of NCCL to TACCL's best algorithm at each buffer size.



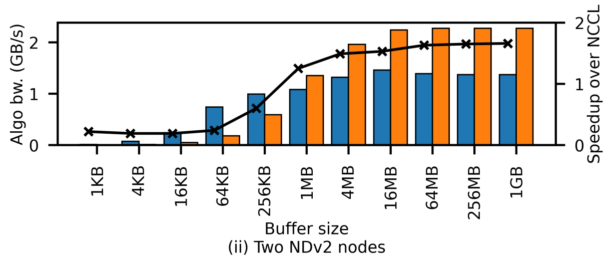


Figure 7: ALLTOALL comparisons of NCCL to TACCL's best algorithm at each buffer size.

NCCL. We show the speedup of TACCL's algorithms over NCCL on the right Y-axis of the plot. We used two different sketches for this topology which will be explained next.

A DGX-2 node has 16 V100 GPUs (Figure 5c) where each pair of GPUs share a PCIe switch with a NIC. This makes it natural to assign one GPU in a pair to be a receiver and the other to be a sender by eliminating outgoing and incoming links, respectively, in the logical topology. We design a sketch (*dgx2-sk-1*) that uses this logical topology, sets chunk size to 2MB, uses two chunk partitions for each buffer, and the sets switch-hyperedge policy to `uc-min`. With this sketch, TACCL synthesizes an ALLGATHER algorithm for two DGX-2 nodes. This algorithm almost saturates the inter-node bandwidth during the entire run of the algorithm and provides a $20\% - 25\%$ speedup over NCCL for large buffer sizes in the 256MB - 1GB range.

Next, we design a sketch (*dgx2-sk-2*) for smaller sizes. This sketch allows both GPUs in a pair to utilize the shared NIC. However, local GPU i on each node is only allowed to send/receive to/from local GPU i on the other node. Since the IB is shared, we double the β cost for each IB transfer to $2 * \beta_{IB}$ cost. In this sketch, chunk size is set to 1KB and the switch-hyperedge policy is `uc-max`. Using this sketch TACCL synthesizes an algorithm that is $4.9\times - 6.7\times$ faster than NCCL in the 1KB - 1MB range, and $10\% - 3.8\times$ faster than NCCL in the 2MB - 64MB range. On inspecting this algorithm, we found that TACCL's synthesized algorithm overlaps inter-node sends with intra-node all-pair ALLGATHER of node-local data chunks followed by an intra-node all-pair ALLGATHER of the node-external chunks received over IB.

Figure 6(i) shows the algorithm bandwidth and the speedup over NCCL baseline for the best of these two sketches for each output buffer size.

ALLGATHER on NDv2. The sketch we used, *ndv2-sk-1*, uses the logical topology discussed in Example 3.2, in which a sender and a receiver GPU were dedicated such that they are on the same PCIe switch as the NIC. We use a single instance when lowering algorithms into TACCL-EF for data sizes 1MB and below, and use 8 instances for larger data sizes. Figure 6(ii) compares the synthesized algorithms to NCCL on two Azure NDv2 nodes. TACCL's synthesized algorithms are $12\% - 35\%$ faster than NCCL for buffer sizes of 1KB - 1MB, and $61\% - 3.4\times$ faster than NCCL for sizes larger than 1MB. These algorithms better saturate the inter-node bandwidth thanks to the dedicated send/receiver GPUs.

We similarly synthesize ALLGATHER algorithms for four NDv2 nodes and present the results in Figure 11(i) in Appendix C. These algorithms are $10\% - 2.2\times$ faster than NCCL depending on buffer size.

7.1.2 ALLTOALL

ALLTOALL on DGX-2. We explore the synthesis of ALLTOALL algorithms by reusing the *dgx2-sk-2* communication sketch designed in the previous section. Figure 7(i) compares the resulting algorithm on two DGX-2 nodes. The synthesized algorithm using this sketch performs up-to 15% faster than NCCL for batch sizes of 2MB and larger. For this sketch, TACCL's synthesizer coalesces chunks sent in inter-node transfer in this algorithm, which reduces the latency of transfers over IB. TACCL also uses a communication sketch with chunk size set as 1KB and a logical topology where GPUs have links to all other GPUs connected via the NIC (*dgx2-sk-3*). This algorithm is up-to 55% faster than NCCL for small buffer sizes ranging from 1KB to 16KB.

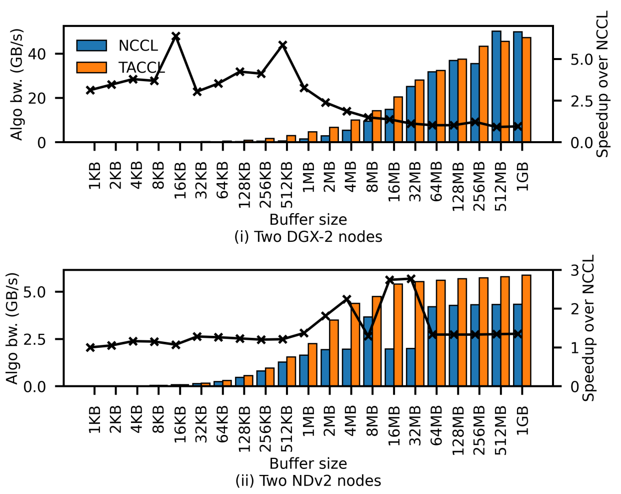


Figure 8: ALLREDUCE comparisons of NCCL to TACCL's best algorithm at each buffer size.

ALLTOALL on NDv2. Figure 7(ii) shows a comparison of TACCL's best algorithms for ALLTOALL on two Azure NDv2 nodes against NCCL. We reuse the communication sketch *ndv2-sk-1* and set the chunk size to 1MB. The generated algorithms run 53% − 66% faster than NCCL for buffer sizes between 16MB - 1*GB* We explore another sketch (*ndv2-sk-2*) with a logical topology in which all GPUs in a node are fully-connected to all the GPUs in the other node and set chunk size as 1KB. The algorithm generated by TACCL using this sketch performs up-to 12% faster than NCCL for buffer sizes from 1KB to 128KB.

