MSCCLang: Microsoft Collective Communication Language

Meghan Cowan
meghancowan@microsoft.com
Microsoft Research
Redmond, WA, USA

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

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

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

Yifan Xiong
yifan.xiong@microsoft.com
Microsoft Research
Beijing, China

ABSTRACT

Machine learning models with millions or billions of parameters are increasingly trained and served on large multi-GPU systems. As models grow in size and execute on more GPUs, collective communication becomes a bottleneck. Custom collective algorithms optimized for both particular network topologies and application-specific communication patterns can alleviate this bottleneck and help these applications scale. However, implementing correct and efficient custom algorithms is challenging.

This paper introduces MSCCLang, a system for programmable GPU communication. MSCCLang provides a domain specific language for writing collective communication algorithms and an optimizing compiler for lowering them to an executable form, which can be executed efficiently and flexibly in an interpreter-based runtime. We used MSCCLang to write novel collective algorithms for AllReduce and AllToAll that are up to 1.9× and 1.3× faster than hand-optimized implementations, respectively.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Compilers**; **Communications management**; • **Theory of computation → Concurrency**.

KEYWORDS

GPU, Collective Communication, Compilers

ACM Reference Format:
Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. 2023. MSCCLang: Microsoft Collective Communication Language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3575693.3575724

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03…$15.00
https://doi.org/10.1145/3575693.3575724

1 INTRODUCTION

Recent trends in machine learning (ML) point towards model sizes growing at a much faster rate than a single GPU's memory capacity and computational power [1, 29]. This necessitates distributing model parameters across multiple GPUs [9, 25, 39] for both model training as well as for model inference. The resulting cost of communication increases as a percentage of total GPU execution time as models become larger. For instance, training Resnet50 [17] with ≈100MB of parameters spends 3% of the time in communication [35], while training DeepLight [10] with ≈2GB of parameters spends 79% of its time in communication on the same distributed system. Therefore, optimizing communication will be critical for future ML workloads.

Communication kernels in ML workloads support Message Passing Interface (MPI) collective communication operations, such as AllReduce, AllGather, and AllToAll [14]. These collectives cooperatively exchange data across GPUs using various communication algorithms [41]. Vendor libraries, like NCCL [27] and RCCL [33], provide high-performance implementations of a few standard algorithms, namely Ring and Tree. Recent research [4, 6, 44, 45] has shown the promise of custom algorithms that are tailored for underlying interconnection topologies and input sizes. However, these works do not implement low-level optimizations such as pipelining, parallelization, and fusion that are necessary for maximizing performance. Partly to avoid the complexity of implementing such low-level optimizations, many works [6, 44, 45] compose existing vendor library implementations; doing so not only incurs the cost of multiple kernel launches but also loses the opportunity to perform optimizations that cross kernel boundaries.

This paper proposes Microsoft's Collective Communication Language (MSCCLang) which is a unified system for generating high-performance implementations of custom communication algorithms. MSCCLang consists of a *domain-specific language (DSL)* for specifying communication algorithms, a *compiler* for generating high-performance executables from these high-level specifications, and an efficient *runtime* for execution. For a given collective communication algorithm, a developer can explore different implementations and optimizations in MSCCLang without fearing data races/deadlocks or writing any C/CUDA code while enjoying the performance of a hand-written code. Additionally, MSCCLang can automatically check whether an implementation properly implements a collective before running on hardware. Lastly, the runtime is API-compatible with NCCL allowing existing ML workloads to easily convert to MSCCLang, inherit NCCL's support of diverse set

GPUs and inter-connections, and safely fall over to NCCL for scenarios unsupported in MSCCLang. MSCCLang is publicly available at https://github.com/microsoft/msccl and https://github.com/microsoft/msccl-tools.

We evaluate MSCCLang on two distributed GPU systems: a cluster of 8×A100 nodes and a cluster of 16×V100 nodes. We show that for a given algorithm, MSCCLang implementations match, and often beat, the performance of a hand-written implementation. This includes an AllToAll algorithm on multiple nodes that is up to 1.3× faster than a hand-optimized implementation and Ring AllReduce algorithm that is up to 1.9× faster than NCCL's optimized implementation. Additionally, we make a case for custom collectives by replacing simple point-to-point communication with a new collective called AllToNext. Lastly, MSCCLang system is used to serve a public facing language model on 8×A100 GPUs and training a large Mixture-of-Experts model for speech, language, and vision on 256×A100 GPUs at Microsoft providing 1.22-1.29× and 1.10-1.89× speed up, respectively.

2 MSCCLANG EXAMPLE

This section introduces MSCCLang through a running example, hierarchical AllReduce, and introduces common terminology used throughout the paper.

Terminology. In a cluster of N nodes or machines with G GPUs each, the *rank* of a GPU is identified by a tuple (n, g) where n is the node index and g is the GPU index within the node, or alternatively by the integer value $n \times G + g$. We refer to GPUs by their tuple and single integer ranks interchangeably.

Collectives operate on *buffers* of data divided into *chunks*, which represent contiguous spans of elements with a uniform size. Chunks are the finest granularity that data is sent with in a collective.

Hierarchical AllReduce. Figure 1 shows the workings of this algorithm. For a topology of $N(=2)$ nodes and $G(=3)$ GPUs per node, the algorithm splits the input buffer into $N \times G(=6)$ chunks. The algorithm proceeds in four phases. The first phase is an intra-node ReduceScatter that computes the sum of buffers within a node with the result "scattered" across the GPUs. In this example, this is done through a Ring algorithm. GPU 1 sends N chunks (chunk 0 and chunk 1) to GPU 2 which adds them to its corresponding chunks before sending them to GPU 0. In the end, GPU 0 has the intra-node sum of these N chunks, which is shown as lightly shaded in the figure. Other GPUs have intra-node sum of N other chunks each by executing a similar ring as shown in the figure.

The second phase is an inter-node ReduceScatter, where GPUs with the same intra-node index communicate to sum their chunks across nodes. For instance, GPU 0 (i.e., $(0, 0)$) and GPU 3 (i.e., $(1, 0)$) use a Ring algorithm to add the intra-node sums of chunk 0 and chunk 1. The result is scattered with each GPU having one chunk of the AllReduce result, which is shown as darkly shaded in the figure. The final two phases are an inter-node AllGather followed by an intra-node AllGather, both of which follow a similar Ring algorithm to distribute these chunks to all GPUs.

MSCCLang Program. The MSCCLang DSL is embedded in Python and allows users to write communication algorithms by declaratively specifying chunks routes across the GPUs to implement a collective. We call such specifications *chunk-oriented*. Figure 3 shows the code for the hierarchical AllReduce algorithm. When interpreted as a Python program, the execution mimics the description in Figure 1. Figure 3a creates the four phases: N and G instances of intra-node and inter-node ReduceScatter and AllGather, respectively. Figure 3b implements ReduceScatter and AllGather using the Ring algorithm. In MSCCLang, a `chunk` is identified by its rank and its index into a buffer in the rank, as shown in Line 8 and Line 17. As this chunk is routed across the ring, ReduceScatter performs `reduce` at Line 11 while AllGather performs a `copy` at Line 20. Section 3 explains the MSCCLang DSL in detail.

MSCCLang Architecture. Figure 2 describes the components of the MSCCLang framework. Given a MSCCLang program, the MSCCLang compiler lowers it into an intermediate representation called MSCCL-IR which is directly interpreted by MSCCLang's runtime. The compiler traces the program to capture the chunk dependencies in a Chunk DAG and performs several optimizations such as aggregation, instruction fusion, and parallelization. The compiler then schedules the program onto thread blocks using MSCCLang DSL directives so that the user may control the optimizations and scheduling choices. The compiler ensures that distributed execution correctly implements the chunk-oriented semantics of the input program with a guaranteed absence of deadlocks and data races.

The MSCCLang runtime executes MSCCL-IR as a single CUDA kernel and performs additional optimizations such as pipelining to improve thread block and link utilization. The key advantage of MSCCLang is that users get algorithmic flexibility to specify custom communication algorithms in a high-level DSL while still getting the performance of hand-written kernels.

