Synthesizing Optimal Collective Algorithms

Zixian Cai[*]
Research School of Computer Science
Australian National University
Canberra, ACT, Australia
zixian.cai@anu.edu.au

Zhengyang Liu[*]
School of Computing
University of Utah
Salt Lake City, UT, USA
liuz@cs.utah.edu

Saeed Maleki
Microsoft Research
Redmond, WA, USA
saemal@microsoft.com

Madanlal Musuvathi
Microsoft Research
Redmond, WA, USA
madanm@microsoft.com

Todd Mytkowicz
Microsoft Research
Redmond, WA, USA
toddm@microsoft.com

Jacob Nelson
Microsoft Research
Redmond, WA, USA
jacob.nelson@microsoft.com

Olli Saarikivi
Microsoft Research
Redmond, WA, USA
olsaarik@microsoft.com

Abstract

Collective communication algorithms are an important component of distributed computation. Indeed, in the case of deep-learning, collective communication is the Amdahl's bottleneck of data-parallel training.

This paper introduces SCCL (for Synthesized Collective Communication Library), a systematic approach to synthesizing collective communication algorithms that are explicitly tailored to a particular hardware topology. SCCL synthesizes algorithms along the Pareto-frontier spanning from latency-optimal to bandwidth-optimal implementations of a collective. The paper demonstrates how to encode the synthesis problem as a quantifier-free SMT formula which can be discharged to a theorem prover. We show how our carefully built encoding enables SCCL to scale.

We synthesize novel latency and bandwidth optimal algorithms not seen in the literature on two popular hardware topologies. We also show how SCCL efficiently lowers algorithms to implementations on two hardware architectures (NVIDIA and AMD) and demonstrate competitive performance with hand optimized collective communication libraries.

CCS Concepts: • **Computer systems organization → Interconnection architectures**; • **Software and its engineering → Cooperating communicating processes**.

Keywords: GPU, Synthesis, Collective Communication, Interconnection, Network

1 Introduction

Recent trends in machine learning towards training and serving large models together with the stagnation of Moore's-law-induced compute performance has led system designers to include novel high-bandwidth interconnect networks both within and across nodes in distributed clusters. For instance, a DGX-1 server consists of two x86 processors and eight GPUs, interconnected by NVIDIA's NVLink network as shown in Figure 1. These networks' designs are motivated as much by the need to perform efficient Allreduce, a crucial primitive in machine learning, as well as by hardware considerations such as signal integrity, cooling and physical layout. A wide variety of similar accelerators with novel high-speed interconnects are used to train machine learning models today, including AMD's MI50 GPUs [1], Graphcore's IPUs [12] and Google's TPUs [11].

These novel topologies require novel communication kernels to maximize performance. Today these kernels are written and optimized manually. For instance, NVIDIA Collective Communication Library (NCCL) has two general algorithms for the supported operations such as Allreduce: a high-bandwidth ring algorithm and a low-latency tree algorithm. These implementations are manually written and they do not necessarily have the best performance for different topologies including DGX-1's. On one hand, repeating this

---

[*]Both authors contributed equally to the paper. The work was done during internships at Microsoft Research.
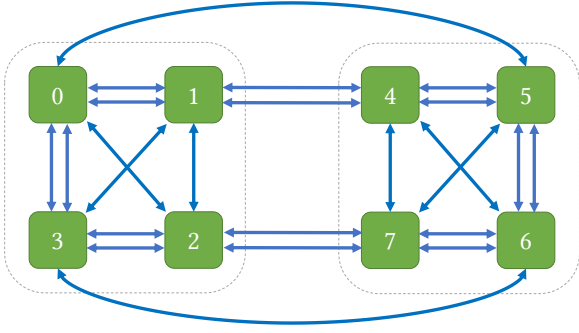
Figure 1. NVLink topology of an NVIDIA DGX-1.

manual effort for other communication primitives such as Alltoall or extending already implemented algorithms to a wide variety of hardware topologies is simply infeasible.

On the other hand, optimizing these communication kernels for performance for each topology and buffer size is crucial. For instance, we found 30% of the training time for the 8.3 billion parameter Megatron language model with model parallelism is spent inside Allreduce where each buffer is of medium size (10-100MB). Also, for data parallelism, the communication buffers could range from a few KBs (one layer) to a few GBs (the entire model). We expect this wide range of sizes as large models are developed and trained on larger distributed clusters.

In this paper, we automatically synthesize high-performance communication kernels. Given a topology, specified as a graph with bandwidth constraints on nodes and edges, and a communication primitive, specified as the pre- and post-condition on data location and computation on it, we generate (Section 3) a quantifier-free SMT formula that captures the set of all feasible algorithms that implement the primitive on the input topology. Exploring this space to appropriately minimize the number of communication steps or decrease the granularity of communication at each step, is a computationally difficult problem. We exploit an SMT solver to synthesize algorithms that explore this tradeoff along the Pareto frontier between latency-optimality and bandwidth-optimality. For every solution from the SMT solver, we automatically generate and lower (Section 4) high-performance implementations.

When using SMT, finding the right encoding can make all the difference for the feasibility of an approach. This paper details the important design choices in our encoding that help it scale to all of our hardware targets. We use the SMT encoding for non-combining collectives, such as Broadcast, while for combining collectives, such as Reduce, we employ a reduction back to the synthesis problem for non-combining collectives. This reduction generalizes a well known fact that some combining collectives may be produced by inverting a non-combining one, e.g. Reduce by inverting Broadcast.

We implement our approach in a tool called Synthesized Collective Communication Library (SCCL), which probes the target hardware topology, synthesizes algorithms for it using Z3 [8] and finally generates CUDA code that efficiently implements that algorithm. These algorithms are synchronous; at every step of the algorithm, one or more of the nodes send and/or reduce data from others.

Some of the algorithms we synthesize are novel, with no known counterparts in the literature occupying the same latency-bandwidth tradeoff. For example, we have produced a latency-optimal 2-step (4-step) algorithm for the Allgather (Allreduce) primitive in the DGX-1 topology (Figure 1) and a bandwidth-optimal 3-step (6-step) algorithm for the Allgather (Allreduce) primitive on the same topology. In addition to providing novel algorithms, our approach informs us when a combination of bandwidth and number of steps is *not possible*. This makes our synthesis approach a tool for probing the algorithmic properties that a given topology provides, which is useful for co-design of hardware interconnects with communication libraries. Our evaluation (Section 5) shows us that this approach scales and beats NCCL in almost all cases.

To summarize, the contributions of our paper are as follows:

- A formalization of the synthesis problem for non-combining collectives.
- A general strategy for encoding the synthesis problem for collective communications algorithms into the quantifier-free linear integer arithmetic (QF_LIA) sublogic of the SMT-LIB logic.
- A reduction from the synthesis problem for combining collectives to that for non-combining collectives.
- A description of how SCCL generates efficient code for the algorithms we synthesize on nodes with NVIDIA or AMD GPUs.
- An evaluation of SCCL's generated algorithms on common server topologies for deep learning workloads and a comparison against NCCL.

2 Overview

This section provides an overview of synthesizing latency- and bandwidth-optimal algorithms, using Allgather for the DGX-1 topology (Figure 1) as the running example.

2.1 Collective Communication Primitives

Collective communication primitives allow nodes in a networked system to perform operations on shared data. As an example, if each node has some input data, the Allgather primitive transfers these data to all of the nodes. One way to implement this is for each node to independently send its data to all other nodes. But, an algorithm in which the nodes collectively work together can be more efficient. The

efficiency of such algorithms depends on the network topology.

2.2 Topology

The network topology specifies how the nodes are connected with each other and the latency and bandwidth constraints on the links connecting them. Consider the DGX-1 topology shown in Figure 1. It consists of 8 GPUs (or nodes, in the above formalism) split into two groups $\{0, 1, 2, 3\}$ and $\{4, 5, 6, 7\}$. The nodes in each group are fully connected. In addition, there are four inter-group links as shown in the figure. These nodes are connected through NVLinks, with some nodes connected with two parallel NVLinks as shown in Figure 1.

The DGX-1's design was heavily influenced by the need to do gradient reduction for machine learning workloads. Specifically, this topology forms two non-overlapping rings: one connecting nodes $\{0, 1, 4, 5, 6, 7, 2, 3\}$ with two NVLinks per edge and another connecting $\{0, 2, 1, 3, 6, 4, 7, 5\}$ with one NVLink per edge. These rings are bidirectional and thus form 6 logical single-NVLink rings. The NCCL library implements Allgather by running 6 simultaneous ring algorithms as we discuss below.