For four NDv2 nodes, TACCL's synthesized algorithms uses communication sketch *ndv2-sk-1* and they are up-to 46% faster than NCCL for buffer size greater than 1MB, as shown in Figure 11(ii) in Appendix C.

7.1.3 ALLREDUCE

ALLREDUCE on DGX-2. As discussed in Section 5.3, TACCL composes REDUCESCATTER with ALLGATHER to implement ALLREDUCE and an algorithm for REDUCESCATTER can be constructed by inverting an ALLGATHER algorithm. Figure 8(i) shows the performance of TACCL algorithms on two DGX-2 nodes. The ALLREDUCE synthesized from the ALLGATHER using *dgx2-sk-2* is 49% − 6.4× faster than NCCL for buffer sizes ranging from 1KB - 4MB. TACCL's generated algorithms by using other communication sketches like *dgx2-sk-1* are 2% − 37% faster than NCCL for buffer sizes ranging from 16MB - 256MB. For buffer sizes of 512MB and greater, our algorithms are at most 9% slower than NCCL. This is because NCCL uses the more optimized fused communication instructions (such as receive-reduce-copy-send) in its ALLREDUCE communication which are unavailable in TACCL's lowering. We leave these such further optimizations for future work.

ALLREDUCE on NDv2. These algorithms are based on the ALLGATHER synthesized from the *ndv2-sk-1* sketch and use two versions with 1 and 8 instances. Figure 8(ii) compares them to NCCL on two NDv2 nodes. The single instance TACCL algorithm outperforms NCCL's ALLREDUCE by up to 28% for buffer sizes of up to 1MB, while the 8 instance algorithm outperforms NCCL by 28% − 2.7× for larger sizes.

On 4 NDv2 nodes, as shown in Figure 11(iii) in Appendix C, the TACCL algorithms are up to 34% faster than NCCL for small buffer sizes and 1.9 × −2.1× faster than NCCL for larger buffer sizes.

7.2 Impact of Varying Synthesizer Inputs

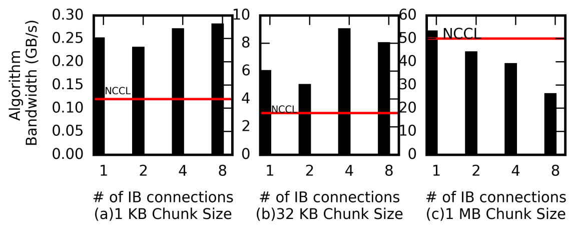
In this section, we explore modifications to communication sketches, as well as the synthesizer hyperparameters and the instances for the lowering, in order to understand their impact on the performance of the synthesized algorithms. Our aim is to demonstrate that the controls offered by TACCL have intuitive effects on the resulting algorithms, which is necessary for effectively communicating user intuition to TACCL.

We present our analysis for the ALLGATHER collective on two Nvidia DGX-2 nodes. Unless mentioned otherwise, we use the following communication sketch as the baseline: same logical topology as *dgx2-sk-1*, chunk size set to 1MB, data partitioning set to 1, and the switch-hyperedge policy set to `uc-max`.

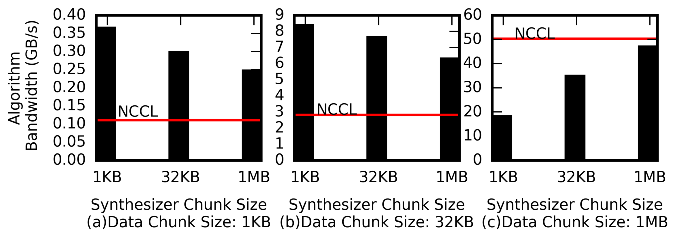
Changing logical topology. We create a logical topology with a dedicated sender and receiver GPU similar to *dgx-sk-1* except we allow a sender to be connected to *n* different receivers in the other node. Figure 9a shows the algorithm bandwidth of ALLGATHER obtained by varying *n*, the number of IB connections per GPU, for a fixed chunk size of 1KB, 32KB, and 1MB. For a 1KB chunk size, we found the algorithm that uses 8 IB connections per NIC performs better than algorithms using fewer connections. As the chunk size increases to 32KB and 1MB, the optimal number of IB connections per NIC reduces to 4 and 1, respectively. The benefits of link sharing shrink as the chunk size increases and β-cost starts dominating over the α-cost.

Changing transfer cost using chunk size. We analyze the sensitivity of TACCL's synthesizer to the data size provided in the communication sketch when its algorithms are applied on a communication using a different data size. Figure 9b shows the performance of ALLGATHER algorithm for three different chunk sizes (1KB, 32KB, and 1MB). Algorithms generally perform well for a range of data sizes close to what they have been synthesized for. We recommend trying a small set of nearby sizes to ensure the best performance.

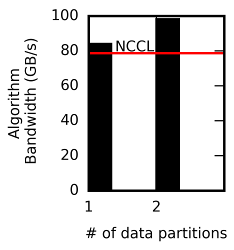
Changing data partitioning. Figure 9c shows the algorithm bandwidth of algorithms generated by partitioning data on



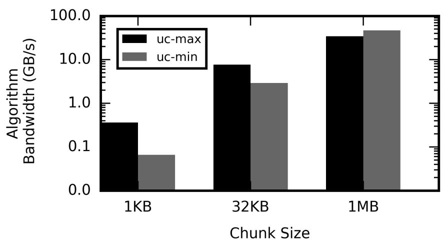
(a) Logical topology



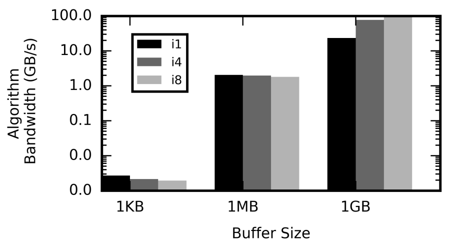
(b) Chunk size



(c) Data partition



(d) Switch-hyperedge strategies



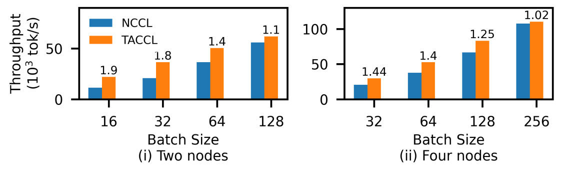
(e) Runtime instances

Figure 9: Algorithm bandwidth of ALLGATHER algorithms on DGX-2 by varying different inputs to TACCL

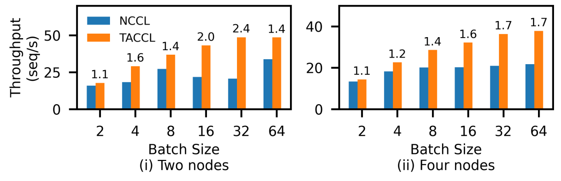
each GPU into a single or two chunks. We set the switch-hyperedge policy to `uc-min` and fix number of instances to 8. At a large buffer size of 1GB, the algorithm generated for two data chunks utilizes bandwidth better as compared to the algorithm generated for a single data chunk per GPU.