3 MSCCLANG DSL

The MSCCLang DSL is a chunk-oriented language for specifying chunk routing through GPUs. The language is embedded in Python as a traced DSL with a fluent API that gives users flexibility in how to express algorithms. This section explains the core components of the MSCCLang DSL for implementing a collective algorithm by chunk routing. Furthermore, Section 5 discusses scheduling extensions that further optimize programs.

3.1 Buffers and Chunks

MSCCLang exposes GPU memory as named buffers, three of which are available on each rank:

- *Input* is a buffer containing input data.
- *Output* is an uninitialized buffer for storing output data.
- *Scratch* is an uninitialized buffer used for temporary storage.

Buffers divide into *chunks* representing contiguous spans of elements with a uniform size. The user controls the number of chunks a buffer divides into, but the size (i.e., number of bytes) is abstract. The size is known at runtime when the concrete buffers are passed into the program. Additionally, users can specify that the input and output buffers are aliased to support in-place collective algorithms. The purpose of the MSCCLang program is to ensure that the output buffer on each GPU has the correct chunks for the collective at the end. Chunks can take three forms:

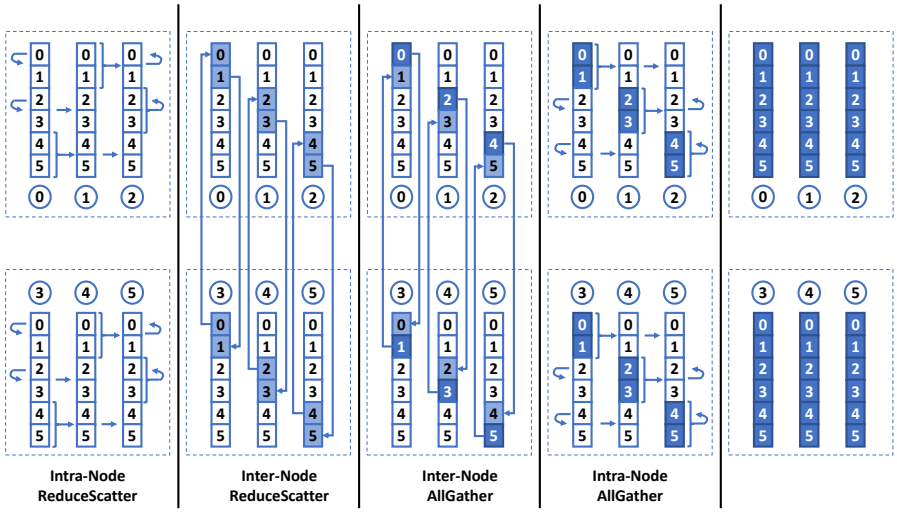


Figure 1: Hierarchical AllReduce on 2 nodes each with 3 local GPUs.

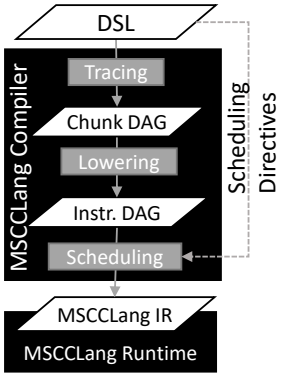


Figure 2: Architecture of MSCCLang

```python
# N: number of nodes
# G: number of GPUs per node
# N*G: number of chunks
def HierarchicalAllReduce(N, G):
  # intra-node ReduceScatter
  for n in range(N):
    local_ranks = [i+n*G for i in range(G)]
    ReduceScatter(local_ranks, 0, N)

  # inter-node ReduceScatter + AllGather
  for g in range(G):
    cross_ranks = [i*G+g for i in range(N)]
    ReduceScatter(cross_ranks, g*N, 1)
    AllGather(cross_ranks, g*N, 1)

  # intra-node AllGather
  for n in range(N):
    local_ranks = [i+n*G for i in range(G)]
    AllGather(local_ranks, 0, N)
```

(a) MSCCLang program for hierarchical AllReduce.

```python
# ranks = list of ranks involved
# offset = offset into the input buffer
# count = num. of chunks sent at each step
def ReduceScatter(ranks, offset, count):
  R = len(ranks)
  for r in range(0, R):
    index = offset + r * count
    c = chunk(ranks[(r+1)%R], 'in', index, count)
    for step in range(1, R):
      next = ranks[(step+r+1)%R]
      c = chunk(next, 'in', index, count).reduce(c)

def AllGather(ranks, offset, count):
  R = len(ranks)
  for r in range(0, R):
    index = offset + r * count
    c = chunk(ranks[r], 'in', index, count)
    for step in range(1, R):
      next = ranks[(step+r)%R]
      c = c.copy(next, 'in', index, count)
```

(b) MSCCLang helper functions for Ring ReduceScatter and AllGather.

Figure 3: MSCCLang programs for hierarchical AllReduce.

- *Input chunks* represent chunks initialized at runtime. The pair (rank, index) uniquely identifies the input chunk in the input buffer.
- *Reduction chunks* result from combining two chunks through a point-wise reduction (e.g., addition). The list of input chunks that combine to form a reduction chunk uniquely identifies it.
- *Uninitialized chunks* are a unit type that stores uninitialized data. At the start of the program, the output and scratch buffers hold uninitialized chunks.

3.2 Collectives

MSCCLang programs are associated with a *collective* that defines a *precondition* and a *postcondition*. The precondition determines the starting state of the input buffer in terms of unique input chunks. The postcondition sets the desired state of the output buffer — for each index of the output buffer, the postcondition specifies either an input chunk or a reduction chunk to correctly implement the collective. Defining a collective's postcondition allows MSCCLang to automatically validate that a prospective algorithm is correct.

For example, the precondition of an AllReduce over R ranks states that each rank's input buffer contains C unique input chunks: $(c_0^r, c_1^r, ..., c_C^r)$. The postcondition states that all rank's output buffer are identical and contain the C reduction chunks: $(\sum_{r=0}^{R} c_0^r, \sum_{r=0}^{R} c_1^r, ..., \sum_{r=0}^{R} c_C^r)$.

The algorithm, not the collective, determines the number of chunks and whether the algorithm the input and output buffer alias each other (i.e. the algorithm is in-place). For example, the hierarchical AllReduce algorithm (Figure 3) is an in-place algorithm that uses $N \times G$ chunks. For programmability, MSCCLang automatically deduces the number of chunks in the scratch buffer based on the highest scratch indices accessed in the program.

3.3 MSCCLang Operations

Table 1 lists the MSCCLang operations used for manipulating chunks. The function `chunk(rank, buffer, index, count=C)`

Table 1: MSCCLang DSL operations.

Operation	Description
`chunk(rank, buffer, index, count=1) → c`	Returns a reference (rank, buffer, index, count) for the chunks currently in the buffer.
`c1.copy(rank, buffer, index) → c2`	Copies chunks referenced by c1 into the destination indices. Returns a new reference c2 for the copied chunks.
`c1.reduce(c2) → c3`	Reduces chunks referenced by c1 and c2 *in-place* into the indices of c1. Returns a new reference c3 for the result.

returns a reference to `C` contiguous chunks currently assigned to the named `buffer` starting at `index`; `count` has a default value of one when it is not set. MSCCLang raises an error if the program accesses an uninitialized chunk.

The `copy` and `reduce` operations move chunks between buffers. Specifically, `c1.copy(rank2, buffer, index2)` copies the chunks referenced by `c1` to `(rank2, buffer2, index2)`. `c1.reduce(c2)` reduces two equal count chunk references `c1` and `c2`. This is an in-place pointwise operation that overwrites `c1` with the reduced chunks. Both the `copy` and `reduce` operations return references to the newly created chunks, which allows fluently chaining `copy` and `reduce` calls.