2.3 Cost Model

We will characterize the communication cost using the (α, β) model [14]. That is, sending a message of size L along a link costs $\alpha + L \cdot \beta$ time. Here, α is the latency of communication and captures the *fixed* costs, such as the overhead of initiating a transfer or invoking a GPU kernel, and β is the inverse bandwidth of the link and captures *per-byte* costs, such as copying data into system buffers. Li *et al.* extensively studies the transfer time of buffers with different sizes over numerous GPU interconnections[16]. Their result show that with NVLinks, the transfer time stays almost constant up-to a large buffer size and only then it start to increase linearly. These results confirm that the (α, β) model is suitable for characterizing communication cost over NVLinks.

The cost of a collective algorithm for an input of size L will be of the form $a \cdot \alpha + b \cdot L \cdot \beta$. We call a the *latency cost* of the algorithm and b the *bandwidth cost* of the algorithm. Given a class of algorithms that implement a collective on a given topology, an algorithm is *latency-optimal* (*bandwidth-optimal*) if no other algorithm in the class has a lower latency (bandwidth) cost. Usually, there is a tradeoff between the latency cost and the bandwidth cost when designing collective algorithms. An algorithm with latency cost a and bandwidth cost b is said to be *Pareto-optimal* with respect to a class of algorithms if for every algorithm in the class with latency cost a' and bandwidth cost b', we have $a = a' \implies b' \geq b$ and $b = b' \implies a' \geq a$.

2.4 Bandwidth-Optimal Algorithm for DGX-1

As described above, the DGX-1 topology has 6 logical rings. Allgather for one ring can be implemented as follows. Each node simultaneously sends its data to the next node in the ring. In subsequent steps, each node stores the received data and sends it to the next node in the ring. In 7 steps all nodes will have received data from all of the other 7 GPUs. The 6-ring algorithm is a generalization of this algorithm. Each node splits its data into 6 chunks and executes the ring algorithm along each of the 6 rings, with one chunk per ring. If L is the size of the input data, each ring algorithm takes 7 steps and communicates $\frac{L}{6}$ bytes. Thus, the cost of the 6-ring algorithm is

$$7 \cdot \alpha + \frac{7}{6} \cdot L \cdot \beta$$

Each node has to receive at least $7 \cdot L$ amount of data, and it has an agglomerated incoming per-byte cost of $\beta/6$ (6 incoming NVLinks). Thus, any algorithm for Allgather has to take at least $\frac{7}{6} \cdot L \cdot \beta$ amount of time. Thus, this algorithm is bandwidth-optimal for the DGX-1 topology. But can we do better with the latency cost?

Using the techniques described in this paper, we have automatically synthesized an algorithm (Section 4) with cost

$$3 \cdot \alpha + \frac{7}{6} \cdot L \cdot \beta$$

To the best of our knowledge, this algorithm was not previously known. Moreover, we prove that this algorithm is Pareto-optimal with respect to the class of algorithms we call k-synchronous algorithms (Section 3.1).

2.5 Latency-Optimal Algorithm for DGX-1

The next question is whether we can improve upon the latency cost of the synthesized algorithm. If each node communicates its data along a binary tree instead of a ring, it would take at least 3 steps. Using the techniques described in this paper, we have automatically synthesized a better algorithm (Section 4) with cost

$$2 \cdot \alpha + \frac{3}{2} \cdot L \cdot \beta$$

Since the DGX-1 topology has a diameter of 2, this algorithm is latency-optimal. To the best of our knowledge, a latency-optimal algorithm for the DGX-1 was not previously known. This algorithm is Pareto-optimal with respect to the class of k-synchronous algorithms.

3 Algorithm Synthesis

This section demonstrates a method to synthesize Pareto-optimal algorithms that implement a collective primitive on a given topology. The Pareto-optimality is defined with respect to a class of algorithms we call k-synchronous algorithms.

We distinguish between *combining* collectives such as Allreduce and Reducescatter that combine chunks through
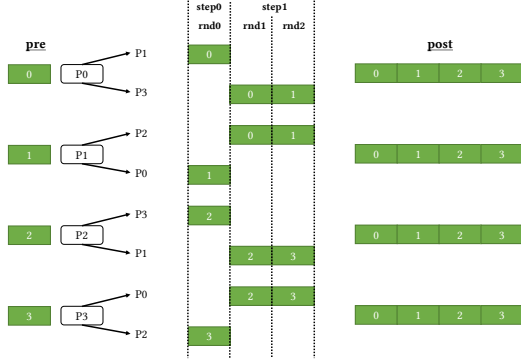
Figure 2. A 1-synchronous algorithm for Allgather on a ring topology.

Name	Relation
All	$[G] \times [P]$
Root	$[G] \times \{n_{root}\}$
Scattered	$\{(c, n) \in [G] \times [P] \mid n = c \bmod P\}$
Transpose	$\{(c, n) \in [G] \times [P] \mid n = \lfloor \frac{c}{P} \rfloor \bmod P\}$

Table 1. Common relations in pre- and post-conditions of collective primitives.

computation, and *non-combining* collectives such as Allgather and Broadcast that simply transfer data among nodes. We will focus on synthesizing non-combining collectives and show how to derive combining collectives from related non-combining ones.

3.1 *k*-synchronous Algorithms

Figure 2 shows the recursive-doubling [25] algorithm for Allgather for a ring topology of four nodes $P0, P1, P2, P3$ with four bidirectional links of equal bandwidth. This algorithm proceeds in two *steps*. In the first step, nodes at "distance" 1, namely $P0, P1$ and $P2, P3$ send their data to each other. Each node now has data from two nodes, which it communicates entirely with nodes at distance 2, i.e., nodes $P0, P3$ and $P1, P2$ in the second step. At the end, each node has data from every other node. Since the second step involves sending twice the amount of data as the first step, we say it has two *rounds* where in each round, it sends data. Thus, this step has a total of 3 rounds. Of the eight (unidirectional) links, this algorithm uses only four of them per step. To improve bandwidth utilization, a better option is to split the input data into equal-sized *chunks* and communicate them independently. For instance, the ring algorithm described in Section 2.4 uses 3 chunks per node.

The algorithm in Figure 2 and many classical collective algorithms [6, 25] are instances of *synchronous* algorithms. A synchronous algorithm proceeds in a sequence of synchronous communication *steps* with nodes waiting for other nodes to finish their rounds before starting the next step. Even if an implementation might not enforce a global barrier across the nodes, these algorithms choose the amount of data to communicate per step based on the bandwidth constraints so that the nodes finish each step at (roughly) the same time.

Many algorithms, like the one in Figure 2, communicate different numbers of chunks per step. We consider each step as consisting of multiple rounds with each node sending at most one chunk per unit-bandwidth on its outgoing links. Intuitively, the number of rounds in an algorithm controls

its bandwidth cost, while the number of steps controls its latency cost. A synchronous algorithm with S steps and R rounds is *k-synchronous* if $R \leq S + k$. The parameter k limits the amount of communication per step and allows an SMT solver to effectively search the space of algorithms bounded by that k.

3.2 Non-combining Collective Instance

Now we will provide a uniform formulation for representing *k*-synchronous algorithms for non-combining collectives. An instance of SynColl is a tuple $(G, S, R, P, B, pre, post)$, where

Parameters:
  – $G \in \mathbb{Z}_{\geq 0}$ is the global number of chunks
  – $S \in \mathbb{Z}_{\geq 0}$ is the total number of steps
  – $R \in \mathbb{Z}_{\geq 0}$ is the total number of rounds
Topology:
  – $P \in \mathbb{Z}_{\geq 0}$ is the number of nodes
  – $B \subseteq \mathcal{P}([P] \times [P]) \times \mathbb{N}$ is the bandwidth relation
Specification:
  – $pre \subseteq [G] \times [P]$ is the pre-condition
  – $post \subseteq [G] \times [P]$ is the post-condition

Note that for a set M we write $\mathcal{P}(M)$ for the power set of M, i.e., the set of all subsets. For an integer x, we write $[x]$ for the set $\{0, 1, \ldots, x\}$. Here, G, S, R are parameters to the desired *k*-synchronous algorithm. The rest are explained below.