Changing switch-hyperedge policy. Figure 9d shows the algorithm bandwidth for algorithms generated and evaluated for 1KB, 32KB, and 1MB chunks. The algorithm bandwidth is displayed in log-scale. We vary the switch-hyperedge policy between `uc-max` and `uc-min`. For smaller buffer sizes, the `uc-max` configuration performs better than `uc-min`, whereas for larger buffer sizes, `uc-min` performs better than `uc-max`.

Changing number of instances. Figure 9e shows algorithm bandwidth with instances ranging from 1 to 8. The switch-hyperedge policy for these algorithms is set to `uc-min`. Increasing the number of instances improves bandwidth utilization — multiple threadblocks seem to be needed to keep the six NVLinks in a V100 busy. However, a larger number of threadblocks also increases latency, which we suspect is due to unfavorable scheduling of synchronization related memory operations onto the NVLinks at the start of each send. Since latency cost dominates for small buffer sizes, using a large number of instances only increases the latency cost. As the buffer size increases, the bandwidth improvements due to more instances become predominant. Since switch-hyperedge policy and number of instances have a similar relation with chunk sizes, we always run `uc-max` algorithms with a single instance and `uc-min` algorithms with 8 instances.



(a) Transformer-XL



(b) BERT

Figure 10: Training throughput using TACCL's collective algorithms on Transformer-XL and BERT compared against NCCL on 2 and 4 Azure NDv2 nodes. Speedup over NCCL is mentioned on top of the bars.

7.3 End-to-End Training.

We evaluate TACCL on distributed training of two large language models, Transformer-XL [4, 13] and BERT [3, 15], on two (and four) Azure NDv2 nodes, i.e. 16 (and 32) GPUs. Transformer-XL uses data parallelism and whereas BERT uses model parallelism. The typical transfer sizes for ALLREDUCE in Transformer-XL is in the 20 - 40MB range, and for BERT it is about 2MB. Both models communicate with `torch.distributed` and, as explained in Section 6, using TACCL algorithms in them is quite straightforward.

We lower the algorithm synthesized by the synthesizer

AllGather		AlltoAll		AllReduce	
Sketch	Time(s)	Sketch	Time(s)	Sketch	Time(s)
dgx2-sk-1	35.8	*dgx2-sk-2*	92.5	*dgx2-sk-1*	6.1
dgx2-sk-2	11.3	*ndv2-sk-1*	1809.8	*dgx2-sk-2*	127.8
ndv2-sk-1	2.6	*ndv2-sk-2*	8.4	*ndv2-sk-1*	0.3

Table 2: Synthesis time for TACCL algorithms for different collectives using different communication sketches.

into TACCL-EF with 1 and 8 instances, and show the performance of both against NCCL. Figure 10a and Figure 10b show the training throughput obtained by using TACCL's collective algorithms for communication instead of NCCL for Transformer-XL and BERT respectively for different batch sizes. TACCL speeds up training of Transformer-XL by $11\% - 1.94\times$ on 2 nodes and by $2\% - 1.44\times$ on 4 nodes. The speedup for BERT is $12\% - 2.36\times$ on 2 nodes and $7\% - 1.74\times$ on 4 nodes. Depending on the memory available per GPU and on how the batch size affects model accuracy, any of these batch sizes might be chosen for use in practice.

We also use algorithms synthesized by TACCL for ALLTOALL and ALLREDUCE collectives for training an internal Microsoft's mixture-of-experts workload on two NDv2 nodes. The ALLTOALL and ALLREDUCE sizes required for this model are $\approx$ 6MB and $\approx$ 256MB, respectively. TACCL improves the end-to-end throughput of this model by 17%.

7.4 Synthesis Time

Table 2 shows the total time it takes for TACCL to synthesize algorithms for different collectives using some of the communication sketches mentioned in Section 7.1. In most cases synthesis takes from seconds to a few minutes, making it amenable to a human-in-the-loop approach. When synthesizing an ALLTOALL collective using some communication sketches, TACCL's contiguity encoding may take more time in proving the optimality of a feasible solution. We put a time limit of 30 minutes on the contiguity encoding in these cases. The contiguity encoding for sketch *ndv2-sk-1* reaches this timeout, but a feasible solution was already found in 4min 14s. We have also been able to synthesize an ALLGATHER for 80 GPUs (10 NDv2 nodes) in under 8 minutes.

8 Related Work

The MPI standard provides a set of collective communication algorithms that enable efficient distributed computations of interconnected nodes [16]. The HPC community has focused on the efficient implementation of these MPI collective algorithms [40, 50] and demonstrated how to build optimized algorithms for specific interconnects, like mesh, hypercube, or fat-tree [7, 8, 41]. In contrast to TACCL, these prior works assume homogeneous interconnects and are often only focused on bandwidth optimality. Hybrid algorithms [7, 10] combine bandwidth- and latency-optimal algorithms based on input sizes, but only qfor mesh networks.

NCCL [37] is a GPU implementation of a subset of the standard MPI collectives, optimized for NVLINK and Infiniband interconnects. While NCCL uses the topology of GPU connections and NIC placement along with buffer size to decide between two main types of communication algorithms — Ring and Tree, it is agnostic to the exact performance profile of the links, and thus (as we show) is often multiple times slower than TACCL's topology aware collectives.

