Demystifying NCCL: An In-depth Analysis of GPU Communication Protocols and Algorithms

Zhiyi Hu[1]*, Siyuan Shen[1]*, Tommaso Bonato[1], Sylvain Jeaugey[2], Cedell Alexander[3], Eric Spada[3], Jeff Hammond[2], Torsten Hoefler[1]
[1]ETH Zürich, Switzerland
zhiyhu@student.ethz.ch, {siyuan.shen, tommaso.bonato, torsten.hoefler}@inf.ethz.ch
[2]NVIDIA Corporation, {sjeaugey, jehammond}@nvidia.com
[3]Broadcom Inc., {cedell.alexander, eric.spada}@broadcom.com

Abstract—The NVIDIA Collective Communication Library (NCCL) is a critical software layer enabling high-performance collectives on large-scale GPU clusters. Despite being open source with a documented API, its internal design remains largely opaque. The orchestration of communication channels, selection of protocols, and handling of memory movement across devices and nodes are not well understood, making it difficult to analyze performance or identify bottlenecks. This paper presents a comprehensive analysis of NCCL, focusing on its communication protocol variants (Simple, LL, and LL128), mechanisms governing intra-node and inter-node data movement, and ring- and tree-based collective communication algorithms. The insights obtained from this study serve as the foundation for ATLAHS, an application-trace-driven network simulation toolchain capable of accurately reproducing NCCL communication patterns in large-scale AI training workloads. By demystifying NCCL's internal architecture, this work provides guidance for system researchers and performance engineers working to optimize or simulate collective communication at scale.

Index Terms—NVIDIA NCCL, Collective communication, Communication libraries, Multi-GPU cluster training

I. Introduction

Efficient GPU-to-GPU communication is essential for achieving high performance in distributed artificial intelligence (AI) and high-performance computing (HPC) workloads. The NVIDIA Collective Communication Library (NCCL) is a prominent library widely adopted for scalable, optimized GPU communication [1], [2]. Unlike general-purpose message-passing frameworks, such as MPI [3], NCCL specifically targets GPU-to-GPU interactions, utilizing interconnect technologies such as NVLink, PCIe, and InfiniBand (IB) to achieve high bandwidth and low latency.

Although NCCL is critical to large-scale GPU systems and open source, its internal mechanisms remain insufficiently documented. This is reflected by the frequent technical questions posted on NCCL's GitHub page [1], where users seek details about the library's inner workings. While the official API documentation is thorough, key aspects such as topology construction, algorithm selection, pipelining, and buffer management across nodes and devices are not clearly described. This lack of transparency makes it difficult for system researchers, network architects, and performance engineers to optimize, or predict NCCL's performance on new hardware and at scale [4], [5].

In this paper, we present a thorough and systematic exploration of NCCL's internal architecture. Our analysis specifically targets four primary aspects of NCCL's implementation: (1) a general overview, including API structure and communication channel management; (2) a detailed examination of communication protocols (Simple, LL, LL128); (3) an analysis of its data-transfer models; and (4) comprehensive analysis of its collective communication algorithms.

The insights gained from this study provide important context for performance modeling and architectural optimization. These insights have been adopted in simulation frameworks such as ATLAHS [6], an application-trace-driven network simulator developed to accurately replicate the communication patterns of NCCL-based machine learning workloads. By clarifying NCCL's internal design principles, this analysis supports system researchers, interconnect designers, and network architects in making more informed optimization decisions for GPU-centric high-performance computing environments.

The analysis in this paper is based on NCCL version 2.19.1. While specific implementation details may evolve in future releases, the core architectural mechanisms and communication strategies discussed here are expected to remain consistent, ensuring that the insights presented remain broadly applicable.

II. NCCL Overview

A. NCCL API

NCCL is specifically designed to provide highly optimized collective communication operations for GPU clusters, emphasizing low latency and high bandwidth. At its core, NCCL manages GPU-to-GPU communication via a clear and efficient API that abstracts complex technical details. NCCL primarily provides four categories of functions to users:

1) Communicator Management: Similar to MPI, all communication operations in NCCL are performed within the context of communicators. Each GPU participating in communication maintains a communicator object, which is used to invoke NCCL operations. Users must first initialize a communicator and define the set of GPUs involved.