3.2.1 Topology. P is the number of nodes in the topology. B gives a flexible way to express different bandwidth constraints we have seen in practice. In its most general form, B bounds the sum of chunks sent along a set of edges in a single round. A point-to-point communication link from s to d with maximum bandwidth (in chunks per round) b can be modeled by $(\{(s, d)\}, b) \in B$. Some topologies might limit the net outgoing bandwidth b from a certain node s. If E is the set of outgoing neighbors of s, we can model this by $(\{(s, e) \mid e \in E\}, b) \in B$. To model shared bus topologies, where only one node can send in a round, we include $(\{(a, b) \mid a \in N, b \in N\}, b)$ in B for the set of nodes N sharing the same link. Note that these constraints are per round, and when performing r_i rounds in step i, we simply multiply the bandwidth constraint by r_i.

Collective	pre	post
Gather	Scattered	Root
Allgather	Scattered	All
Alltoall	Scattered	Transpose
Broadcast	Root	All
Scatter	Root	Scattered

Table 2. Specifications of collective primitives as SynColl instances using a small set of common relations for pre- and post-conditions.

3.2.2 Collective Specification.

The *pre* relation specifies the nodes where the chunks reside at the beginning of the algorithm and the *post* relation specifies the set of nodes where a chunk needs to be transferred to. Table 1 specifies useful relations that can be used to specify common collectives as shown in Table 2. For instance, Allgather starts in a state where chunks are in the Scattered relation in Table 1. In other words, the c chunks of the input at node n are given chunk identifier $i \cdot P + n$ for $0 \leq i < c$. From this Scattered state, Allgather requires all the input chunks to be copied to all nodes, as specified by All relation in Table 1. Similarly, Broadcast requires all the chunks from the root n_{root} to be copied to all nodes.

While SynColl uses a global number of chunks G, it is more typical in existing literature to consider the per-node number of chunks C. We will use the per-node number when discussing the cost model and search algorithm in Sections 3.6 and 3.7 and when presenting our evaluation in Section 5. Note that how these two counts relate to each other is collective dependent: for Broadcast $G = C$, while for Allgather $G = P \cdot C$. The formalization must still use a global numbering of chunks, as some exotic collectives, e.g. MPI's Allgatherv, may not have a single per-node chunk count.

3.3 Candidate Solution

Given an instance of SynColl $(G, S, R, P, B, pre, post)$, a candidate solution is a pair (Q, T). Here Q is a sequence $r_0, r_1, \ldots, r_{S-1}$ such that $\sum_i r_i = R$ and denotes the number of rounds per step. T is a set of sends of the form (c, n, n', s), which specifies that chunk c must be sent from node n to node n' at step s. This defines a *run* defined as a sequence $V_0, V_1, \ldots, V_S$ such that $V_0 = pre$ and for all $0 \leq s < S$, V_{s+1} reflects the chunks present at a given node after accounting for the sends at step s:

$$V_{s+1} = V_s \cup \{(c, n') \mid (c, n) \in V_s \wedge (c, n, n', s) \in T\}$$

This candidate solution is a valid k-synchronous algorithm for the instance if $V_S \subseteq post$ and the following bandwidth constraint hold

$$\forall s \in [S], (L, b) \in B$$
$$|\{(c, n, n', s) \in T \mid (n, n') \in L\}| \leq b \cdot r_s$$

At each step s consisting of r_s rounds, the number of sends in each link should be bounded by the bandwidth constraint multiplied by r_s.

3.4 SMT Encoding for Non-combining Collectives

Given an instance, the SMT encoding incorporates the constraints above allowing the SMT solver to systematically search over candidate solutions (Q, T). It is straightforward to encode each r_s of Q as integer variables whose sum is R. In contrast, one has to be careful in encoding T. For instance, our initial attempt to encode every tuple $(c, n, n', s) \in T$ as a Boolean variable was not successful, because Z3, the SMT solver we used, did not solve larger problem instances fast enough. One way we were able to scale Z3 is to use a careful combination of Boolean, integer, and pseudo-Boolean constraints as we describe below.

We split the encoding of T into integer variables $time_{c,n} \geq 0$, indicating the earliest step a chunk c becomes available at node n and Boolean variables $snd_{n,c,n'}$ determining whether a node n sends chunk c to n' (at any step). To help with pruning the encoding, let $E = \{(n, n') \mid \forall (L, b) \in B \ (n, n') \in L \Rightarrow b > 0\}$, i.e., the pairs of nodes with non-zero bandwidth between them. Pseudo-Boolean constraints allow one to use Boolean variables as 0, 1 integers which we will use in the exposition below.

The following two constraints enforce the pre- and post-conditions

$$\forall (c, n) \in pre \ \ time_{c,n} = 0 \qquad \text{(C1)}$$
$$\forall (c, n) \in post \ \ time_{c,n} \leq S \qquad \text{(C2)}$$

If a chunk becomes available in a node, but is not part of the precondition, then the node should have received the chunk from some other node. For optimality, we also enforce that the node does not redundantly receive the chunk more than once.

$$\forall (c, n) \notin pre \ \ time_{c,n} \leq S \Rightarrow \Sigma_{(n',n) \in E} \ snd_{n',c,n} = 1 \quad \text{(C3)}$$

To send a chunk, it must exist on the source node before it is received on the destination node.

$$\forall (c, n) \in E \ \ snd_{n,c,n'} \Rightarrow time_{c,n} < time_{c,n'} \qquad \text{(C4)}$$

The following enforces the bandwidth constraint at all steps $1 \leq s \leq S$ and bandwidth constraint $(L, b) \in B$:

$$\Sigma_{(c,(n,n')) \in [G] \times L} \left(snd_{n,c,n'} \wedge time_{c,n'} = s \right) \leq b \cdot r_s \qquad \text{(C5)}$$

Note, we have multiplied the bandwidth constraints by r_s to allow r_s rounds at step s. Finally, the following bounds the total rounds R:

$$\Sigma_{1 \leq s \leq S}(r_s) = R \qquad \text{(C6)}$$

Once the problem instance has been encoded, the SMT solver will attempt to find a model M, which maps the variables $time_{c,n}$, $snd_{n,c,n'}$ and r_s to concrete values such that

Constraints C1 through C6 are satisfied. If a model exists then an algorithm (Q, T) can be constructed with:

$$Q = M(r_0), \ldots, M(r_{S-1})$$
$$T = \{(c, n, n', t) \mid M(snd_{n,c,n'}) \wedge M(time_{c,n}) = t + 1\}$$

If the SMT solver says the problem is unsatisfiable, then no algorithm exists for the problem instance.

3.5 Combining Collectives

It is well known that certain combining collectives are *inverses* of non-combining collectives. For instance, a Reduce algorithm can be generated by inverting an algorithm for Broadcast on a topology where all links have been reversed. Intuitively, whenever the Broadcast sends the same chunk to two different nodes, in its inverse the Reduce algorithm will receive the two *versions* of the chunk from these nodes and apply the reduction operation. The node will send the resulting chunk to the node it received the chunk from in the Broadcast. Similarly, we can generate Reducescatter algorithms by inverting Allgather algorithms.

Generally the inverting procedure works for any combining collective that has a single root node for each chunk. Notably, this does not include Allreduce, which replicates the result onto all nodes. For synthesizing Allreduce algorithms, we first notice that Allreduce can be expressed as a combination of Reducescatter followed by an Allgather. We synthesize Allreduce algorithms by synthesizing an Allgather algorithm and preceding it with its inverse Reducescatter algorithm.

3.6 Cost Model

Say we have synthesized a k-synchronous algorithm with C chunks, S steps, and R rounds. We will use the (α, β) cost model [14] to evaluate cost of this algorithm. Here, α is the latency of each link in the topology and β is the time taken sending a byte along a unit-bandwidth link. If the input data of L bytes is divided into C chunks, a step s with r_s rounds takes $\alpha + \frac{r_s}{C} \cdot L \cdot \beta$ time. Therefore, the entire algorithm will finish in time

$$S \cdot \alpha + \frac{R}{C} \cdot L \cdot \beta$$

3.7 Pareto-optimal Algorithms

The discussion above shows that for a given topology and a collective with an input size L, the cost of a k-synchronous algorithm can be characterized by the tuple $(S, \frac{R}{C})$. An algorithm with cost (a, b) is *Pareto-optimal* with respect to the class of k-synchronous algorithms if for every algorithm in this class with cost (a', b') we have $a = a' \Rightarrow b' \geq b$ and $b = b' \Rightarrow a' \geq a$. An algorithm with cost (a, b) is considered *latency-optimal* (*bandwidth-optimal*), if for every k-synchronous algorithm with cost (a', b') we have $a' \geq a$ ($b' \geq b$).