Programs manipulate references rather than chunks to prevent operations on stale data. A program can create multiple references to the same (rank, buffer, index) location potentially referring to stale chunks that are overwritten by later operations. MSCCLang only allows the latest reference for any location to be used and will generate an error otherwise. This enforces a chunk-oriented coding style with the program always operating on the latest reference, thereby making MSCCLang programs data race free by construction.

MSCCLang lets users express operations between buffers uniformly with `copy` and `reduce` regardless of whether they are on the same GPU or not. The next section explores how MSCCLang enables this abstraction.

4 LOWERING MSCCLANG PROGRAMS

This section explains how MSCCLang's compiler lowers programs into instructions by first *tracing* them into a directed acyclic graph (DAG) of operations, which we call the Chunk DAG, and then further lowering into an Instruction DAG. Section 5 discusses how the compiler schedules these instructions into code targeting the low-level MSCCL-IR for our runtime, as well as the optimization interface the DSL provides for controlling scheduling decisions.

4.1 Tracing

The compiler traces a program by sequential execution into a Chunk DAG, which captures the global view of chunk movement and naturally exposes the program's parallelism. The graph includes source nodes for all input chunks. Every `copy` and `reduce` operation is also a node, and the edges between nodes are dependencies between operations that arise from chunk movement (true dependencies) and reusing buffer indices (false dependencies).

Figure 4 depicts a subset of the Chunk DAG of Figure 3 that traces chunk 0 across every rank. The Chunk DAG preserves the hierarchical structure of the program, with the first two levels of reduces corresponding to the intra-node ReduceScatter and the the last reduce corresponding to the inter-node ReduceScatter.

4.2 Instruction Generation

The compiler expands each chunk operation node into instruction nodes to generate the Instruction DAG. Instructions are either point-to-point communication primitives or local primitives that are executed by a single GPU. The instructions are listed below:

`send(buffer,index)/recv(buffer,index)` sends/receives from the given buffer at the chunk index to/from the remote GPU.

`reduce(srcBuf,srcInd,dstBuf,dstInd)` locally applies a predefined reduction operation to the corresponding chunks and stores the result in the destination.

`copy(srcBuf,srcInd,dstBuf,dstInd)` performs a local copy of a chunk from a source location to a destination.

`recvReduceCopy(srcBuf,srcInd,dstBuf,dstInd)` is a fused instruction that receives a chunk, reduces it with a source chunk, and locally copies it to the destination. Abbreviated as **`rrc`**.

`recvReduceCopySend`, **`recvReduceSend`**, **`recvCopySend`** are additional fused instructions that performs receive, send, and an optional reduction of a chunk. Abbreviated as **`rrcs`**/**`rrs`**/**`rcs`**.

The fused instructions can be implemented by composing `send`, `recv`, `reduce`, and `copy` instructions. However, fused implementations can optimize away global memory accesses as intermediate values are transferred through GPU registers.

The compiler expands chunk operations differently depending on whether they are local or remote. A remote `copy` expands into a `send` and a `receive` instruction, and a remote `reduce` expands into a `send` and a `receiveReduceCopy` instruction. For local `copy` or `reduction` operations MSCCLang generates only a single local instruction. Note that instructions such as `receiveCopySend` cannot be generated this way as it requires looking at two chunk operations.

The compiler connects the two instructions resulting from a remote operation by a communication edge that indicates that the receiving side synchronizes with the sender. It also preserves the original edges of the Chunk DAG as processing edges, which represent the execution-order dependencies within ranks.

4.3 Instruction Fusion

The initial instruction generation pass only uses a subset of the available instructions and excludes the fused instructions that combine a receive and a send. The compiler performs a series of peephole

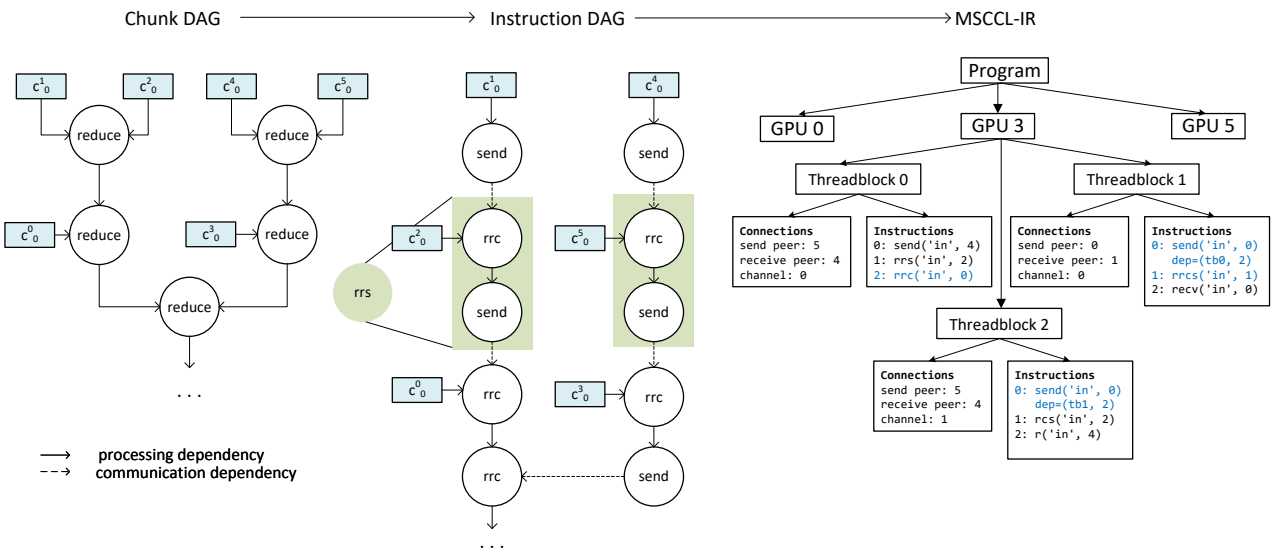


Figure 4: MSCCLang compilation of the hierarchical AllReduce. A subset of the program for chunk 0 is traced into a Chunk DAG of operations, lowered into an Instruction DAG of instructions, and scheduled onto thread blocks.

optimizations to combine consecutive base instructions into fused instructions.

`rcs`. Rewrites a back-to-back receive and send on the same chunk into a fused `receiveCopySend`: If there are multiple sends dependent on the receive, the send on the longest path in the Instruction DAG is fused.

`rrcs`. Rewrites a back-to-back `receiveReduceCopy` and a `send` on the same chunk into a `receiveReduceCopySend`.

`rrs`. Is a special case of the previous optimization; if the result of the `rrc` is never used locally (i.e. it is later overwritten), the reduction result does not need to be saved locally and a more efficient `receiveReduceSend` instruction is used instead.

Figure 4 depicts the hierarchical AllReduce Instruction DAG for chunk 0 up to the inter-node ReduceScatter. The compiler has expanded each operation node into two instruction nodes, with communication edges connecting a matching send and receive. Highlighted in green, is a back-to-back `rrc` and `send` that is fused into a `rrs` instruction.

5 SCHEDULING MSCCLANG PROGRAMS

After a program is lowered into an Instruction DAG, it is scheduled into a MSCCL-IR program that specifies the program's execution. At a high-level this process assigns every instruction to a thread block that will execute it and every communication edge to a channel identifying the connection data is transferred through.

MSCCL-IR. MSCCL-IR is a tree data structure (Figure 4) that divides a collective into individual GPU programs that are subdivided into thread blocks containing a list of instructions that are sequentially executed.

Thread blocks can make two uni-directional connections to a *send peer* and a *receive peer* that are used for sending and receiving chunks. GPUs are allowed to have multiple redundant connections to the same GPU that are identified by a *channel*. Channels are similar to tags in MPI that identify the route a send takes.

Our design restricts thread blocks to have at most one send and receive connections so that two thread blocks do not serialize over the same connection. Similarly, a connection can only have one sending and receiving thread block. The compiler ensures this constraint is honored by during scheduling.