When all devices are managed within a single process or thread, `ncclCommInitAll` can be used to create the communicator collectively. For multi-process or multi-threaded environments, each process calls `ncclCommInitRank` with

a shared unique identifier to correctly establish the communicator across processes.

After communication tasks have finished, communicators should be properly released to free resources. NCCL provides two functions for this purpose:

- `ncclCommDestroy`: Safely destroys a communicator, ensuring all pending communication operations are completed before cleanup.
- `ncclCommAbort`: Immediately terminates the communicator and cancels ongoing operations. This is intended for error recovery or handling unexpected failures to avoid deadlocks.

2) Collective Communication: NCCL provides 5 collective operations: `ncclAllReduce`, `ncclBroadcast`, `ncclReduce`, `ncclAllGather`, and `ncclReduceScatter`. Historically, NCCL included an in-place variant of `ncclBroadcast`, called `ncclBcast`, to mimic the behavior of `MPI_Bcast`, which always operates in-place. However, to support more general use cases and achieve a more regular API, NCCL later introduced `ncclBroadcast` with separate send and receive buffers. `ncclBcast` is now largely deprecated and maintained primarily for compatibility with MPI-style interfaces.

3) Point-to-Point Communication: NCCL supports point-to-point operations through `ncclSend` and `ncclRecv`.

4) Group Calls: To aggregate operations and reduce overhead, NCCL offers `ncclGroupStart` and `ncclGroupEnd`. These functions bracket a sequence of NCCL calls and delay their execution until the group ends. Grouped operations may include multiple Send/Recv calls (to emulate SendRecv, All-to-One, One-to-All, or All-to-All patterns) or a set of collective operations. This aggregation can significantly reduces launch overhead and latency by ensuring that all grouped operations are executed together as part of a single NCCL launch.

B. Launching Strategies

NCCL supports three common execution models for launching operations on multiple GPUs, and each of these approaches presents distinct trade-offs.

- **One CPU process per GPU:** This model provides greater control over process placement. By binding each GPU to a separate process, the associated CPU code can be scheduled on the local non-uniform memory access (NUMA) domain, improving data locality and reducing memory access latency.
- **One CPU thread per GPU:** When a single CPU process manages multiple GPUs through multiple threads, it enables efficient intra-process memory sharing. This setup allows for direct access to memory across ranks, including GPU buffers, reducing memory-copy overhead during communication.
- **One CPU thread for multiple GPUs:** While the single-threaded model suffers from sequential kernel launches and reduced concurrency, it offers simplicity, minimal CPU overhead, and deterministic execution, making it suitable for small-scale deployments or prototype environments where ease of implementation is prioritized over the highest performance.

C. Communication Channels

NCCL orchestrates communication through three hardware components: the GPU, the CPU, and the network interface. GPUs execute reductions and move data between buffers. CPUs launch kernels and manage host-side coordination, and NICs transfer packets across nodes. When only a single streaming multiprocessor (SM) handles the GPU work, large messages can overload that SM, underuse other SMs, and fail to saturate links such as NVLink or InfiniBand [7], [8].

To avoid this bottleneck, NCCL subdivides every collective into **communication channels**. Each channel is launched as a separate CUDA block that runs on its own SM, and the library partitions the input buffer so that channels operate on disjoint chunks in parallel. This fine-grained parallelism raises aggregate throughput, especially for large payloads that would otherwise serialize on one SM. Spreading work across channels also helps balance traffic across multiple NICs on NVLink platforms, as each channel can independently exit the node through a different NIC. This improves link utilization, reduces idle time, and balances load across interconnects such as NVLink, PCIe, and InfiniBand.