Note that latency- or bandwidth-optimal algorithms are not necessarily Pareto-optimal as they can be "wasteful" in the other parameter. Pareto-optimal algorithms form a *Pareto-frontier* with different algorithms in the frontier being better than others for a given input size L based on the α and β parameters of the topology.

Algorithm 1 Synthesizing Pareto-Optimal Algorithms

1: **procedure** Pareto-Synthesize(k, *Coll*, P, B)
2:     $a_l = Diameter(P, B)$
3:     $b_l = InvBisectionBandwidth(P, B)$
4:     $(pre, post) = Lookup(Coll)$ ▷ Table 2
5:     **for** $S = a_l, a_l + 1 \ldots$ **do**
6:         $A = \{(R, C) \mid S \leq R \leq S + k \wedge \frac{R}{C} \geq b_l\}$
7:         **for** $(R, C) \in A$ in ascending order of $\frac{R}{C}$ **do**
8:             $G = ToGlobal(Coll, C)$
9:             **if** $SMT(G, S, R, P, B, pre, post) = SAT$ **then**
10:                 Report synthesized algorithm (S, R, C)
11:                 **if** $\frac{R}{C} = b_l$ **then**
12:                     **return**
13:                 **break**

The procedure above systematically synthesizes Pareto-optimal k-synchronous algorithms. The inputs are the parameter k, the name of the collective to synthesize, and the topology parameters P, B (Section 3.2.1). The procedure computes the latency lower bound a_l from the diameter of the topology, and the bandwidth lower bound b_l from the inverse bisectional bandwidth of the topology. The procedure starts enumerating steps S starting with a_l. Then it generates A, the candidate set of tuples (R, C) that satisfy the round constraint and the inverse bandwidth constraint. Note that without the k parameter, this set would be unbounded. The procedure checks if a (S, R, C) algorithm exists in the increasing order of the bandwidth cost $\frac{R}{C}$ using the encoding discussed in Section 3.4. If one exists, the reported algorithm is guaranteed to be Pareto-optimal for the current steps S. As we increase the number of S, we get algorithms with lower bandwidth cost. Additionally, if the current bandwidth cost matches the lower bound b_l, the procedure returns. As we have already generated the Pareto-optimal algorithm with b_l bandwidth cost, it is not necessary to increase S further. Note, that it is possible for this procedure to never terminate as there can sometimes be unbounded number of Pareto-optimal algorithms for certain topologies and collectives. While the synthesis procedure above is for non-combining collectives, synthesis for combining collectives is similar (Section 3.5).

4 Code Generation

The prior section described a synthesis procedure for generating Pareto-optimal algorithms. This section describes a

tool called SCCL that implements this procedure and generates high-performance collective implementations for both NVIDIA and AMD GPUs.

Every synthesized algorithm, at its core, is a sequence of commands that describe *what* data needs to be sent (i.e., which chunk), *where* it needs to be sent (i.e., a source and destination), *when* it needs to be sent (i.e., during which synchronous step), and *with* which chunk(s) it needs to be reduced. SCCL generates SPMD multi-process C++ code combined with CUDA kernels that implement these commands.

Each GPU involved in the computation has its own code as part of a top-level switch statement. Communication between GPUs is enabled using CUDA IPC memory handles, which allows a GPU to access a remote GPU's memory using shared pointers. Thus, communication between GPUs simply involves writing data to appropriate buffers. However, there are a few crucial choices that impact the communication performance.

DMA engines and kernel copies: Data may be moved either by executing load or store instructions through a kernel, or by using a specialized DMA engine via cudaMemcpy. A kernel copy allows data movement and computation to be fused in a kernel while a DMA engine has a higher initial α cost but may have higher bandwidth, leading to a lower β cost. On NVLink, DMA engine bandwidth is about 10% better than kernel copy bandwidth, due to details of the wire-level protocol. Transfers are packetized, with each packet including a header (containing address, error correction data, etc.) and a variable-length payload. DMA engines are able to emit maximum-sized packets, but kernel copy packets are limited to the 128-byte cache line size.

Push and pull models: Each DMA engine is located on a particular GPU. Data movement between two GPUs can be executed by either the receiver's DMA engine (a *pull* model) or by the sender's DMA engine (a *push* model). Kernel copies have the same two approaches. This may have performance implications due to the link protocol: the push model only needs to send write request packets with a payload, whereas a pull model first sends request packets and then receives response packets with data. When communicating bidirectionally, the request packets reduce the bandwidth available for the response packets. Thus, even though the push model may require extra memory, we have found it to be up to 10% faster than the pull model.

Single and multiple kernels: One way to implement a synthesized algorithm is by emitting several kernels, one per step, which forces a global synchronization between steps and, as a consequence, introduces large overheads. Alternatively, SCCL fuses all steps into one kernel and thus we implement the synchronizations between GPUs as a fine-grained signal and wait mechanism with shared flags. In our single kernel implementation, each chunk for each connection has

a dedicated flag; a chunk on a GPU is valid only when the associated flag is set. There is a __threadfence_system() between the data movement operations and the operation to set the flag on the remote GPU signals that the transfer is complete.

Size and Number of Thread Blocks: SCCL dedicates a given number of thread blocks to each link and for each step, it uses the same number of thread blocks to communicate through that link. For different input sizes, the number of thread blocks significantly affects performance and in later sections we show how we empirically search for the fastest configuration for various input sizes.

5 Evaluation

This section demonstrates how we model and synthesize collectives for two multi-GPU systems with proprietary interconnects used for training large deep learning models. In both cases, we demonstrate 1) how to model the interconnect using SCCL, 2) what transport we utilize in lowering synthesized collectives, and 3) the Pareto-frontier of algorithms we find for the respective interconnects.

5.1 Hardware

The following section describes the hardware topology we model for our NVIDIA and AMD machines.

5.1.1 NVIDIA DGX-1: 8 V100 GPUs. A DGX-1 is a multi-GPU server sold directly by NVIDIA in addition to being a pay-as-you-go rental option in most cloud providers. It contains two 20-core Intel Xeon E5-2698 v4 processors with 512 GB DRAM split across the two sockets, along with 8 NVIDIA V100 GPUs, each with 32 GB of HBM2 memory. The GPUs are connected using NVIDIA's proprietary NVLink interconnect; each GPU has 6 25 GB/s NVLink ports. Figure 1 shows the topology: the 8 GPUs are interconnected with 2 non-overlapping Hamiltonian cycles. One of those cycles has two NVLink connections between each pair of GPUs. The GPUs are also connected to the CPUs by PCIe 3.0 x16 links, but we do not use them due to the wide disparity between per-GPU NVLink and PCIe bandwidth (~150 GB/s vs. ~14 GB/s). We also run synthesis on this platform.

5.1.2 Gigabyte Z52: 8 AMD MI50 GPUs. A Gigabyte Z52 system is a consumer grade multi-GPU system. It has two 64-core AMD EPYC 7002 processors with 1 TB DRAM split across the two sockets, as well as 8 AMD MI50 GPUs, each with 32 GB of HBM2 memory. 4 GPUs are connected to each socket with PCIe links, denoted by a box in Figure 3. Like NVIDIA, AMD also provides a proprietary high-speed interconnect called xGMI that links GPUs together. Each blue line is an xGMI link between a pair of GPUs. Note that the xGMI connections build two disconnected islands: 3 GPUs per island are on 1 socket while a lone GPU is on the *other* socket (i.e., GPU 1 and 5). The Gigabyte system uses PCIe
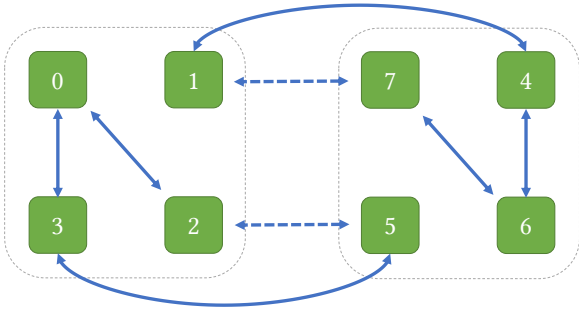
Figure 3. Topology of a Gigabyte MI50 8 GPU AMD System.