5.1 DSL Scheduling Directives

Optimizing a program's schedule is crucial for extracting performance. The MSCCLang DSL provides a set of *scheduling directives* so that users can optionally specify optimizations on top of a program. These optimizations trade off between parallelization, by assigning instructions across multiple thread blocks and occupancy constraints, as each thread block consumes GPU resources.

Channel Directives. A program may use multiple connections between the same pair of GPUs that are differentiated by their channels. Programs can specify that an operation utilizes a particular channel with an optional parameter `ch`. The code below schedules two copies between the same pair of GPUs on different channels which ensures they can execute in parallel:

```
c.copy(rank, buf, idx, ch=0)
c.copy(rank, buf, idx, ch=1)
```

In the hierarchical AllReduce, we manually scheduled intra-node ReduceScatters onto channel 0, the inter-node AllGathers and ReduceScatters onto channel 1, and the intra-node AllGather onto channel 2.

Chunk Parallelization. An important performance aspect is the amount of parallelism used for transfers. Chunk parallelization is an automatic optimization that breaks up a transfer into multiple smaller transfers that execute in parallel. In the hierarchical AllReduce, we parallelize the intra-node ReduceScatters and AllGathers using the `parallelize` modifier:

```
for n in range(N):
    local_ranks = [i+n*G for i in range(G)]
    with parallelize(N):
        ReduceScatter(local_ranks, 0, N)
    . . .
```

Parallelizing a code fragment by N has the effect of creating N parallel instances of the underlying copy and reduce operations, where each operation operates on $1/N$ of the data. The compiler duplicates instruction nodes corresponding to the fragment and ensures each instances's channels do not intersect so that instances execute in parallel.

There are two advantages of chunk parallelization. First, this enables parallelization of compute heavy aspects of the algorithm such as reductions. Second, parallelization can increase the utilization of high-bandwidth links by allowing multiple thread blocks to simultaneously use the underlying link. Our experience has shown that a single thread block in an NVIDIA A100 GPU is not capable of saturating the bandwidth of its outgoing NVLink. The user should carefully choose the parallelization factor as increasing it beyond a certain point will reduce performance due to competition for bandwidth.

Aggregation. When multiple chunks are transferred from one GPU to another and these chunks are contiguous, it is sometimes more efficient to aggregate these chunks in a single network transfer. Assuming an $\alpha - \beta$ communication cost model, each send has a start up α cost and a per-byte β cost. By aggregating sends, the compiler amortizes the start up cost. However, aggressive aggregation can slow down a program. All aggregated chunks must be ready before the send is executed, so that one delayed chunk blocks the progress of multiple chunks.

Users specify aggregated sends by passing multi-count chunk references to `copy` and `reduce` operations. For example, Line 8 and Line 17 indicate that N chunks are should be aggregated into a single send during the intra-node ReduceScatters and AllGathers.

5.2 Scheduling

Given a program's Instruction DAG and scheduling directives, the compiler assigns every instruction node to a thread block and remaining edges onto channels. This assignment respects the constraints that each thread block can have at most one send and receive peer. Additionally for correctness, the assignment does not introduce deadlocks, which are possible due to the sequential execution order of instructions within a thread block.

Channel Assignment. The compiler assigns every communication edge according to the user's scheduling directives. Any remaining communication edges are assigned to the lowest valid channel with two exceptions. First, communication edges generated from a parallelized code fragment are scheduled onto different channels so that they don't serialize. Second, a series of fused instructions share the same channel. The compiler ensures this by assigning the lowest channel that satisfies all communication edges in the chain.

Thread Block Assignment. The compiler's thread block assignment policy implements a greedy heuristic that attempts to schedule instructions in the order they will be ready. The high level steps of the routine are as follows:

(1) *Assign instruction priority* by calculating the depth (maximum hops from a root node) and reverse depth (maximum hops to a leaf node) for every instruction in the Instruction DAG. This prioritizes instructions that are enabled earlier and have more downstream dependencies respectively.
(2) *Create thread blocks* by scanning through all instructions per GPU and creating thread blocks for every unique (send-peer, receive-peer, channel) tuple.
(3) *Sort instructions* into a global topological order with respect to their dependencies with a heap using the priority to order instructions.
(4) *Assign instructions to thread blocks* to their matching matching thread block in the topological order. If an instruction has multiple candidates (e.g. local copies can happen on any thread block) then the thread block whose latest assigned instruction is earliest is chosen.

An Instruction DAG is guaranteed to have a *global* topological order because it was generated by sequentially tracing a MSCCLang program. By assigning instructions to thread blocks in a topological order that respects communication and processing edges, all implicit dependencies introduced by thread block sequential execution cannot produce cycles so that the MSCCL-IR does not have deadlocks.

Cross Thread block Synchronization Insertion. An instruction's dependencies are captured in the Instruction DAG as communication edges between sends and receives, and processing edges that indicate execution order within a rank. The MSCCL-IR program must respect this order to be correct and data race free. While communication edges implicitly synchronize because receives block until a send, certain processing edges need explicit synchronization.

Instructions within a thread block are executed sequentially, and thus any processing edges between them are already satisfied. However, instruction across thread blocks execute out-of-order. Processing edges between different thread blocks are explicitly preserved in the MSCCL-IR file as cross thread block dependencies. Instructions with cross thread block dependencies have a `dep` modifier which identifies the instruction(s) that must execute before.

6 MSCCLANG RUNTIME

The MSCCLang runtime executes program by directly interpreting MSCCL-IR programs. The runtime is an extension of NCCL, and it inherits infrastructure for establishing point-to-point (P2P) connections over various inter-connects including NVLink, PCIe, shared host memory, InfiniBand (IB) and TCP. All MSCCL-IR generated by our compiler is guaranteed to be correct, but some programs might only be performant for a range of buffer sizes. Therefore, the runtime dynamically selects the right algorithm to invoke based on user configurable size ranges and falls back to NCCL's built-in algorithms otherwise. This allows a user to hyper-optimize MSCCLang programs to a specific use case.

```
1 struct Instruction {
2   int step, opCode, srcOff, dstOff, count;
3   void *srcPtr, *dstPtr;
4   int depBid[D], depStep[D];
5   bool hasDep; };
6
7 MSCCLang_interpreter(Instructions instrs[N]) {
8   int bid=blockIdx.x; int tid=threadIdx.x;
9   // chunk tiling
10  for (int t=0; t < chunkSize; t += tileSize){
11    // instruction loop
12    for (int s=0; s<N; s++) {
13      auto instr = instrs[s];
14      // check for dependencies
15      if (tid < D)
16        wait(semaphore[instr.depBid[tid]], instr.depStep[tid]);
17      // select the instruction
18      switch (instr.opCode) {
19        case SEND:
20          send(instr.srcPtr+instr.srcOff, instr.count*tileSize);
21        case RECV:
22          ...
23      }
24      // set the semaphore if necessary
25      if (instr.hasDep) {
26        thread_fence(); sync_threads();
27        if (tid==0) set(semaphore[bid], s);
28      }}}}
```

Figure 5: MSCCLang interpreter

6.1 Point-to-Point Connections

Remote Buffers. NCCL abstracts different kinds of interconnects from CUDA code by providing intermediate buffers of constant size of b bytes for sends to write to and receives to read from. These buffers are subdivided into s FIFO *slots* which allows s sends to finish without waiting for receives ($1 \leq s \leq 8$). MSCCLang compiler prevents a schedule with more than s outstanding sends to avoid deadlocks. By default, 512KB $\leq b \leq$ 5MB and $1 \leq s \leq 8$ (exact values are defined by the protocol, explained later).

Remote buffers are allocated on different memories depending on the inter-connection type. For NVLink or PCIe connections within a node, buffers are allocated on the receiving GPU. For cross-node IB connections, two buffers are allocated with one on the sending GPU and another on the receiving GPU. The IB driver transfers data between the buffers via GPUDirect RDMA [26], with a CPU helper thread initiating RDMA transfers. Other types of interconnects involve host memory, but we omit their description as they are not used on our evaluation systems.

Channels. As explained in Section 5, each P2P connection in NCCL requires a *channel*, which is an internal NCCL data structure that distinguishes different P2P connections between the same pair of GPUs.