However, aggressive use of multiple channels can negatively impact network efficiency. When the per-channel chunk size becomes smaller than the 512 KiB FIFO buffer size employed by NIC transports, the proxy thread sends partially filled buffers. This under-utilization can degrade PCIe and network throughput, particularly when multiple queue pairs (QPs) are active to enable Equal-Cost Multi-Path Routing (ECMP) load balancing. NCCL addresses this issue by heuristically reducing `nChannels` for smaller messages (refer to the function `calcP2pChunkSize` in `enqueue.cc`). Nevertheless, selecting an optimal channel count remains a trade-off between GPU-side parallelism and network utilization efficiency.

Channel management in NCCL is coordinated at the communicator level, where each GPU receives a unique rank between 0 and $n-1$, where n is the total number of GPUs participating in the communicator. During communicator initialization, NCCL establishes an initial set of channel structures, with their total count primarily guided by system topology and architectural defaults. When a collective operation is invoked, NCCL dynamically selects the algorithm and protocol for that particular task. Based on this runtime choice, NCCL's internal tuning model then determines how many of these pre-established channels to utilize for that operation, considering the selected strategy, current message size, available bandwidth, and configured threads per channel. Although earlier versions allowed users to influence channel behavior by setting environment variables like `NCCL_NTHREADS`, such manual tuning is now discouraged. In recent versions, these settings are typically ignored and may even lead to incorrect behavior.

The logical communication topology assigned to each channel directly shapes how data flows among GPUs during

each operation. In a **ring topology**, each GPU identifies its immediate predecessor and successor to form a unidirectional communication ring. In a **tree topology**, each GPU tracks its parent and child ranks, establishing a logical communication tree. To increase bandwidth utilization, NCCL employs a double **binary tree** structure [9], [10]: no node is a non-leaf in both trees, and at most one node appears as a leaf in both. The second tree is constructed by mirroring the first when the number of nodes is even, or by a one-position shift when it is odd. These topologies are established during communicator initialization and reused across all collective operations.

For grouped point-to-point operations using `ncclGroupStart` and `ncclGroupEnd`, NCCL assigns each transfer to a separate channel when possible, enabling multiple independent sends and receives to run in parallel. This provides task-level parallelism across transfers.

III. Communication Protocols

NCCL employs multiple communication protocols to optimize data transfer efficiency during collective operations. The three protocols, **Simple**, **LL** (Low Latency), and **LL128**, are designed to achieve different trade-offs between bandwidth and latency. This section provides an overview of the mechanisms behind each protocol. Table I summarizes the key characteristics of the three protocols.

TABLE I
Comparison of NCCL Communication Protocols

	Simple	LL	LL128
Design Goal	High bandwidth	Low latency	Low latency and high bandwidth
Synchronization Mechanism	Memory fences (high overhead)	Flag-based synchronization	Flag-based synchronization
Payload	Data chunks	4B data + 4B flag	120B data + 8B flag
Bandwidth Utilization	Near peak	25 ~ 50% of peak [11]	~ 95% of peak [11]
Latency Per-hop	~ $6\mu s$	~ $1\mu s$	~ $2\mu s$

A. Simple Protocol

The *Simple protocol* is designed to maximize bandwidth utilization and is used for large message transfers. It operates by dividing the data into relatively large chunks and dispatching them across communication channels. This chunking strategy ensures that the high throughput of the network interface and GPU memory system is fully leveraged.

To preserve memory consistency, the protocol uses memory fences to enforce correct ordering and visibility of data. A receiver must wait until a full chunk has been transferred before accessing it. While effective at ensuring correctness, the use of memory fences introduces significant overhead. This overhead becomes a limiting factor for small messages, where the cost of synchronization dominates overall transfer time. As a result, while the Simple protocol achieves near-peak bandwidth for large messages, it suffers from high latency when handling small payloads.

B. LL (Low Latency) Protocol

To address the latency issues associated with the Simple protocol, NCCL includes the *LL protocol*, which is optimized for small message sizes where bandwidth is typically underutilized. Instead of relying on memory fences, the LL protocol uses lightweight flag-based synchronization. A small flag is transmitted alongside the data to signal its validity, enabling the receiver to proceed as soon as the data becomes available without requiring costly memory barriers.