Recent works like SCCL [9], Blink [51], and Plink [29] specialize algorithms for the underlying topology. SCCL solves an integer programming encoding based on discrete-time values in the form of steps and rounds of the algorithm in order to achieve the pareto-frontier of latency- and bandwidth-optimal algorithms. SCCL is able to synthesize a novel pareto-optimal ALLGATHER algorithm for an Nvidia DGX1 node, but its restrictive formulation constrains it to only only synthesize algorithms for single-node topologies. TACCL on the other hand synthesizes collective algorithms for multi-node topologies. Blink uses a heuristic spanning-tree packing algorithm to maximize bandwidth utilization within a node and a hierarchical approach across. Blink has good performance over NCCL in the case when NCCL cannot create rings spanning all GPUs inside a node. TACCL, on the other hand, outperforms NCCL when using the entire node of GPUs. Plink constructs a logical topology based on bandwidth and latency probes of the physical topology to avoid oversubscribed and congested links and searches for a reasonable clustering of nodes for a two-level hierarchical reduction strategy. Plink builds that hierarchical reduction from known primitives and does not search over the space of possible algorithms.

There are also hierarchical approaches to implement collectives [12, 29, 42, 51]. For example, Horovod [42] implements an ALLREDUCE by a local ReduceScatter, a global ALLREDUCE, and then a local ALLGATHER. These methods do not search over possible algorithms, but instead pick from a known set of decompositions. Concurrent to our work, Ningning et al. [52] use syntax guided synthesis to combine base MPI primitives among a subset of nodes to hierarchically generate larger MPI primitives for the entire network. In contrast, TACCL uses a fine grained approach for algorithm synthesis while using communication sketches for scalability. Combining these two complementary approaches is an interesting opportunity for future work.

Program sketching [24, 47, 49] is a popular technique that has been applied to a variety of problems from synthesizing stencil computations [48], converting hand drawings to images [17] to social media recommendations [11]. Our work builds on this body of work to use sketching to effectively search a large space of communication algorithms.

Lastly, network flow problems have used linear programming to solve routing and scheduling problems for traffic engineering [22, 23, 25, 44, 46] and topology engineering [45].

These techniques, however, cannot be used for generating collective algorithms since communication collectives do not follow all flow properties. Non-source GPUs in a collective can send the same chunk over different links in parallel while having received that chunk only once, which violates an important flow-conservation property used extensively in network flow problem literature. TACCL on the other hand makes use of communication sketches and an encoding relaxation technique to solve a continuous-time integer linear programming that faithfully models communication collectives.

9 Conclusion and Future Work

TACCL is a topology and input-size aware collective communication library for multi-node distributed machine learning training and inference. TACCL uses user-provided communication sketches to guide synthesis of collective algorithms. Using a three-step technique of relaxed routing, heuristic ordering, and contiguity and exact scheduling, TACCL generates efficient collectives for multi-node topologies. We also make some brief observations about TACCL below:

Scalability. TACCL can synthesize algorithms for large-scale nodes - we have been able to synthesize an ALLGATHER algorithm for 8 Azure NDv2 nodes using TACCL in under 5 minutes. As compared to NCCL, this algorithm has up-to $1.7\times$ higher algorithm bandwidth for different data sizes. We also evaluated TACCL's synthesis for 8 Nvidia DGX-2 nodes (128 GPUs) and found a solution in around 11 hours. While TACCL scales to multi-node topologies, the synthesis technique is still based on solving an NP-hard problem that grows exponentially with a quadratic power with scale. As a future work, we would like to scale TACCL further by hierarchically composing synthesized algorithms.

Generality across different topologies. Apart from hierarchical topologies like Nvidia DGX-2 and Azure NDv2, TACCL can also be applied to non-hierarchical topologies like a 2D-Torus. We were able to synthesize an ALLGATHER algorithm for a 2D 6×8 Torus using TACCL. We made use of the symmetry attribute in communication sketches to explore synthesis for this topology. However, the amount of exploration we can do with different communication sketches may be more limited in these cases than for hierarchical topologies.

Exploring communication sketches. Communication sketches have proven effective in narrowing the search space of algorithms. Interestingly, different communication sketches can optimize different ranges of input sizes. Communication sketches reflect the intuition of developers, and by intelligently exploring the space of communication sketches we can obtain a range of collective algorithms with different performance characteristics. Learning an automated controller for exploring communication sketches is an interesting direction for collective algorithm synthesis in the future.

To conclude, TACCL uses the abstraction of communication sketches and a novel problem formulation to generate efficient algorithms for collectives like ALLGATHER, ALLTOALL, and ALLREDUCE. The algorithms thus generated are up-to $6.7\times$ faster than the state-of-the-art NCCL and result in $11\% - 2.4\times$ faster end-to-end training time.

Acknowledgements

We would like to thank our shepherd, Aurojit Panda, the anonymous reviewers at NSDI'23, and the members of the Systems and Storage Lab at UT Austin for their insightful comments and suggestions. This work was partially supported by NSF CAREER #1751277, the UT Austin-Portugal BigHPC project (POCI-01-0247-FEDER-045924), and donations from VMware.

References

[1] GPUDirect RDMA, 2021. https://developer.nvidia.com/gpudirect.

[2] Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[3] Megatron-LM. `https://github.com/NVIDIA/Megatron-LM`, 2022.

[4] Transformer-XL. `https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/LanguageModeling/Transformer-XL`, 2022.

[5] Azure ND-series, 2021. https://docs.microsoft.com/en-us/azure/virtual-machines/nd-series.

[6] Azure NDv2-series, 2021. https://docs.microsoft.com/en-us/azure/virtual-machines/ndv2-series.

[7] Michael Barnett, Rick Littlefield, David G Payne, and Robert van de Geijn. Global combine on mesh architectures with wormhole routing. In *[1993] Proceedings Seventh International Parallel Processing Symposium*, pages 156–162. IEEE, 1993.

[8] Shahid H Bokhari and Harry Berryman. Complete exchange on a circuit switched mesh. In *1992 Proceedings Scalable High Performance Computing Conference*, pages 300–301. IEEE Computer Society, 1992.