Protocols. NCCL implements three communication protocols, Simple, LL128, LL, that trade off latency and bandwidth. Simple has the highest bandwidth and latency, LL has the lowest bandwidth and latency, and LL128's performance is in-between [27]. The protocol also defines the remote buffer size and the number of slots. The user may set a desired protocol in the DSL, which is stored in the MSCCL-IR.

6.2 Interpreter

Initialization. In the initialization phase of the runtime, an MSCCL-IR program is parsed and stored in the GPU memory. When the runtime invokes the interpreter for a given program, it concurrently launches all the required thread blocks with a cooperative kernel launch [8]. Note that all thread blocks must execute at the same time due to potential cross thread block dependencies between them. Consequently, the compiler can only generate IRs that do not have more thread blocks than the available Streaming Multiprocessors (SMs). The connections needed by the thread blocks in every program (Figure 4) are also created.

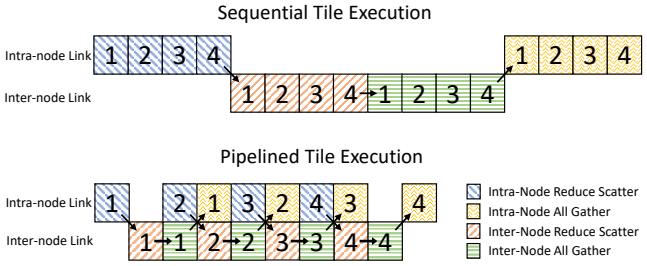


Figure 6: Order tiles occupy intra-node and inter-node links in the hierarchical AllReduce. The top shows a naive sequential execution while the bottom shows the pipelined execution MSCCLang implements.

Instruction Data Structure. The execution engine for MSCCLang runtime is an efficient interpreter written in CUDA shown in Figure 5 which runs a list of instructions on each thread block. Line 1 shows the elements of an instruction: `step` is the instruction index in an array, `opcode` identifies the instruction type, `srcPtr` and `dstPtr` are the input and output pointers, and `srcOff` and `dstOff` are their corresponding offset, respectively. The pointers can be one of input, output, or scratch buffers, and offset is the chunk index into the buffer. `count` is the number of consecutive chunks this instruction will execute on (see aggregation in Section 2). Last arguments are for cross thread block synchronizations: `depBid` and `depStep`. These two arrays are a list of thread block IDs and instruction steps, respectively, that this instruction is dependent on. `hasDep` is a boolean flag indicating whether there are other instruction dependent on this instruction.

Pipelining. The outer-most loop in the interpreter is the pipelining loop shown in Line 10 of Figure 5. As described in Section 6.1, the remote buffers for each P2P connection have a fixed size. Therefore, if the size of a chunk is larger than a remote buffer slot, it is split into multiple *tiles* such that it fits in a slot.

Rather than serially process each tile within a chunk, the interpreter pipelines execution for performance. Consider the hierarchical AllReduce in Figure 1. It starts with an intra-node ReduceScatter followed by inter-node ReduceScatter and AllGather, and ends with an intra-node AllGather. If the interpreter serially executes each chunk's tile, the inter-node communication links are not utilized during intra-node phases and vice versa (Figure 6). Instead, the interpreter pipelines execution of the tiles by processing tile 1, then processing tile 2, etc., so that both the inter-node and intra-node links are utilized concurrently.

Pipelining improves performance by increasing link and SM utilization in the system. Users may configure MSCCLang's tile size for more aggressive pipelining. However, as tile sizes reduce, the

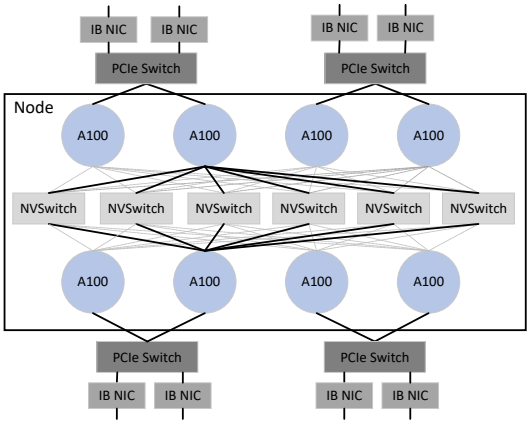


Figure 7: Network topology of an NDv4 node. 8 A100 GPUs are fully connected to each other through NVLinks to 6 NVSwitches (NVLinks shown for two GPUs). Every pair of GPUs shares a PCIe switch to 2 InfiniBand NICs for communication outside the node.

performance benefit of pipelining decreases due to the increased startup cost of executing more sends.

Instruction Loop. The inner-most loop in the interpreter in Line 12 decodes instructions in the input MSCCL-IR and executes them in-order. There is a list of switch-case statements in Line 18 that decides which instructions to execute.

Cross Thread Block Synchronization. Cross thread block synchronization is not naturally supported in CUDA. However, the interpreter runs all thread blocks concurrently, which allows thread blocks to synchronize via semaphores stored in global memory. Each thread block has a semaphore (`semaphore[bid]`) in Figure 5 that is initialized to 0. When an instruction `hasDep` is set (Line 25), a CUDA `__syncthreads` and a `__threadfence` is issued to flush the caches and then the semaphore is set to the running step `s` (Line 27). If this instruction is dependent on instructions from other thread blocks, all semaphores for dependent thread blocks wait to be set (Line 16).

7 EVALUATION

We evaluate MSCCLang by implementing classic and custom algorithms for the commonly used collectives AllReduce and AllToAll. We optimize each algorithm's schedule for various GPU system configurations and input sizes. All our programs require less than 30 lines of code, and took between 15 minutes to an hour to write and manually optimize.

Experimental Setup. Experiments are performed on two GPU clusters: Azure ND A100 v4-series (NDv4) and NVIDIA DGX2s. Each NDv4 (Figure 7) contains 8 NVIDIA A100 GPUs connected by 12 third-generation NVLinks to 6 NVSwitches for a total of 600 GB/s bi-directional bandwidth. For cross-node communication, each pair of GPUs within a node share a single PCIeSwitch that connects to 2 HDR InfiniBand NICs, each running at 25 GB/s bandwidth.

Each DGX2 node contains 16 NVIDIA V100s divided into two boards of 8 GPUs. GPUs on each board are connected by 6 second-generation NVLinks to 6 NVSwitches, and every NVSwitch is connected by 8 NVLinks to its counterpart NVSwitch on the other board. For cross-node communication each pair of GPUs share a single PCIeSwitch that is connected to 1 HDR InfiniBand NIC running with 25 GB/s bandwidth.

MSCCLang is built on top of NCCL-2.8.4-1 [27]. When applicable, we compare against collectives implemented in NCCL or expert hand-optimized implementations. For custom algorithms and collectives for which hand-optimized CUDA implementations were previously not available, we provide best effort hand-written kernels. All experiments are averaged over 50 iterations after a warmup period of 20 iterations. Each algorithm's optimizations are tuned for the two systems. We analyze results for the A100 system since similar trends are seen on the V100 system and discuss certain exceptions.

7.1 AllReduce

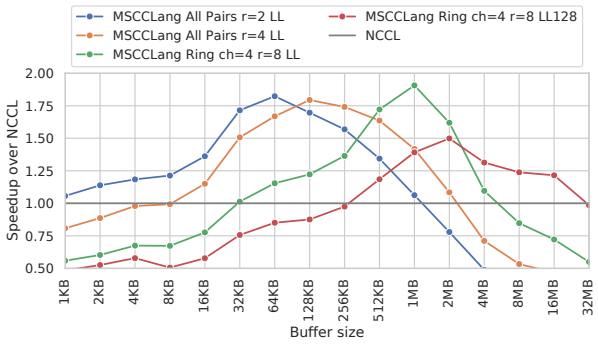
AllReduce is an MPI collective that globally reduces input buffers across GPUs and replicates the results to all GPUs. We implemented three AllReduce algorithms that target single-node and multi-mode systems.