4.0 x16 links with measured bandwidth (~27 GB/s) that approaches xGMI's measured bandwidth (~33 GB/s). As such, we use PCIe to connect the rings.

5.2 Modeling Bandwidth Constraints

The hardware in this paper has distinct and interesting topologies. This section describes how we model those respective topologies in SCCL.

5.2.1 NVIDIA DGX-1: 8 V100 GPUs.
Each NVLink connection is point-to-point; thus our bandwidth constraints are simply the enumeration of each pair of GPUs connected via NVLink. As each NVLink connection can send 1 chunk per round, B has entries $(\{(n, n')\}, 1)$ for each pair of GPUs in one cycle and entries $(\{(n, n')\}, 2)$ for GPUs in the other.

5.2.2 Gigabyte Z52: 8 AMD MI50 GPUs.
Unlike NVLink, xGMI connections are not simply point-to-point but also transparently act as a router. For example, GPU 2 can send a message to GPU 3 even though they lack a physical connection: GPU 0 routes messages on GPU 2's behalf. However, this utilizes multiple links, and thus if GPU 0 concurrently sends a message to GPU 3, it can expect half the bandwidth of the link. We thus only model the direct connections in Figure 3. One way to connect the rings is to utilize PCIe and let GPU 1 connect to all other GPUs within its same socket (0, 2, and 3) and GPU 5 connect to GPUs within its same socket (4, 6, and 7). Because PCIe is shared, we could also enforce that only 1 PCIe connection occurs on every round, per socket. For example, the entry in B for the left socket is $(\{(0, 1), (1, 0), (1, 2), (2, 1), (1, 3), (3, 1)\}, 1)$. However, we were unable to utilize both xGMI and PCIe at the same time so our model of the bandwidth ignores the dotted xGMI connections in Figure 3. As such, we explicitly model the topology as a ring with GPUs 1 and 5 connecting the xGMI islands. Lastly, because the bisection bandwidth between the two xGMI islands is limited by the PCIe links that connect them, any bandwidth optimal algorithm will be limited by the bandwidth of these PCIe links. Therefore, we model the

Collective	C	S	R
Allgather/Reducescatter	6	7	7
Allreduce	48	14	14
Broadcast/Reduce	$6m$	$6 + m$	$6 + m$

Table 3. NCCL hand-written collectives and their chunks and steps. For Reducescatter C should be multiplied by 8.

same β cost for xGMI and PCIe and assume all links can send a single chunk per step.

5.3 NCCL and RCCL Baselines

We use NCCL (version 2.7.8-1) and RCCL (installed from ROCm 3.5.0) for baselines on NVIDIA and AMD hardware, respectively. NCCL is a hand-written and optimized communication library from NVIDIA. RCCL is a port of NCCL that uses the ROCm HIP compiler and targets AMD hardware. They share the same core algorithms and differ only in how they interact with the underlying hardware.

Table 3 gives an overview of the collectives that NCCL implements and number of chunks and steps they use on a DGX-1. NCCL's algorithms are all based on either rings or trees. However, Table 3 uses only ring algorithms, as we observed that on DGX-1 NCCL's trees are just simple paths, which are no better than using rings for any input size.

Our analysis of the chunks (C), steps (S), and rounds(R) is from our manual inspection of the NCCL source. For Reduce and Broadcast NCCL implements a pipelined algorithm, which chooses a multiplier m such that chunks stay approximately equally sized. Their running times are then $(6 + m) \cdot \alpha + \frac{6+m}{6m} \cdot L \cdot \beta$ and they get closer to bandwidth optimality as m gets larger.

As we show in the next section, SCCL is able to synthesize all these NCCL collectives and more, including Scatter, Gather, and Alltoall.

5.4 Synthesizing Collective Algorithms

Table 4 and Table 5 enumerate various algorithms we synthesize for NVIDIA DGX-1 and Gigabyte's AMD architecture. For each collective, we synthesize a latency and bandwidth optimal implementation, along with others that exist at various points along the latency-bandwidth curve. The first column combines collectives which are the inverse of each other (i.e., Scatter and Gather) and those that can be reduced to the non-combining collective using the reduction explained in Section 3.5 (e.g. Reduce to Broadcast).

5.4.1 Optimality.
Note we find many latency and bandwidth optimal algorithms for each collective, as we search over k-synchronous algorithms for different values of k. Consider the Allgather collective: we find many algorithms with various numbers of steps. However, the latency optimal algorithms (2 steps) dominate all others in the α term of the cost

Collective	C	S	R	Optimality	Time
Allgather	1	2	2	Latency	0.3 s
(Reducescatter)	2	3	3		0.8 s
	3	4	4		1.5 s
	4	5	5		2.3 s
	5	6	6		3.3 s
	6	7	7	Bandwidth	4.6 s
	6	3	7	Bandwidth	6.6 s
	2	2	3	Latency	0.9 s
Allreduce	8	4	4	Latency	0.3 s
	16	6	6		0.6 s
	24	8	8		1.3 s
	32	10	10		2.9 s
	40	12	12		5.6 s
	48	14	14	Bandwidth	12.8 s
	48	6	14	Bandwidth	23.0 s
	16	4	6	Latency	0.8 s
Broadcast	2	2	2	Latency	0.1 s
(Reduce)	6	3	3		0.3 s
	12	4	4		1.0 s
	18	5	5		8.5 s
	6	3	5		0.9 s
Gather	1	2	2	Latency	0.3 s
(Scatter)	2	3	3		0.9 s
	3	4	4		1.6 s
	4	5	5		2.7 s
	5	6	6		3.8 s
	6	7	7	Bandwidth	6.0 s
	6	3	7	Bandwidth	11.4 s
	2	2	3	Latency	1.0 s
Alltoall	8	3	3		2.6 s
	8	2	3	Latency	3.0 s
	24	8	8	Bandwidth	133.7 s
	24	2	8	Both	24.3 s

Table 4. DGX-1 collectives with chunks (C), steps (S) and rounds (R). Time includes both encoding and solving. For Reducescatter and Scatter C should be multiplied by 8.

Collective	C	S	R	Optimality	Time
Allgather	1	4	4	Latency	0.5 s
(Reducescatter)	2	7	7	Bandwidth	1.3 s
	2	4	7	Both	1.7 s
Allreduce	8	8	8	Latency	0.4 s
	16	14	14	Bandwidth	0.9 s
	16	8	14	Both	1.6 s
Broadcast	2	4	4	Latency	0.1 s
(Reduce)	4	5	5		0.2 s
	6	6	6		0.3 s
	8	7	7		0.5 s
	10	8	8		0.6 s
Gather	1	4	4	Latency	0.4 s
(Scatter)	2	4	7	Both	1.8 s
Alltoall	8	4	8	Both	8.2 s

Table 5. AMD collectives with chunks (C), steps (S) and rounds (R). Time includes both encoding and solving. For Reducescatter and Scatter C should be multiplied by 8.

5.4.2 Synthesizing All Collectives.
Collective communication libraries need to support a large and diverse set of hardware architectures. Efficiently implementing latency and bandwidth optimal algorithms for various topologies is time-consuming and error-prone. SCCL's synthesis based approach allows it to easily extend the set of algorithms through search: SCCL synthesizes algorithms for Alltoall, Gather and Scatter where no such counterparts exist in NCCL.

5.4.3 Synthesis time.
The longest synthesis time is just over 2 minutes and most of the time under 10 seconds. The synthesis problem is non-trivial and its complexity is defined by both the collective, as well as the hardware topology we synthesize for. The clever encoding described in Section 3.4 was critical for achieving these fast synthesis times. As a point of comparison, synthesizing the 24-chunk 8-step bandwidth-optimal Alltoall algorithm with a more direct encoding with a Boolean variable for each tuple $(c, n, n', s) \in T$ did not finish within 60 minutes. With the better encoding the synthesis finishes in just over 2 minutes.

5.5 Performance Evaluation
In this section, we compare SCCL's generated algorithms with NCCL and RCCL on the NVIDIA and AMD hardware. Our code generation uses a protocol similar to the simple protocol (i.e., NCCL_PROTO=Simple). Thus, we use NCCL with the simple protocol as our baseline. We investigate the performance of Allgather, Allreduce, and Alltoall as they are popular primitives in different workloads including machine learning. For each hardware platform and collective, we generate multiple algorithms; for each algorithm, we lower using (1) a single kernel-launch, or (2) multiple cudaMemcpy calls with one per step. Each algorithm uses a push model for

model. Likewise, the bandwidth optimal algorithms dominate all others with their low ratio of rounds to chunks (7/6). We synthesized algorithms in the 0-synchronous class ($R = C$) as the code generation is much easier.