[9] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium*

on Principles and Practice of Parallel Programming, pages 62–75, 2021.

[10] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[11] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Using program synthesis for social recommendations. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, page 1732–1736, New York, NY, USA, 2012. Association for Computing Machinery.

[12] Minsik Cho, Ulrich Finkler, Mauricio Serrano, David Kung, and Hillery Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *IBM Journal of Research and Development*, 63(6):1:1–1:11, 2019.

[13] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[14] Wei Deng, Junwei Pan, Tian Zhou, Deguang Kong, Aaron Flores, and Guang Lin. Deeplight: Deep lightweight feature interactions for accelerating ctr predictions in ad serving. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*, WSDM '21, page 922–930, New York, NY, USA, 2021. Association for Computing Machinery.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[16] Jack Dongarra et al. MPI: A message-passing interface standard version 3.0. *High Performance Computing Center Stuttgart (HLRS)*, 2(5):32, 2013.

[17] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. *Advances in neural information processing systems*, 31, 2018.

[18] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *CoRR*, abs/2101.03961, 2021.

[19] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. In-network aggregation for shared machine learning clusters. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 829–844, 2021.

[20] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022.

[21] Roger W. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Computing*, 20(3):389–398, 1994.

[22] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):15–26, August 2013.

[23] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):3–14, August 2013.

[24] Jinseong Jeon, Xiaokang Qiu, Jeffrey S Foster, and Armando Solar-Lezama. Jsketch: sketching for java. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 934–937, 2015.

[25] Srikanth Kandula, Ishai Menache, Roy Schwartz, and Spandana Raj Babbula. Calendaring for wide area networks. In *SIGCOMM'14*.

[26] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761. USENIX Association, April 2021.

[27] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *CoRR*, abs/2006.16668, 2020.

[28] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: A rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 41–54, New York, NY, USA, 2018. Association for Computing Machinery.

[29] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud. In *Proceedings of Machine Learning and Systems 2020*, pages 82–97. 2020.

[30] Microsoft SCCL, 2021. https://github.com/microsoft/sccl.

[31] Using deepspeed and megatron to train megatron-turing nlg 530b, the world's largest and most powerful generative language model. https://www.microsoft.com/en-us/research/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/. Accessed October 2021.

[32] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 993–1011, 2022.

[33] NCCL Tests, 2021. https://github.com/NVIDIA/nccl-tests.

[34] NCCL Tree Algorithm, 2019. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4.

[35] Nvidia DGX Systems, 2021. https://www.nvidia.com/en-us/data-center/dgx-systems/.

[36] Nvidia InfiniBand, 2021. https://www.nvidia.com/en-us/networking/infiniband-adapters/.

[37] Nvidia NCCL, 2021. https://github.com/nvidia/nccl.

[38] Nvidia NVLink and NVSwitch, 2021. https://www.nvidia.com/en-us/data-center/nvlink/.

[39] NVIDIA NVSWITCH The World's Highest-Bandwidth On-Node Switch , 2021. https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf.

[40] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E Fagg, Edgar Gabriel, and Jack J Dongarra. Performance analysis of mpi collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[41] David S Scott. Efficient all-to-all communication patterns in hypercube and mesh topologies. In *The Sixth Distributed Memory Computing Conference, 1991. Proceedings*, pages 398–399. IEEE Computer Society, 1991.

[42] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow, 2018.

[43] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[44] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. Cost-effective cloud edge traffic engineering with cascara. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 201–216. USENIX Association, April 2021.

[45] Rachee Singh, Nikolaj Bjorner, Sharon Shoham, Yawei Yin, John Arnold, and Jamie Gaudette. Cost-Effective Capacity Provisioning in Wide Area Networks with Shoofly. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 534–546, New York, NY, USA, 2021. Association for Computing Machinery.

[46] Rachee Singh, Manya Ghobadi, Klaus-Tycho Foerster, Mark Filer, and Phillipa Gill. Radwan: Rate adaptive wide area network. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 547–560, New York, NY, USA, 2018. Association for Computing Machinery.

[47] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, USA, 2008. AAI3353225.

[48] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 167–178, New York, NY, USA, 2007. Association for Computing Machinery.

[49] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 404–415, New York, NY, USA, 2006. Association for Computing Machinery.

[50] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[51] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 172–186, 2020.

[52] Ningning Xie, Tamara Norman, Dominik Grewe, and Dimitrios Vytiniotis. Synthesizing optimal parallelism placement and reduction strategies on hierarchical systems for deep learning. *CoRR*, abs/2110.10548, 2021.

[53] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. Is network the bottleneck of distributed training? In *Proceedings of the Workshop on Network Meets AI & ML*, pages 8–13, 2020.

Appendix

A Communication Sketch Input

TACCL adopts a user-in-the-loop approach where algorithm designers provide a communication sketch to guide communication algorithm synthesis by TACCL. TACCL's synthesizer takes in a profiled topology provided by TACCL profiler along with a communication sketch provided by a human-in-the-loop. A communication sketch comprises of a logical topology, switch-hyperedge strategy, symmetry information, input size, and other hyperparameters. Listing 1 gives an example of how users can provide a communication sketch input to the TACCL synthesizer. Here, we show an example of the communication sketch *dgx2-sk-1* used in the evaluation to synthesize an ALLGATHER algorithm for 2 Nvidia DGX-2 nodes (each node has 16 GPUs and 8 NICs, every two GPUs in the node share a NIC).

The sketch annotates the NVSwitch in each node and sets a `uc-min` switch-hyperedge strategy. Further, the inter-node sketch fixes the sender and receiver GPUs in a node for inter-node data transfers. In our example, the odd-numbered GPUs sharing a NIC are chosen as senders and the even-numbered GPUs are chosen as receivers for inter-node communication. The user also annotates how the inter-node relay GPUs would split the inter-node bandwidth using a *beta_split* attribute. Since only a single GPU per NIC is chosen in our example to perform inter-node send and similarly receive, the bandwidth is not split. Optionally, the user can also map chunks to sender GPUs so that only mapped GPUs are used for inter-node transfers for the chunk. The *chunk_to_relay_map* attribute defines the parameters for the mapping function. The communication sketch also allows users to play with rotational symmetry for data routing. Given a symmetry offset and a group size, a chunk transfer over a link is set to be equivalent to a rotationally symmetric chunk over a rotationally symmetric link. In our example of the *symmetry_offset* attribute, using $[2,16]$ fixes an intra-node symmetry with an offset of two, and using $[16,32]$ fixes a symmetric data transfer pattern between the two DGX-2 nodes. Hyperparameters like input data partitioning and input size can also be provided via the communication sketch.