7.1.1 Ring AllReduce. A Ring AllReduce with `R` ranks, divides each rank's input buffer into `R` chunks. Ranks are logically connected in a ring, and each chunk traverses the ring twice starting from the corresponding rank. The first traversal reduces all corresponding chunks and the second traversal copies the result to all ranks. We implement our ring with a ReduceScatter followed by an AllGather from Figure 3b using a list of all ranks in the node, an offset of 0, and a count of 1 for the parameters.

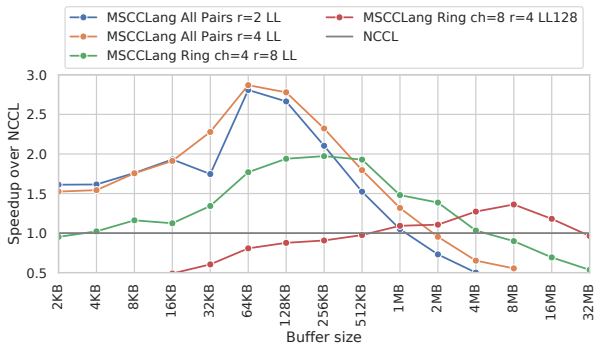
Our ring implementation distributes a single logical ring across multiple channels by varying the channel of copy and reduce operations. We tune the number of channels per ring, parallelization, and protocol for the system. While examining NCCL's codebase, we found and experimentally validated that NCCL's Ring schedule is roughly equivalent to scheduling a logical ring onto one channel, parallelizing the entire program 24 times, and varying the protocol based on the buffer size.

We compare our Ring implementations against NCCL's Ring implementation in Figure 8a. The MSCCLang Ring implementation outperforms NCCL by up to 1.9× when the buffer size is between 32KB and 3MB. Distributing a logical ring across multiple channels enables better overlapping of sends and receives resulting in performance gains. However, this distribution uses more resources thus limiting the chunk parallelization. For buffer sizes greater than 32MB, more parallelization is required, and the best MSCCLang configurations matched NCCL's performance by scheduling a logical ring onto one channel and parallelizing the program 24 times.

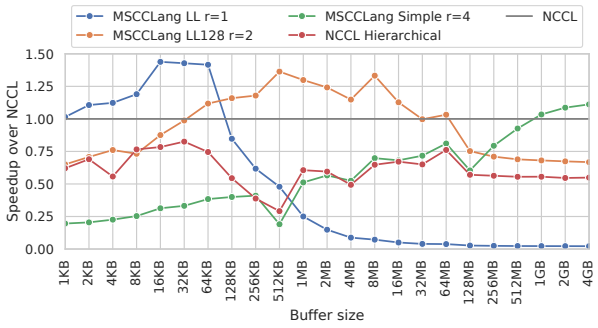
7.1.2 All Pairs AllReduce. One advantage of MSCCLang is the ability to explore different algorithms easily. All Pairs is an algorithm we developed while exploring algorithmic optimizations for AllReduce that targets small buffer sizes. This algorithm uses two communication steps: each rank gathers a chunk from every rank, computes the sum, and broadcasts the chunk to every other rank.



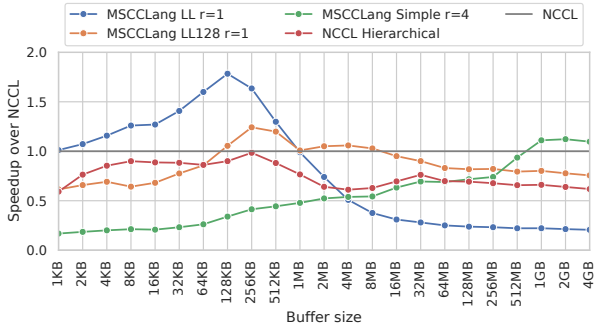
(a) 1-node, 8×A100 AllReduce.



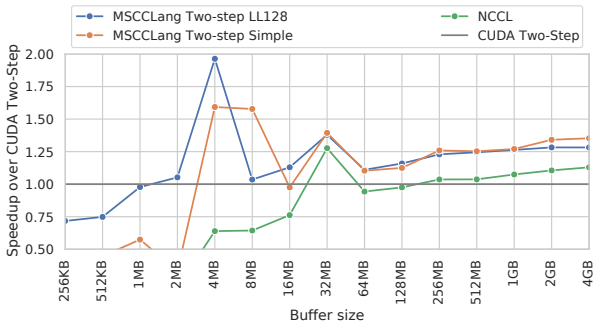
(b) 1-node, 16×V100 AllReduce.



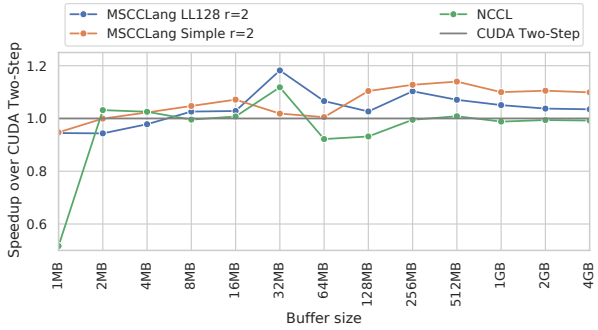
(c) 2-node, 16×A100 AllReduce



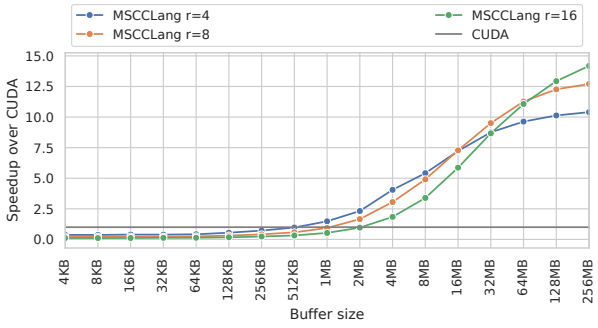
(d) 2-node, 32×V100 AllReduce



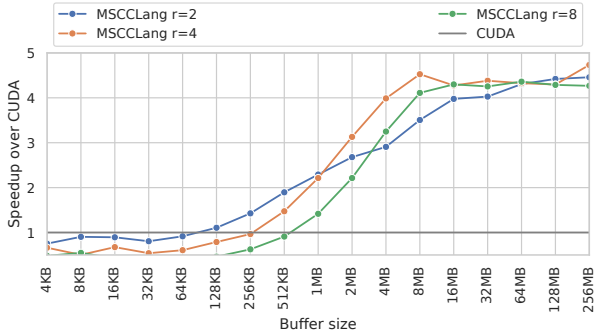
(e) 16-node, 256×A100s AllToAll



(f) 4-node, 64×V100 AllToAll



(g) 3-node, 24×A100 AllToNext



(h) 4-node, 64×V100 AllToNext

Figure 8: Speedup of MSCCLang collective algorithms. Plots on the right are for A100 nodes and plots on the left for DGX2 nodes. The parameter, *r*, specifies the parallelization factor of the whole program.

```
# N: number of nodes
# G: number of GPUs per node
def alltoall(N, G):
  for n in range(N):
    for g in range(G):
      for m in range(N):
        for i in range(G):
          c = chunk((m,i), 'in', (n,g))
          if n == m:
            c.copy((n,g), 'out', (m,i))
          else:
            c.copy((m,g), 'sc', (n,i))

      # Coalesced IB send
      c = chunk((m,g), 'sc', n*G, sz=G)
      c.copy((n,g), 'out', m*G)
```

Figure 9: MSCCLang Two-Step AllToAll.