Each transmission in the LL protocol consists of 4 bytes of data followed by a 4-byte flag, sent together using 8-byte atomic operations. This approach significantly reduces synchronization overhead and improves responsiveness for latency-sensitive workloads. LL forces the intermediate buffer to reside in host memory so that the CPU can poll the flag and detect when the data is ready to be sent through the NIC. This is necessary because polling GPU memory over PCIe is much slower than DRAM access and requires explicit synchronization to ensure data visibility on the host. While this design enables low latency, it prevents the use of GPU Direct Remote Direct Memory Access (RDMA), severely limiting bandwidth. As a result, LL typically achieves only 25–50 percent of peak bandwidth, depending on the interconnect. Consequently, it is preferred only for small transfers where latency is critical and bandwidth utilization is secondary.

C. LL128 Protocol

The *LL128 protocol* improves upon LL by maintaining its low-latency properties while significantly increasing bandwidth efficiency, particularly over high-performance interconnects like NVLink. Like LL, it uses flag-based synchronization to eliminate memory fences, but it transmits data in 128-byte units rather than 8-byte units. Out of the 128 bytes, 120 bytes are dedicated to data, and 8 bytes are reserved for the flag, allowing the protocol to utilize approximately 95 percent of the peak bandwidth.

On the network path, LL128 resembles the Simple protocol in that the sending GPU aggregates a relatively large chunk of data before notifying the CPU that it is ready to send. Although this chunk-based aggregation limits pipelining across nodes, LL128 still benefits from fine-grained pipelining within a node due to its smaller transmission granularity. This combination of low latency and high throughput makes LL128 well suited for a broad range of message sizes.

However, LL128 comes with stricter hardware requirements. It depends on atomic 128-byte writes, which must not be split or reordered by the memory system or interconnect. In systems where such operations are not guaranteed, due to PCIe limitations or other architectural constraints, NCCL disables LL128 to avoid data corruption. Protocol selection is thus influenced not only by message size, but also by system-level capabilities.

TABLE II
NCCL COMMUNICATION CHARACTERISTICS AND TRANSPORTS

	Intra-Node		Inter-Node	
Transport	P2P SHM NVLS	p2p.cc shm.cc nvls.cc	NET COLLNET	net_ib.cc net_socket.cc coll_net.cc
Physical Interconnect	NVLink PCIe		InfiniBand RoCE TCP/IP (Socket)	
Optimizations	GPUDirect P2P P2P_DIRECT		GPUDirect RDMA	

D. Protocol Selection and Comparison

NCCL dynamically selects among the Simple, LL, and LL128 protocols at runtime based on user settings (i.e., NCCL_PROTO), the collective algorithm, and internal performance heuristics. If not explicitly specified, NCCL uses a tuning model that factors in system topology, GPU architecture, message size, and predefined performance metrics to choose the best algorithm-protocol pair. This selection is constrained by resource availability, such as memory for protocol-specific buffers. Typically, LL/LL128 are chosen for small messages to reduce latency, while Simple is used for larger messages to maximize throughput.

IV. Data-Transfer Methods and Transport Layer

Efficient data movement is central to NCCL's communication performance, particularly in multi-GPU and multi-node environments. As summarized in Table II, NCCL employs distinct data transfer strategies and transport mechanisms depending on whether communication occurs within a single node (intra-node) or across multiple nodes (inter-node), with each transport optimized for specific hardware and interconnect types to support scalable collectives.

A. Intra-node Data Transfer

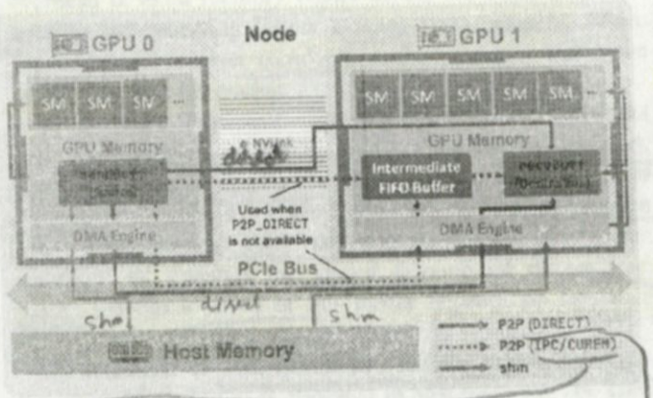

Fig. 1. Illustration of intra-node data transfer paths in NCCL. Each path is color-coded to indicate the selected transport and hardware support.