Listing 1: Example sketch *dgx2-sk-1* for ALLGATHER

```
{
    // sketch for intra-node policy
    "intranode_sketch": {
        "strategy": "switch",
        "switches":
            [[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]],
        "switch_hyperedge_strategy": ["uc-min"]
    },

    // sketch for communication policy between any
        two nodes
    "internode_sketch": {
        "strategy": "relay",
        "internode_conn": {"1" : [0], "3" : [2], "5"
            : [4], "7" : [6], "9" : [8], "11" :
            [10], "13" : [12], "15" : [14]}, // "i":
            [j1, j2] implies GPU i in a node will
            only send data to GPU j1 and j2 of
            another node
        "beta_split": {"1": 1, "3": 1, "5": 1, "7" :
            1, "9" : 1, "11" : 1, "13" : 1, "15" :
            1}, // "i": n implies inter-node sends
            from a GPU i of a node will use 1/n-th
            of the inter-node bandwidth
        "chunk_to_relay_map": [2,1] // maps chunk to
            a sender relay GPU. [r1,r2] means chunk
            c will be send to another node via GPU
            (rp//r1)*r1 + r2, where rp is the
            precondition GPU for chunk c
    },

    // enforces rotational symmetry.
    // [(o,g), ..]: o is the rotational offset and
        g is the group size for the rotational
        symmetry.
    // : eg. send(c,src,r) == send( (c + o)%g, (src
        + o)%g, (r + o)%g)
    "symmetry_offsets": [[2, 16], [16, 32]],

    "hyperparameters": {
        "input_chunkup": 2, // Data at each GPU is
            partitioned into 2 chunks that can be
            independently routed
        "input_size": "1M"
    }
}
```

B TACCL Synthesizer in Detail

As explained in Section 5, TACCL's synthesizer has routing, heuristic ordering, and contiguity and exact scheduling stages. We provide a detailed description of each of these stages in this section. We first formally introduce some terms that we will use later. Let $\mathcal{C}$ denote the set of chunks that are required to be routed in the algorithm for collective *coll*. Let $\mathcal{R}$ denote the set of GPU ranks involved in *coll*. Let *coll*.precondition and *coll*.postcondition denote the precondition and post-condition of the collective respectively.The tuple $(c,r) \in coll.\text{precondition}, c \in \mathcal{C}, r \in \mathcal{R}$, if chunk c is present at rank r at the start of the collective. Similarly, the

$(c,r) \in coll$.postcondition if chunk c has to be present at rank r at the end of the collective. Further, let $\mathcal{L}$ denote the set of links, such that $(r1,r2) \in \mathcal{L}, r1 \in \mathcal{R}, r2 \in \mathcal{R}$ if there exists a link from rank $r1$ to rank $r2$ in the logical topology determined by the topology and communication sketch. Let $\mathcal{S}_r^{send}$ denote the set of switched destinations for rank r, such that $dst \in \mathcal{S}_r^{send}$ if link (r,dst) is a part of a switch-hyperedge. Similarly, $\mathcal{S}_r^{recv}$ denotes the set of switched sources for rank r, such that $src \in \mathcal{S}_r^{recv}$ if link (src,r) is a part of a switch-hyperedge. $\alpha(r1,r2)$, $\beta(r1,r2)$ are the alpha and beta costs respectively of the link $(r1,r2) \in \mathcal{L}$. The term $lat(r1,r2)$ is the sum of $\alpha(r1,r2)$ and $\beta(r1,r2)$ cost of the link, which denotes the total transfer cost of a single chunk over link $(r1,r2)$. Table 3 lists the variables that the TACCL's synthesizer solves for. We will describe each variable in detail in this section.

B.1 Routing

The main aim of the routing stage is to give us the path that every chunk takes in the collective. Our objective is to minimize the time (denoted by continuous variable $time$) it takes to reach the post-condition of the collective.

$$\text{Minimize} \quad time \tag{1}$$

The time taken for the collective algorithm is the latest time at which a chunk becomes available on a rank that is in the post-condition of the collective. We use a continuous variable $start[c,r]$ to denote the time that chunk c becomes available on rank r, and end up with the following constraints for $time$

$$time \geq start[c,r] \quad \forall (c,r) \in coll.\text{postcondition} \tag{2}$$

For chunks on ranks that belong to the collective's precondition, we set the start time to zero.

$$start[c,r] = 0 \quad \forall (c,r) \in coll.\text{precondition} \tag{3}$$

We also add correctness constraints in our formulation for routing - chunks are sent from a GPU rank only after they have been received on that rank. We introduce a continuous variable $send[c,src,r]$ to denote the time of sending chunk c from rank src to rank r and add the following constraint to our formulation:

$$send[c,src,r] \geq start[c,src] \quad \forall c \in \mathcal{C} \quad \forall (src,r) \in \mathcal{L} \tag{4}$$

We use a binary variable $is\_sent[c,src,r]$ to indicate if chunk c is sent over the link (src,r) in our algorithm. We note that the routing stage does not strictly respect bandwidth constraints of any link - the generated solution may send two chunks simultaneously over a link at the time cost of one chunk. The chunk start time on a rank will be determined only by the chunk send time on the source, independent of other chunk transfers on the link (eq. 5). LHS→RHS in the equation signifies an indicator constraint, i.e., if LHS is 1, RHS will hold.

$$is\_sent[c,src,r] \rightarrow start[c,r] = send[c,src,r] + lat(src,r) \quad \forall c \in \mathcal{C} \quad \forall (src,r) \in \mathcal{L} \tag{5}$$