Note that NCCL's Allgather algorithm is bandwidth optimal, and while it is also the lowest latency algorithm that NCCL provides, it is not latency optimal. We are able to synthesize both a bandwidth optimal algorithm with better latency (6-chunks 3-steps 7-rounds), as well as a latency optimal algorithm. In general, our synthesized latency optimal algorithms have no counterpart in NCCL and our bandwidth optimal algorithms are better than NCCL's for Allgather, Broadcast, and Reduce.

copying and when SCCL is compared with NCCL, we exhaustively search the size and the number of thread blocks and report the best performing combination for both SCCL and NCCL. See Section 4 for more details.

Figure 4 compares SCCL's generated code for Allgather with NCCL's Allgather. A point on Figure 4a (x,y) shows the running time in y milliseconds as a function of send input buffer size in x Kbytes while a point on Figure 4b shows the y speedup of SCCL's generated code over NCCL's Allgather as a function of send input buffer size in x Kbytes. We plot one line per algorithm denoted as (C, S, R) for respectively chunks, steps, and rounds as defined in Table 4. To show the impact of our lowering, we plot two versions of a bandwidth optimal algorithm $(6, 7, 7)$ (which utilizes a push-copy) and $(6, 7, 7)$ cudaMemcpy. The latter of which shows the significant impact lowering can have on the performance. To simplify the figure, we only show algorithms that were faster on at least one input size we experimented with. As it can be seen from the lines, SCCL is up-to 2.2× faster than NCCL's Allgather on small sizes and 1.14× faster on larger sizes. It is possible for SCCL to automatically switch between multiple implementations based on the input size. In which case, SCCL will consistently outperform NCCL.

Likewise, Figure 5 shows the running time in milliseconds (Figure 5a) or speedup (Figure 5b) for Allreduce as a function of the receive input size. Each line denotes (C, S, R) for respectively chunks, steps, and rounds, respectively. With the exception of 4 middle sizes, SCCL beats NCCL's Allreduce with an 8-chunk algorithm for small input sizes by up-to 1.8× and with a 48-chunk algorithm for large input sizes by up-to 1.06×.

Alltoall is a complex algorithm which is very difficult to write efficiently by hand. Unlike the prior collectives, NCCL does not natively support Alltoall; instead, NCCL suggests using N point-to-point exchanges (for N GPUs) and thus its resulting algorithm is neither bandwidth nor latency optimal. Because SCCL uses program synthesis to generate optimal algorithms, it is able to synthesize three Alltoall algorithms in a matter of minutes. Figure 6a shows the latency in milliseconds of SCCL and NCCL as a function of input size while Figure 6b shows speedup over NCCL again as a function of input size. Each line denotes (C, S, R) for respectively chunks, steps, and rounds and demonstrates a speedup of over 6.8× for large input sizes and over 1.4× for small input sizes, depending on whether we pick a latency or bandwidth optimal implementation from SCCL. This significant speedup really shows off the power of SCCL's automated approach to building algorithms tailored specifically to a hardware architecture.

Lastly, we demonstrate Allgather on the Gigabyte AMD workstation. Like the other plots, a point on Figure 7 (x,y) shows the latency or speedup y for Allgather as a function of the receive input size in bytes x. We plot two algorithms, $(1, 4, 4)$ and $(2, 7, 7)$; it is clear that (i) the lower latency algorithm $(1, 4, 4)$ is better at smaller input sizes, (ii) the higher bandwidth algorithm $(2, 7, 7)$ is faster for large input sizes, and (iii) SCCL's generated code is faster than RCCL for large sizes but slower for medium and small sizes. The Gigabyte machine, in particular, is new hardware and SCCL can synthesize new algorithms and implementations for it; this shows SCCL can help design future interconnects and co-design them with communication libraries.

These graphs in concert show that SCCL is able to synthesize algorithms along the Pareto-optimal frontier and also lower than to hardware so as to be competitive with a hand optimized baseline.

6 Related Work

The message passing interface (MPI) [9] is a widely-used standardized abstraction for communication primitives in a multi processor system. Implementations of MPI provide reliable and portable implementations of collective primitives. Efficient algorithms for implementing these primitives is a long-studied research area [6, 20, 25], including optimized algorithms for specific architectures like mesh, hypercube, or fat-tree[4, 5, 22] and for clusters of shared-memory processors [21, 24, 26, 27]. The class of k-synchronous algorithms studied in this paper is designed to include many of the algorithms proposed in these works and implemented in popular MPI implementations such as MPICH [25] and OpenMPI [10].

We evaluated OpenMPI, either through builtin CUDA capability or through Unified Communication X (UCX) [28]. They lack custom implementations for architectures such as the DGX-1, and result in subpar performance compared with our NCCL baselines. NCCL [18] is a library for multi NVIDIA GPU systems and it utilizes the underlying hardware transport such as NVLink, NVSwitch or Infiniband for an efficient implementation of collective primitives. RCCL [2] is a port of NCCL for AMD GPUs and the HIP compiler suite. While these libraries provide efficient implementations for a limited set of algorithms, SCCL is able to synthesize a wide range of algorithms suitable for different input sizes and generate collective primitives that are not even a part of standard MPI set.

There are also hybrid algorithms [3, 6] that switch between latency- and bandwidth-optimal algorithm along each dimension of a mesh network. However, to the best of our knowledge, these prior works do not seek to identify algorithms that are Pareto-optimal for a given topology. In contrast to these prior works, the goal of this paper is to automatically synthesize Pareto-optimal algorithms for a given topology.

There are also hierarchical approaches to implement collective primitives in distributed systems. Horovod [23] implements collective primitives by using NCCL locally in node and MPI across nodes. Others such as BlueConnect [7] and

(a) Average latency

(b) Speedup
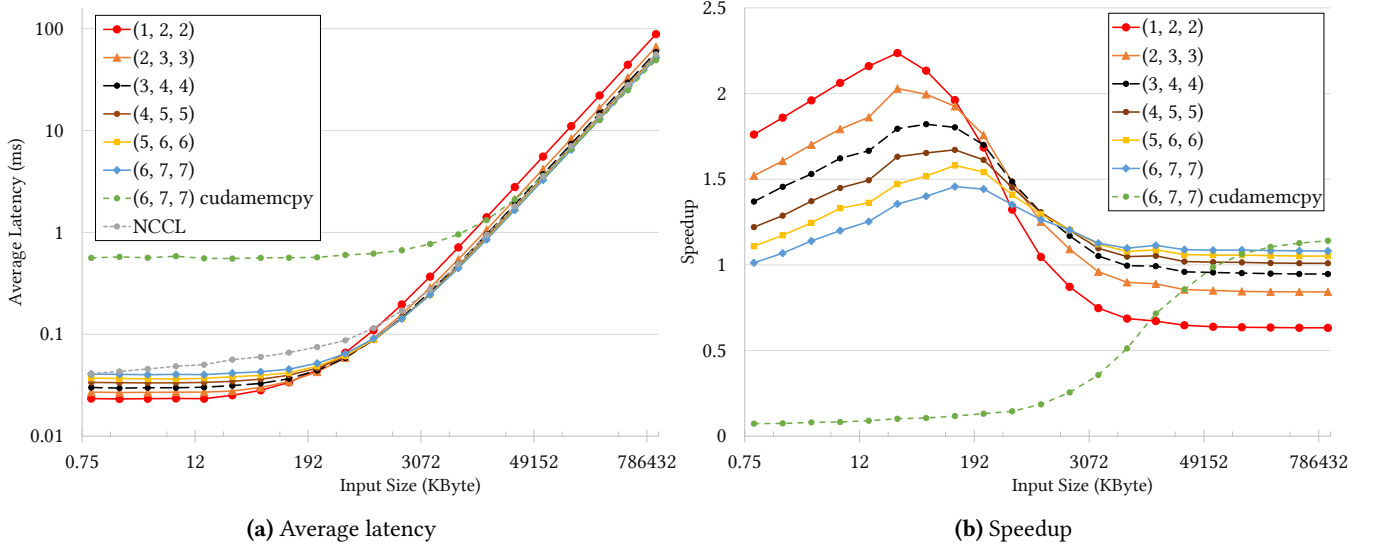
Figure 4. Allgather performance comparison with NCCL