NCCL employs a sophisticated and hierarchical approach to intra-node communication, prioritizing the lowest latency and highest bandwidth paths available between GPUs residing on the same physical machine (see Figure 1). This strategy heavily leverages NVIDIA's GPUDirect Peer-to-Peer (P2P) technology, which enables GPUs to directly access each other's memory without staging through CPU system memory.

At the core of its intra-node strategy is the P2P transport, primarily managed within src/transport/p2p.cc. When GPUs are interconnected via NVIDIA NVLink, NCCL gives precedence to this path, implementing GPUDirect P2P over NVLink to utilize these dedicated, high-speed, direct GPU-to-GPU links. If NVLink is unavailable, NCCL can utilize GPUDirect P2P communication over the PCIe bus, also managed by the P2P transport layer. This offers a fallback that is generally much more performant than host-memory staging using cudaMemcpy.

A key optimization in NCCL's P2P transport is the P2P_DIRECT mode, which is enabled when communicating ranks belong to the same process. While both single-process and multi-process communications utilize GPU-to-GPU transfers without CPU involvement, P2P_DIRECT mode significantly improves efficiency in two ways. First, it bypasses the need for IPC handles by employing direct GPU memory pointers within the same address space. More importantly, it eliminates an intermediate data copy by using primitives like directSend and directRecv, which transfer data directly between source and destination buffers rather than routing through an **intermediate FIFO buffer**. Despite this optimized data path, NCCL still maintains correct synchronization using atomic head and tail counters within shared structures (e.g., ncclSendMem and ncclRecvMem) to ensure proper ordering and prevent data races. Thus, P2P_DIRECT provides substantial performance benefits through both simplified memory addressing and a more direct data transfer path, building upon the foundational GPUDirect P2P capability.

NCCL may leverage the Shared Memory (SHM) transport not only when direct GPU-to-GPU P2P communication is unavailable, but also when P2P is suboptimal. In particular, inter-socket P2P over PCIe often generates P2P packets that are poorly handled by CPUs and result in degraded performance. SHM avoids this by routing traffic through system memory, using PCIe-to-memory and memory-to-PCIe transfers, which CPUs are typically better optimized to process. In SHM mode, one GPU's controlling process writes data to a shared memory segment, which is then read by the other GPU's process.

Note that in some multi-socket systems, NCCL may use NICs for intra-node communication between GPUs when each GPU resides on a separate CPU socket with a local NIC supporting GPUDirect RDMA. Rather than traversing the CPU interconnect, NCCL may route data through a GPU–NIC–NIC–GPU path, leveraging PCIe bandwidth to avoid CPU bottlenecks. This behavior is determined by NCCL's topology-aware logic and can be controlled using environment variables such as NCCL_CROSS_NIC.

B. Inter-node Data Transfer

Inter-node communication in NCCL orchestrates data exchange between GPUs located in different physical nodes. This process involves the GPU executing NCCL kernels, a proxy

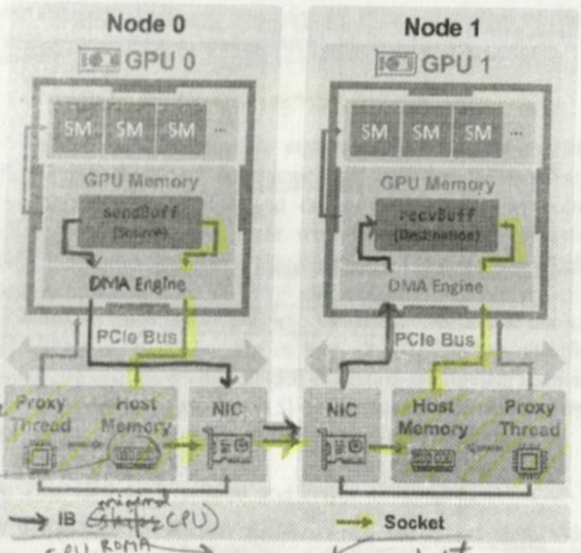

Fig. 2. Illustration of intra-node data transfer paths in NCCL. Each path is color-coded to indicate the selected transport and hardware support.