Instead of bandwidth constraints, this encoding uses *relaxed bandwidth constraints*. They are expressed by aggregating the link transfer time of all chunks sent over a link and using it to to lower bound the total time of the algorithm (eq. 6). For switched connections, the total time is lower bounded by the sum of link transfer times of all chunks sent over all switched outgoing links from a source, and also by the sum of link transfer times for chunks received from all incoming links to a destination (eq. 7 and eq. 8).

$$time \geq \sum_{c \in C} (lat(src,r) * is\_sent[c,src,r]) \quad \forall (src,r) \in \mathcal{L} \tag{6}$$

$$time \geq \sum_{c \in C} \sum_{dst \in \mathcal{S}_r^{send}} (lat(r,dst) * is\_sent[c,r,dst]) \quad \forall r \in \mathcal{S}_{send} \tag{7}$$

$$time \geq \sum_{c \in C} \sum_{src \in \mathcal{S}_r^{recv}} (lat(src,r) * is\_sent[c,src,r]) \quad \forall r \in \mathcal{S}_{recv} \tag{8}$$

Based on the communication sketch, we also add constraints for `uc-max` and `uc-min` strategies for switch-hyperedges to maximize and minimize the number of links utilized in a switch respectively. We introduce a new binary variable $is\_util[src,r]$ for links (src,r) that are a part of a switch-hyperedge. This variable is 1 if any chunk is sent over link (src,r), and 0 otherwise.(eq. 9 and eq. 10). According to the switch-hyperedge strategy, we add this variable, weighted with a small constant γ, to the objective function (eq. 11). γ is negative for `uc-max` and positive for `uc-min`.

$$is\_util[src,r] >= is\_sent[c,src,r] \quad \forall c \in \mathcal{C} \forall (src,r) \in \mathcal{L} \tag{9}$$

$$is\_util[src,r] <= \sum_{\forall c \in C} is\_sent[c,src,r] \quad \forall (src,r) \in \mathcal{L} \tag{10}$$

$$\text{Minimize} \quad time + \gamma \times (\sum_{(src,r):\text{switched links}} is\_util[src,r]) \tag{11}$$

We also add symmetry constraints according to the symmetry offsets provided by user in the communication sketch. For a chunk c and link (src,r), we identify a rotationally symmetric chunk $\hat{c}$ and link $(\hat{src},\hat{r})$ and add the following constraints:

$$start[c,r] = start[\hat{c},\hat{r}] \tag{12}$$

$$send[c,src,r] = send[\hat{c},\hat{src},\hat{r}] \tag{13}$$

$$is\_sent[c,src,r] = is\_sent[\hat{c},\hat{src},\hat{r}] \tag{14}$$

MILP Variables	Explanation
Routing	
time	time spent in the collective algorithm
$start[c,r]$	time at which chunk c becomes available at GPU r
$send[c,src,r]$	time at which chunk c is sent from GPU src to GPU r
$is\_sent[c,src,r]$	indicates if chunk c is sent from GPU src to GPU r
$is\_util[src,r]$	indicates if any chunk is sent from GPU src to GPU r
Contiguity	
$is\_together[c,o,r]$	indicates if chunks c and o are sent to GPU r together from the same source, thus sharing the bandwidth and reducing the latency cost of transfer

Table 3: Variables used in TACCL's MILP formulation. Variables with prefix *is_* are binary variables and others are continuous variables.

Further, for chunks that start on one node and have a final destination on another node, we add inter-node transfer constraints which specify that at least one inter-node link will be used to transfer that chunk.

$$\sum_{(r_1,r_2)\in\mathcal{L}:r_1\in\text{node}_1,r_2\in\text{node}_2} is\_sent[c,r_1,r_2] \geq 1 \tag{15}$$

B.2 Ordering Heuristics

We start the heuristic ordering by determining the paths each chunk takes using the solution of the path encoding. We then consider the first link in every path as a candidate for scheduling a chunk transfer. Using heuristics like *chunk-with-shortest-path-until-now-first* and *chunk-with-longest-path-from-now-first*, we select a path (and thus a chunk) which should be scheduled in this round. We keep a running estimate of link time, which is the earliest time at which a chunk can be scheduled over the link. We also keep a running estimate of chunk time, which is the earliest time at which the next link transfer can be scheduled for a chunk. At the start, the link time for every link is 0 and the chunk time for every chunk is 0. When a path is chosen in the first round, the chunk associated with the path is scheduled to traverse the first link in the path. The link time of that link increases by link latency and chunk time of that chunk increases by link latency. The link candidate from the selected path is also updated to be the next link in the path. For the next rounds, we decide which path's candidate link to schedule next using the tracked link and chunk times along with the scheduling heuristics. This keeps going until we have scheduled a data transfer over all the links in all the paths. We find that the best heuristics differ for architectures with NVLinks and those with NVSwitches, in terms of whether to start selecting links to schedule in the same order as the paths or in the opposite order of the paths. The heuristic ordering has the following three outputs:

- chunk_order(r_1,r_2), an ordered list of chunks transferred along each link (r_1,r_2). If chunk c_1 is present before chunk c_2 in chunk_order(r_1,r_2), it denotes that c_1 is scheduled to be sent before c_2 over link (r_1,r_2).
- switch_send_order(r), an ordering on the chunks sent from a switch source r to any of the switch destinations $dsts$. If (c_1,dst_1) is present before tuple (c_2,dst_2) in switch_send_order(r), it means that a send of c_1 over link (r,dst_1) should be scheduled before a send of chunk c_2 over link (r,dst_2).
- switch_recv_order(r), an ordering on the chunks received on a switch destination r from any of the switch sources $srcs$. If (c_1,src_1) is present before tuple (c_2,src_2) in switch_recv_order(r), it means that a receive of c_1 over link (src_1,r) should be scheduled before a receive of chunk c_2 over link (src_2,r).

B.3 Contiguity and Exact Scheduling

Finally, we describe the formulation for the contiguity and exact scheduling stage. Given the link and switch ordering from the heuristic ordering stage, the aim of this stage is to find the sweet spot in the trade-off between lower link latency by sending multiple data chunks contiguously as a big data chunk and reduced pipelining benefits due to the big data-chunk transfer. We provide the main set of constraints in our formulation below, leaving out other less important constraints.