(a) Average latency

(b) Speedup
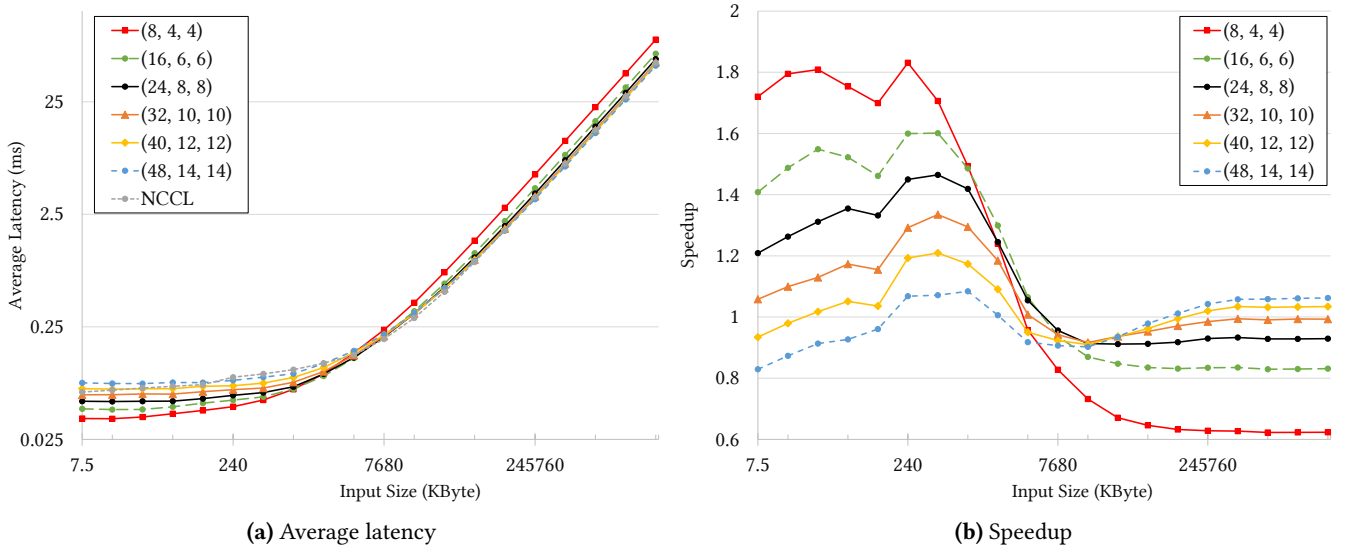
Figure 5. Allreduce performance comparison with NCCL

PLink [17] exploit the hierarchical network topology of a cloud system or a data center to improve the performance of collective primitives. In this paper, we focus on synthesizing algorithms for a single node with multiple GPU, while the above approaches are beneficial on multi node systems.

Motivated by recent resurgence in machine-learning workloads, recent research has focused on optimizing the communication of distributed machine learning. Blink [29], the closest to our work, automatically synthesizes bandwidth-efficient collective primitives for a given topology. This work is based on packing spanning trees and is suitable for one-to-many collective primitives such as broadcast and reduce,

and implements Allreduce as a reduce followed by a broadcast. Blink is not guaranteed to generate bandwidth-optimal algorithms as it heuristically selects a few trees based on an approximate spanning-tree packing algorithm. Moreover, Blink's focus is not on generating latency-optimal algorithms. In contrast, this work generates latency- and bandwidth-optimal algorithms for a given topology. There are also other works [13, 15, 19, 30] on optimizing distributed machine learning that do so by overlapping computation and communication and are orthogonal to this work.

(a) Average latency

(b) Speedup
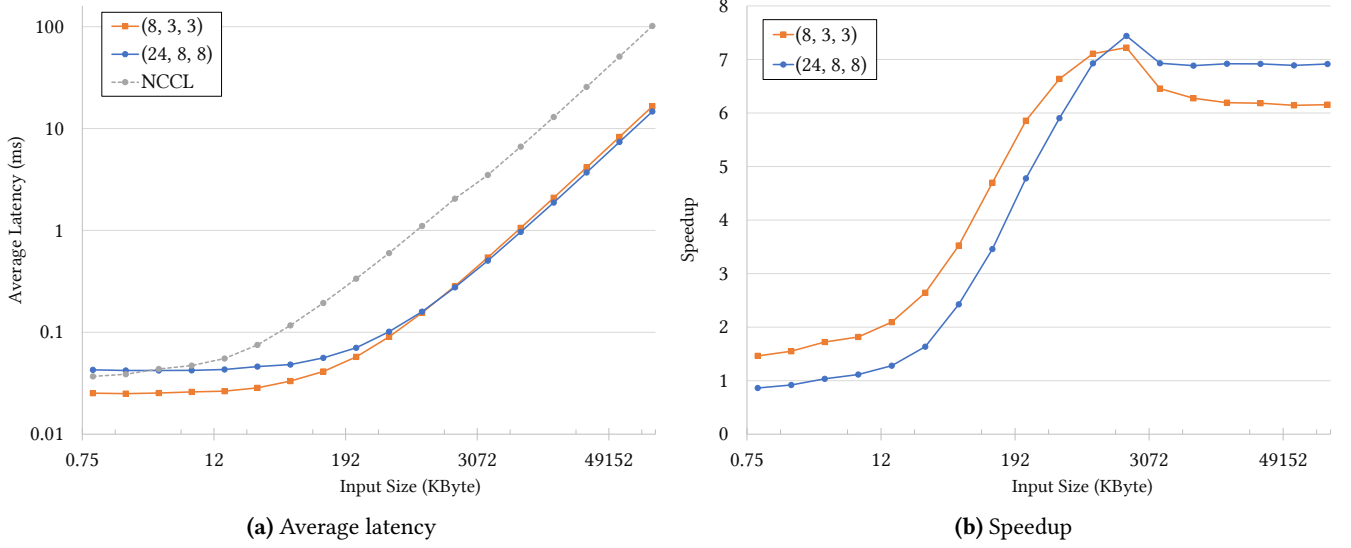
Figure 6. Alltoall performance comparison with NCCL

(a) Average latency

(b) Speedup
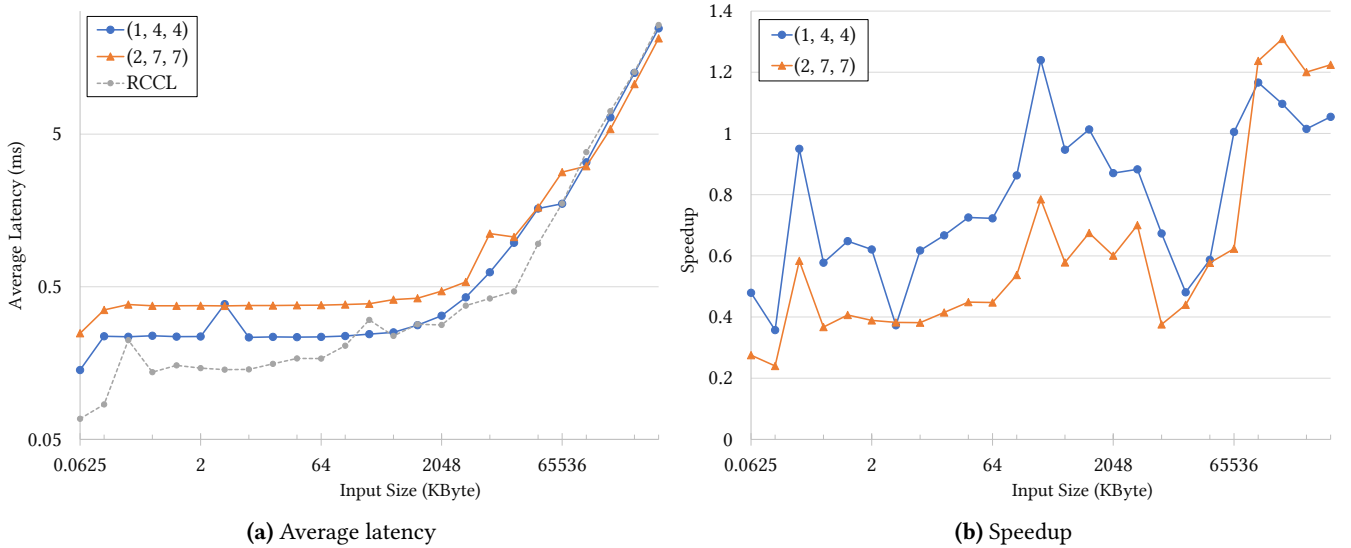
Figure 7. Allgather performance comparison with RCCL

7 Conclusion

This paper introduces SCCL: a systematic method to synthesize algorithms in the Pareto-frontier spanning from the latency-optimal algorithm to the bandwidth-optimal algorithm for a given collective on an input topology. We characterize a class of algorithms that captures a broad set of known algorithms and prove Pareto-optimality of both known algorithms and synthesized new algorithms. We automatically generate an implementation of these algorithms that is competitive with manually hand-tuned communication kernels in use today.