thread running on the CPU to manage network operations, and the underlying network fabric. As shown in Figure 2, NCCL selects between two primary network transports, namely a standard TCP Socket transport or a high-performance InfiniBand (IB) Verbs transport, based on the available hardware.

1) Socket-Based Communication: When the network interface does not support RDMA, NCCL employs the socket transport, implemented in `transport/net_socket.cc`. In this mode, the intermediate buffers are allocated as CUDA pinned memory in the host memory. On the sender side, data is copied from the GPU to this buffer before being sent over the network using standard socket calls. On the receiver side, data is received into the host buffer and then copied to the GPU. This reliance on the host memory as a staging area incurs the overhead of extra memory copies across the PCIe bus. Both sending and receiving follow a rendezvous protocol, where the sender and receiver coordinate buffer readiness before the actual data transfer occurs.

2) IB Verbs Transport: For high-performance networks such as InfiniBand or RoCE, NCCL uses the IB transport, implemented in `net_ib.cc`. The IB transport leverages RDMA capabilities to enable direct data movement between nodes with minimal CPU intervention. As with the socket transport, all transfers are staged through an intermediate buffer, but the location of this buffer depends on hardware support and configuration.

By default, if the NIC cannot access GPU memory directly, the intermediate buffer is allocated in host memory. The GPU kernel copies data to this buffer, and the proxy thread posts an RDMA operation to move the data from host memory to the remote node with RDMA write [12]. On the receiving side, the process is reversed: the NIC writes incoming data into a host buffer, and the proxy thread coordinates a copy from host to device memory. The proxy thread's role is to manage these DMA and RDMA operations. As with the socket transport, a rendezvous protocol is used to synchronize the sender and receiver before the data transfer.

In the following paragraphs, we highlight some key features and optimizations implemented in the IB transport.

a) The GPUDirect RDMA Optimization: A key optimization in the IB transport is GPUDirect RDMA (GDRDMA), which enables the NIC to access GPU memory directly, eliminating the need for host memory staging. GDRDMA is used only when both the NIC and GPU are connected to the same PCIe switch. In this case, the intermediate buffer is allocated in *GPU memory*. The CPU proxy thread registers this GPU memory with the RDMA-capable NIC using mechanisms such as `nv_peer_mem` [13] or the Linux DMA-BUF subsystem [14], allowing the NIC to map and access GPU memory directly. The NIC's DMA engine then performs RDMA reads or writes directly to or from the GPU, bypassing the CPU and host memory entirely.

b) Per-peer Multi-channel Connections: As an optimization to improve bandwidth utilization and reduce congestion, the IB transport instantiates 2 logical channels per remote GPU and per NIC (controlled by the `NCHANNELS_PER_NET_PEER` parameter) by default. Each logical channel maintains its own `ncclIbSendComm` structure, embedding an independent bundle of InfiniBand QPs. During execution, the host-side network proxy alternates between the two `sendComm` handles when issuing `ncclNet->isend()` calls, thereby splitting traffic across the QP sets. This lightweight round-robin strategy increases the effective per-QP chunk size, introduces path diversity for ECMP-aware fabrics, and enhances overall interconnect efficiency—all without incurring additional GPU-side state or coordination overhead.