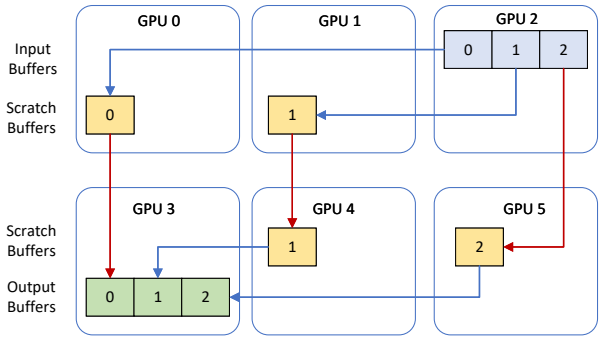


Figure 10: AllToNext cross node send on a system with $(N = 2, G = 3)$ **GPUs. Sends over NVLink are shown in blue and sends over IB are shown in red.**

Since we do not have a CUDA baseline to compare against, we plot the speedup of MSCCLang's All Pairs against NCCL's Ring algorithm in Figure 8a.

The speedups provided by All Pairs are driven by algorithmic and scheduling optimizations. Ring and All Pairs exchange the same volume of data, but All Pairs has better latency because it uses 2 communication steps compared with Ring's $2R - 2$ steps. For buffer sizes from 1KB to 1MB, All Pairs is up to 1.8× faster than NCCL, depending on the number of instances used to optimize the program.

7.2 Hierarchical AllReduce

The final AllReduce algorithm we analyze is a Hierarchical AllReduce algorithm described in Section 2. Figure 8c plots the speedup of the Hierarchical AllReduce implemented in MSCCLang against NCCL. Depending on the input size, we apply different optimizations to the same base algorithm. For small sizes we are up to 1.4× faster than NCCL. For large buffers, greater than 1GB, our implementation is up to 11% than NCCL.

In red, we plot the speedup of same algorithm implemented with NCCL collectives. The implementation is significantly slower than the MSCCLang's and NCCL's implementation due to the overhead of multiple kernel launches and lack of cross-kernel optimizations. Using MSCCLang optimizations to execute the algorithm in a single kernel and pipeline thread blocks significantly improved the algorithm's performance such that it is faster than NCCL's implementation for a large range of buffer sizes.

7.3 Two-Step AllToAll

AllToAll is an MPI collective that transposes a buffer of data between GPUs such that chunk i on GPU j ends up on GPU i at index j, and is commonly implemented as a set of point-to-point sends and receives between all GPUs. Because each GPU exchanges data with every other GPU, AllToAll is a very communication intensive collective.

On AllToAlls spanning 10s to 100s of nodes, the naive implementation requires only one communication step, but sends many small chunks to other nodes over IB which is expensive due to the high overhead costs of IB. We implement a Two-Step AllToAll algorithm that aggregates cross-node sends, reducing the total overhead cost of the IB sends. The algorithm is described in Figure 9, and we used MSCCLang's default scheduling with 1 instance and tuned the protocol for the buffer size.

We compare the performance of our implementation against a hand-optimized CUDA implementation of the Two-Step algorithm in Figure 8e. At large sizes the MSCCLang implementation is up to 1.3× faster than the hand-optimized implementation. Note, at smaller sizes between 2MB-64MB there are large fluctuations in speedup caused by congestion in the IB network which is shared with other cloud tenants; however the general trends show that MSCCLang's optimizations improve performance.

For reference, we also plot NCCL's performance relative to the hand-optimized implementation. In general, both Two-Step implementations provide significant improvements over NCCL. However, for larger buffer sizes, greater than 512MB the hand-optimized Two-Step implementation is slower than NCCL, while the MSCCLang implementation is 20% faster.

The hand-optimized version is implemented using point-to-point primitives exposed by NCCL, but lacks scheduling decisions made by the compiler that divides communication across multiple parallel thread blocks. The MSCCLang seamless handles aggregating chunks in the scratch buffer (Line 12), while the handwritten implementation requires a separate kernel that copies and contiguously arranges chunks in a scratch buffer for the aggregated IB send resulting in extra synchronization overhead. Furthermore, the MSCCLang implementation is much more succinct and requires only 15 lines of code while the hand optimized kernel requires roughly 70 lines of code.

7.4 Custom Collectives: AllToNext

A key feature of MSCCLang is the ability to implement *new* collective communication patterns quickly and efficiently. We demonstrate this on a new collective called AllToNext. This collective involves R GPUs, where GPU i sends a buffer of data to GPU $i + 1$, with the last GPU sending nothing. This communication pattern exists in applications that process data in a pipelined fashion across multiple GPUs. In a naive implementation of AllToNext, every GPU directly sends its buffer to the next GPU. However, on a distributed system made up of multiple nodes with heterogeneous links, throughput is bottlenecked by the low-bandwidth inter-node network links. Furthermore, on a nodes with multiple network

links this will only use one link, wasting inter-node bandwidth. For example, in an A100 node, there are 8 IB links but a single send can only utilize 1 link.

We designed the AllToNext algorithm specifically to address this problem by utilizing all available IB links in a node. Within a node, GPUs directly send their buffers to the next GPU; when transferring across a node (Figure 10), all GPUs within the nodes cooperatively send the buffer to utilize all IB links. Specifically, on a scaled down A100 system ($N = 2, G = 3$), when GPU $(0, 2)$ sends to GPU $(1, 0)$, it will divide its buffer into $G = 3$ chunks and scatter it among all GPUs in node 0. Each GPU $(0, g)$ directly sends its chunk to the corresponding GPU on the next node $(1, g)$, and finally all chunks will be gathered back onto GPU $(1, 0)$.

Figure 8g plots the speedup of AllToNext on 3×A100 nodes against a handwritten CUDA baseline where each GPU directly sends its entire buffer to the next GPU using NCCL's send and receive primitives. When sending small buffers, AllToNext performs worse than baseline due to overhead from the extra communication steps. For larger buffers however, AllToNext begins to show improvement over the baseline, and is ultimately up to 14.5× for a large buffers. The best performing selection of r depends buffer sizes. For small buffer sizes, less parallelization provide better performance, as the benefit from parallelizing communication doesn't offset the cost of initializing extra resources. As the buffer sizes increase, programs with more parallelization produce larger speedups as the initialization overhead is amortized over more communication.

7.5 SCCL Comparison

SCCL [4] is an automatic collective communication algorithm generator which similarly considers both latency and bandwidth of each link for optimal solution. SCCL implements these algorithms from scratch using its own point-to-point communication protocol that works for GPUs interconnected with NVLinks only. The focus of SCCL is mostly on generating custom algorithms while the focus of MSCCLang is on implementing any custom algorithm for any interconnection such as InfiniBand, shared host memory, or NVLinks.

Figure 11 compares the performance of (1,2,2) AllGather algorithm from SCCL on DGX-1 8×V100 GPUs (see Table 4 from Section 5.4 in [4]) using SCCL and MSCCLang implementations. It is clear that MSCCLang implementation is faster for small sizes thanks to LL protocol, but Simple protocol is not as performant as SCCL protocol for middle sizes. The reason is that SCCL implementation uses a direct copy from source to destination for point-to-point communication while MSCCLang protocols use FIFO slots for intermediate buffers as explained in Section 6.1. This means that SCCL protocol has less memory footprint than MSCCLang Simple protocol and therefore, it is more efficient. SCCL direct copy protocol can also be implemented in MSCCLang Simple protocols, but we leave it for future work.

7.6 End-to-End Results

Custom algorithms generated by MSCCLang are used by Azure OpenAI to accelerate Copilot. These algorithms speed up the GPU time by 20%. MSCCLang is also used for training a large Mixture-of-Experts model [28] for speech, language, and vision on 256×A100 GPU providing 1.10-1.89× speed up depending on the model architecture when using the Azure Container for PyTorch [32].

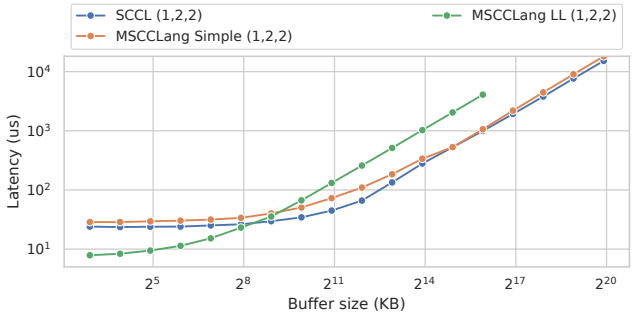


Figure 11: (1,2,2) AllGather algorithm from SCCL on DGX-1 8×V100 GPUs.