Our objective is still to minimize the time of the collective and constraints eq. 1-eq. 4 must still hold in this formulation. We add a new binary variable $is\_together(c_1,c_2,r)$ for all chunks c_1 and c_2 that are sent over the same link to rank r. If $is\_together(c_1,c_2,r)$ is 1, chunks c_1 and c_2 are sent as a single data-chunk over a link to rank r.

$$\begin{aligned} is\_together[c,o,r] &\rightarrow send[c,src,r] = send[o,src,r] \\ &\forall c,o \in \text{chunk\_order}(src,r) \quad \forall(src,r)\in\mathcal{L} \end{aligned} \tag{16}$$

The transfer time of a data chunk c along a link (src,r) will be determined by all other data chunks that it has to travel together with:

$$\begin{aligned} lat[c,src,r] =& \alpha(src,r) + \beta(src,r) * \\ & \Big(\sum_{o\in\text{chunk\_order}(src,r)} is\_together[c,o,r]\Big) \\ & \forall c \in \text{chunk\_order}(src,r) \quad \forall(src,r)\in\mathcal{L} \end{aligned} \tag{17}$$

$$\begin{aligned} start[c,r] =& send[c,src,r] + lat[c,src,r] \\ & \forall c \in \text{chunk\_order}(src,r) \quad \forall(src,r)\in(L) \end{aligned} \tag{18}$$

We also add strict bandwidth constraints for this formulation, allowing only one data chunk per link transfer time if the data chunks are not sent contiguously over the link. Let $pos(c, src, r)$ determine the position of chunk c in the chunk_order(src, r), then

$$\begin{aligned} \neg is\_together[c,o,r] \rightarrow send[o,src,r] \geq send[c,src,r] \\ +lat[c,src,r] \quad \forall c \in \text{chunk\_order}(src,r) \\ \forall o \in \text{chunk\_order}(src,r) \\ \text{if} \quad \text{pos}(o,src,r) \geq \text{pos}(c,src,r) \quad \forall (src,r) \in \mathcal{L} \end{aligned} \tag{19}$$

Similarly, we add bandwidth constraints for switch, allowing a source to send data to only one switched destination at a time, and a receiver to receive data from only one switched sender at a time. Let $sw-pos-send(c,r,dst)$ determine the position of tuple (c,dst) in the switch_send_order(r), and let $sw-pos-recv(c,src,r)$ determine the position of tuple (c,src) in the switch_recv_order(r), then,

$$\begin{aligned} send[o,r,dst_o] \geq send[c,r,dst_c] + lat[c,r,dst_c] \\ \forall (c,dst_c) \in \text{switch\_send\_order}(r) \\ \forall (o,dst_o) \in \text{switch\_send\_order}(r) \\ \text{if} \quad \text{sw-pos-send}(o,r,dst_o) \geq \text{sw-pos-send}(c,r,dst_c) \\ \forall r \in \mathcal{S}^{send} \end{aligned} \tag{20}$$

$$\begin{aligned} send[o,src_o,r] \geq send[c,src_c,r] + lat[c,src_c,r] \\ \forall (c,src_c) \in \text{switch\_recv\_order}(r) \\ \forall (o,src_o) \in \text{switch\_recv\_order}(r) \\ \text{if} \quad \text{sw-pos-recv}(o,src_o,r) \geq \text{sw-pos-recv}(c,src_c,r) \\ \forall r \in \mathcal{S}^{recv} \end{aligned} \tag{21}$$

C Standalone Experiments on Four Azure NDv2 Nodes

Figure 11 shows additional algorithm bandwidth and the speedup over NCCL graphs of TACCL for ALLGATHER, ALLTOALL, and ALLREDUCE on 4-node NDv2 cluster. We synthesize all collectives using the *ndv2-sk-1* communication sketch (see Section 7.1 for details), and lower them using 1 or 8 instances. We plot the best of the two algorithms over different buffer sizes.

TACCL's ALLGATHER algorithms are $10\% - 2.2\times$ faster than NCCL across all buffer sizes. For ALLTOALL, the *ndv2-sk-1* sketch is most effective for large buffer sizes, and helps generate algorithms that are up-to 46% faster than NCCL for buffer size greater than 1MB. TACCL ALLREDUCE algorithms are up-to 34% faster than NCCL for small buffer sizes and $1.9\times - 2.1\times$ faster than NCCL for larger buffer sizes.

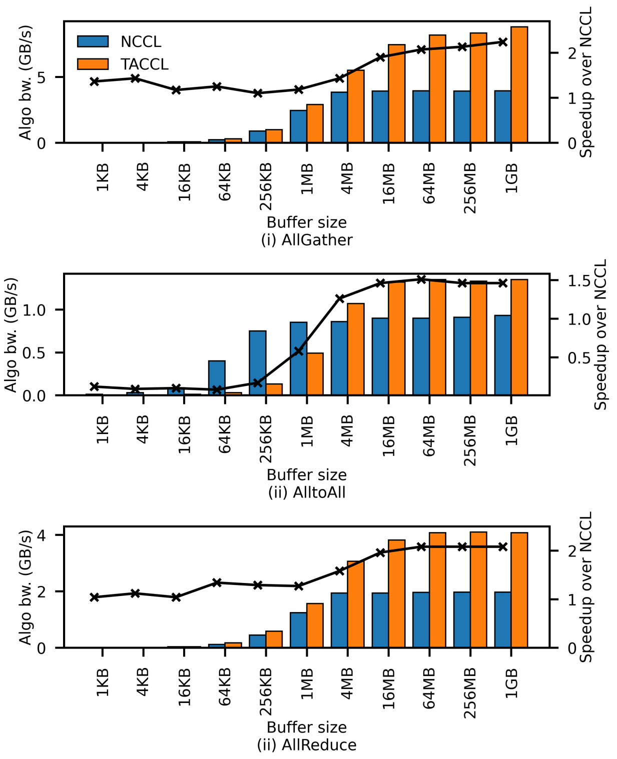


Figure 11: Algorithm bandwidth of TACCL algorithms compared against NCCL (left Y-axis) and their speedup over NCCL (right Y-axis) for ALLGATHER, ALLTOALL, and ALLREDUCE collectives on four NDv2 nodes.