References

[1] AMD Radeon Instinct MI50 2020. AMD Radeon Instinct MI50 Accelerator. https://www.amd.com/system/files/documents/radeon-instinct-mi50-datasheet.pdf.

[2] AMD RCCL Library 2020. ROCm Communication Collectives Library. https://github.com/ROCmSoftwarePlatform/rccl.

[3] Mike Barnett, Satya Gupta, David G Payne, Lance Shuler, Robert van de Geijn, and Jerrell Watts. 1994. Building a high-performance collective communication library. In *Supercomputing'94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. IEEE, 107–116.

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

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

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

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

[8] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. https://doi.org/10.1007/978-3-540-78800-3_24

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

[10] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 97–104.

[11] Google TPU 2020. Google Cloud TPU. https://cloud.google.com/tpu.

[12] Graphcore IPU 2020. Graphcore Intelligence Processing Unit. https://www.graphcore.ai/products/ipu.

[13] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. 2019. TicTac: Accelerating distributed deep learning with communication scheduling. (March 2019).

[14] Roger W Hockney. 1994. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel computing* 20, 3 (1994), 389–398.

[15] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-based parameter propagation for distributed DNN training. (March 2019).

[16] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 94–110. https://doi.org/10.1109/TPDS.2019.2928289

[17] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. 2020. PLink: Discovering and Exploiting Locality for Accelerated Distributed Training on the public Cloud. In *Proceedings of Machine Learning and Systems 2020*. 82–97.

[18] NVIDIA NCCL Library 2020. NVIDIA Collective Communications Library. https://github.com/NVIDIA/nccl.

[19] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 16–29.

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

[21] Peter Sanders and Jesper Larsson Träff. 2002. The hierarchical factor algorithm for all-to-all communication. In *European Conference on Parallel Processing*. Springer, 799–803.

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

[23] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. arXiv:1802.05799 [cs.LG]

[24] Steve Sistare, Rolf Vandevaart, and Eugene Loh. 1999. Optimization of MPI collectives on clusters of large-scale SMP's. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. 23–es.

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

[26] Vinod Tipparaju, Jarek Nieplocha, and Dhabaleswar Panda. 2003. Fast collective operations using shared and remote memory access protocols on clusters. In *Proceedings International Parallel and Distributed Processing Symposium*. IEEE, 10–pp.

[27] Jesper Larsson Träff. 2002. Improved MPI all-to-all communication on a Giganet SMP cluster. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 392–400.

[28] UCX 2020. Unified Communication X. https://www.openucx.org/.

[29] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. 2020. Blink: Fast and Generic Collectives for Distributed ML. In *Conference on Machine Learning and Systems (MLSys 2020)*.

[30] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. 2017. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 181–193.

A Artifact Appendix

A.1 Abstract

This artifact contains the source files for SCCL. SCCL has two parts; a synthesizer for synthesizing the optimal communication schedules for a given topology and a code generator that lowers the synthesized schedule to CUDA code. The synthesizer and the code generator can be executed on any modern x86-64 computers but the evaluation of lowered code requires a system with CUDA-enabled GPUs and peer-to-peer access. The lowered code in this paper was evaluated on an NVIDIA DGX-1 and a Gigabyte Z52 system.

This artifact provides instructions on how to set up the environment, build and launch the docker image and do a test run of SCCL. It will also give the command lines to reproduce the results in paper, and finally, it will discuss how to use SCCL to synthesize schedules for custom topologies and parameters.

A.2 Artifact check-list (meta-information)

- **Algorithm:** SCCL
- **Program:** sccl.py is a python script that automatically synthesizes communication schedules and lowers them to CUDA source code.
- **Compilation:** Each generated code comes with a Makefile which requires NVCC and MPICC for compilation.
- **Run-time environment:** Linux operating system, CUDA run-time, and MPI run-time.
- **Hardware:** An NVIDIA DGX-1 with 8 V100 GPUs connected with NVLinks and a Gigabyte Z52 system with 8 MI50 GPUs connected with PCI and xGMI.
- **Metrics:** Evaluating the synthesis time and the latency of generated collective communication primitives.
- **Output:** A schedule for transferring buffers of data required for the desired collective communication primitive along with the lowered CUDA code.

- **Experiments:** Code generation for different versions of Allreduce, Allgather, and Alltoall on different topologies and executing them.
- **Publicly available?:** The code is available per request.

A.3 Description

A.3.1 How delivered. The source code can be accessed through Github[1].

A.3.2 Hardware dependencies. Out experiments were evaluated on an NVIDIA DGX-1 and a Gigabyte Z52 system. The DGX-1 machine has 8 V100 NVIDIA GPUs with NVLink connection among them and has dual Intel Xeon E5-2698 v4 processors with a total of 512 GB host memory. The Z52 system consists of 8 MI50 AMD GPUs connected via xGMI and PCI and runs with dual AMD EPYC 7002 processors with a total of 1TB host memory.

A.3.3 Software dependencies. Our experiments were evaluated on Ubuntu version 20.04, kernel version 4.19 with NVIDIA Docker version 2.5.0, CUDA version 10.2, OpenMPI version 4.0.2, Python version 3.8.5 and Z3 version 4.8.8, and the performance of our generate code are compared with NCCL version 2.7.8-1. CUDA, OpenMPI, Python, Z3 and NCCL are automatically installed when building the docker image.

A.4 Installation

The installation is done through docker. The `build_docker.sh` in the downloaded tar file includes all of the required steps to get the docker container running.

A.5 Evaluation and expected result

A.5.1 Evaluating the synthesizer. SCCL can be queried to synthesize different collective communication primitives. For example, it can synthesize an Allgather algorithm with 6 chunks in 7 steps and 7 rounds on a DGX-1. This will take a few seconds to execute and find the schedule for sending the chunks across the network. Command line for this example is given in the README.

The output of SCCL's shows the synthesized schedule which follows the following pattern

```
send 1 from 0 to 3 at step 0
send 2 from 0 to 2 at step 0
send 3 from 0 to 1 at step 0
```

The output specifies when and what chunk is communicated between a pair of GPUs. GPU i's chunks are identified by $[6i, 6i + 5]$ (6 chunks per GPU) where $i \in [0, 7]$ is one of the 8 GPUs.

An adjacency matrix is followed, displaying the topology of DGX-1 where rows (columns) correspond to sources (destinations) and the value represents the number of parallel chunks that can be transferred from the source GPU to the destination GPU in a round.

The bandwidth utilities per step are displayed after the topology matrix. This corresponds to the schedule that SCCL synthesizes for each step. A value at row s and column d represents the number of chunks sent from GPU s to GPU d and normalized by the number of rounds in that step. This is limited by the entry of the topology matrix at (s, d). After the bandwidth utilization matrix, an overall link utilization is displayed.

Once SCCL synthesized the schedule for the communication, a CUDA implementation following the schedule is generated.

Table 4 and 5 can be generated by following the README file.

A.5.2 Evaluating the generated CUDA code. This section describes the instructions for reproducing the numbers in Figures 4, 5, 6 and 7. The OSU Micro-Benchmarks (OMB) was adopted for the performance evaluation of the generated CUDA code. The instructions for executing the generated CUDA code through OMB is given in the README file.

A.6 Experiment customization

SCCL can synthesize collectives with customized topologies, chunks, steps and rounds by expressing the network topology and setting the command line arguments. The README file includes the instructions.

---

[1]https://github.com/parasailteam/nccl/blob/synthesizer/ppopp-ae/sccl-artifact.tar.gz