8 RELATED WORK

Optimizing Collectives. The message passing interface (MPI) [11] is a popular abstraction for communication primitives. Efficient algorithms for implementing these primitives is a long-studied research area [5, 31, 41]. Prior works have included optimizing algorithms for specific topologies like mesh, hypercube, or fat-tree [2, 3, 37] and for clusters of shared-memory processors [34, 40, 42, 43]. Motivated by recent ML workloads, Horovod [38] implements collective primitives by using NCCL locally in node and MPI across nodes. Others such as BlueConnect [7] and PLink [24] exploit the hierarchical network topology of a cloud system or a data center to improve the performance of collective primitives. Recent work focuses on automatically generating new collective algorithms, either by packing trees [44] or using a constraint solver to generate pareto-optimal algorithms [4]. In contrast, this work focuses on a high-level language for specifying these algorithms and efficiently running them on state-of-the-art accelerators.

In-network aggregation is another direction to accelerate reduction based communication primitives using custom hardware. Mellanox Scalable Hierarchical Aggregation and Reduction Protocol (SHArP) [13] is one of the techniques available in InfiniBand switches. Other programmable switches including SwitchML [36] and ATP [22] also share the similar idea to offload GPU reduction to network switches in order to accelerate AllReduce in deep learning workloads. Apart from switches, BluesMPI [16], ACCL [18] and BytePS [21] also offload communication primitives to SmartNIC, FPGA, and spare CPU nodes, respectively. Those works all introduce extra hardware thus increase bandwidth limit for primitives, while MSCCLang focuses on software stack only to program and optimize collective communication algorithms within existing hardware.

Recent works [15, 20, 30, 46] have shown the advantage of overlapping computation and communication when optimizing distributed ML workloads. While our focus here is on specifying communication collectives, extending MSCCLang to further specify the scheduling of computation is an interesting future work.

Dataflow Languages. The chunk-oriented programming style of MSCCLang is motivated by *dataflow* programming languages. The design of the language is particularly influenced by declarative coordination languages such as Linda [12] and Concurrent Collections [19]. Rather than use explicit tuples, MSCCLang uses implicit chunk identifiers to coordinate multiple ranks. Cilk [23] also influenced the aspect of MSCCLang where the deterministic semantics of the program is specified by the sequential semantics of the host language.

9 CONCLUSION

MSCCLang is a novel software system designed for implementing GPU collective communications. MSCCLang provides a domain specific language for flexible collective implementations and a compiler for lowering the DSL to low-level representation that is efficiently executed by an optimized runtime. We evaluated MSCCLang by implementing the common collectives AllToAll and AllReduce on different GPU systems that outperform the state-of-the-art GPU collective library. Additionally, we introduce a custom collective, AllToNext, that demonstrates the flexibility to develop new collectives that are not in the standard MPI interface. We believe the programmability of MSCCLang will empower ML researchers to optimize existing or explore new collectives in their GPU workloads.

ACKNOWLEDGMENTS

We would like to thank our colleagues at Microsoft Research in the RiSE and Systems & Networking Group for their early feedback on this work and the Azure HPC team for their help on experiments. We also thank our anonymous reviewers for their detailed feedback.

REFERENCES

[1] AI and Compute 2022. AI and Compute. https://openai.com/blog/ai-and-compute/.
[2] 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.
[3] 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.
[4] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2021. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 62–75.
[5] 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.
[6] Minsik Cho, Ulrich Finkler, and David Kung. 2019. BlueConnect: Novel hierarchical all-reduce on multi-tired network for deep learning. In *Proceedings of the 2nd SysML Conference*.
[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] Cooperative Groups 2017. Cooperative Groups: Flexible CUDA Thread Programming. https://developer.nvidia.com/blog/cooperative-groups/.
[9] DeepSpeed 2021. DeepSpeed: Accelerating large-scale model inference and training via system optimizations and compression. https://www.microsoft.com/en-us/research/blog/deepspeed-accelerating-large-scale-model-inference-and-training-via-system-optimizations-and-compression/.
[10] Wei Deng, Junwei Pan, Tian Zhou, Deguang Kong, Aaron Flores, and Guang Lin. 2021. DeepLight: Deep Lightweight Feature Interactions for Accelerating CTR Predictions in Ad Serving. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining* (Virtual Event, Israel) *(WSDM '21)*. Association for Computing Machinery, New York, NY, USA, 922–930. https://doi.org/10.1145/3437963.3441727
[11] Jack Dongarra et al. 2013. MPI: A message-passing interface standard version 3.0. *High Performance Computing Center Stuttgart (HLRS)* 2, 5 (2013), 32.
[12] David Gelernter. 1985. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.* 7, 1 (January 1985), 80–112. https://doi.org/10.1145/2363.2433
[13] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. 2016. Scalable hierarchical aggregation protocol (SHArP): a hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*. IEEE, 1–10.
[14] William Gropp, Ewing Lusk, and Anthony Skjellum. 2014. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press.
[15] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. 2019. TicTac: Accelerating distributed deep learning with communication scheduling. (March 2019).
[16] Jahanzeb Maqbool Hashmi and Dhabaleswar K Panda. 2021. BluesMPI: Efficient MPI Non-blocking Alltoall Offloading Designs on Modern BlueField Smart NICs. In *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24–July 2, 2021, Proceedings*, Vol. 12728. Springer Nature, 18.
[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90
[18] Zhenhao He, Daniele Parravicini, Lucian Petrica, Kenneth O'Brien, Gustavo Alonso, and Michaela Blott. 2021. ACCL: FPGA-Accelerated Collectives over 100 Gbps TCP-IP. In *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE, 33–43.
[19] Intel Concurrent Collections 2021. Intel Concurrent Collections for C++. https://icnc.github.io/.
[20] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-based parameter propagation for distributed DNN training. (March 2019).
[21] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 463–479.
[22] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 741–761.
[23] Charles Leiserson and Aske Plaat. 1997. Programming Parallel Applications in Cilk. *Siam News* (07 1997).
[24] 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.
[25] Megatron GPT-3 Inference 2021. Megatron GPT-3 Large Model Inference with Triton and ONNX Runtime. https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s31578/.
[26] NVIDIA GPUDirect 2022. NVIDIA GPUDirect: Enhancing Data Movement and Access for GPUs. https://developer.nvidia.com/gpudirect.
[27] NVIDIA NCCL 2022. NVIDIA Collective Communication Library (NCCL). https://github.com/nvidia/nccl.
[28] ORT MoE 2022. ONNX Runtime Mixture of Experts. https://github.com/pytorch/ort.
[29] Parameter counts in Machine Learning 2022. Parameter counts in Machine Learning. https://www.alignmentforum.org/posts/GzoWcYibWYwJva8aL/parameter-counts-in-machine-learning.
[30] 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.
[31] 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.
[32] PyTorch on Azure 2022. PyTorch on Azure. https://azure.microsoft.com/en-us/resources/developers/pytorch/.
[33] ROCm RCCL 2022. ROCm Communication Collectives Library (RCCL). https://github.com/ROCmSoftwarePlatform/rccl.
[34] 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.
[35] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 785–808. https://www.usenix.org/conference/nsdi21/presentation/sapio

[36] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. 2019. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701* (2019).

[37] 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.

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

[39] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR* abs/1909.08053 (2019). arXiv:1909.08053 http://arxiv.org/abs/1909.08053

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

[45] Ningning Xie, Tamara Norman, Dominik Grewe, and Dimitrios Vytiniotis. 2021. Synthesizing Optimal Parallelism Placement and Reduction Strategies on Hierarchical Systems for Deep Learning. *arXiv preprint arXiv:2110.10548* (2021).

[46] 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.

Received 2022-07-07; accepted 2022-09-22