c) QP Layout: For every pair of ranks, the RDMA plugin establishes *two* reliable connection (RC) QPs, one in each direction. The *forward* QP is responsible for the bulk data stream: the proxy issues one or more `RDMA_WRITE` work requests that push user data directly into the peer buffer, and finally a zero-byte `RDMA_WRITE_WITH_IMM`. The immediate data field of this last request encodes the transfer size and is polled by the receiver to detect completion. The *reverse* QP carries only a tiny clear-to-send (CTS) message, which is a single `RDMA_WRITE` that advertises the remote buffer address, rkeys, and tag information. While the same functionality could theoretically be multiplexed on a single QP, separating the CTS onto its own channel isolates latency-critical control traffic from the bandwidth-hungry data stream, allowing the network to deliver it with minimal head-of-line blocking.

d) Local Flush with Loop-back `RDMA_READ`: When GPUDirect RDMA is enabled, the sender must ensure that all outstanding PCIe writes reach GPU memory before the kernel consumes the data. NCCL implements this by issuing a dummy `RDMA_READ` after the last receive completes. A dedicated "flush" QP is connected to itself, meaning that its ready-to-receive (RTR) stage uses its own local QP number as the destination. Consequently, the read never leaves the host,

but the verbs layer still waits for the PCIe completion of prior writes, providing an inexpensive ordering barrier.

V. NCCL Collective Algorithms

Collective algorithms are central to NCCL, enabling efficient, synchronized communication between GPUs. They manage data movement and dependencies, optimize communication paths, and scale with increasing GPU counts. NCCL implements these algorithms by breaking each collective operation into low-level communication primitives and distributing them across multiple parallel channels. The choice of algorithm, typically ring or tree, depends on the specific collective operation and relevant execution parameters such as message size and topology [15]–[17]. This section outlines the design and main features of NCCL's collective algorithms.

A. Overview of Algorithm and Protocol Support

While NCCL provides six protocols, not all are applicable to every algorithm, and their availability may vary based on hardware features and runtime constraints.

Table III summarizes the algorithms and communication protocols supported by each of the 5 collective operations in NCCL version 2.19. This information was extracted from the corresponding header files in the `src/device` directory. In addition to the commonly used Ring and Tree algorithms, the table also highlights support for specialized algorithms, namely CollNet and NVLS. NVLS and CollNet are specialized algorithms primarily designed to optimize AllReduce performance, with NVLS also offering support for ReduceScatter and AllGather by leveraging specific hardware capabilities.

The CollNet algorithms are intended for scenarios where the network infrastructure itself can participate in collective operations, such as using NVIDIA SHARP (Scalable Hierarchical Aggregation and Reduction Protocol) technology, allowing reductions or other partial collective computations to be offloaded to network switches, thereby reducing data movement and latency [18].

CollNet algorithms leverage NVIDIA SHARP (Scalable Hierarchical Aggregation and Reduction Protocol) technology for network-assisted collective operations. CollNet Direct enables all-to-all communication within the node. In contrast, CollNet Chain arranges GPUs linearly, and performs reductions up the chain and broadcasts down [19].

NVLS algorithms are designed to take advantage of NVIDIA's NVLink Switch (NVSwitch) systems, which provide high-bandwidth, direct GPU-to-GPU communication paths within a multi-GPU server or NVSwitch fabric, enabling more efficient collective operations [2]. Both the plain NVLS and NVLS Tree algorithms use NVLink SHARP for intra-node reduction but differ in inter-node handling: NVLS continues the reduction via CollNet and SHARP-enabled switches, while NVLS Tree uses a tree-based fan-out [19].

However, this paper will not include analyses of NVLS and CollNet, as their implementations rely heavily on specific hardware (NVSwitch and SHARP-enabled networks), making them less representative. We acknowledge that NCCL continues to evolve, recently introducing additional algorithms such as Parallel Aggregated Trees (PAT) in version 2.23 [20], [21]. Nonetheless, as newer algorithms have yet to achieve widespread adoption, our subsequent discussions will remain centered on the Ring and Tree algorithms.

B. Communication Primitives

NCCL implements high-level collective operations by composing them from a set of low-level communication primitives. These primitives form the foundation of NCCL's collective algorithms, encapsulating basic operations such as sending, receiving, reducing, and copying data across GPUs.

Common primitives include `send`, `recv`, `recvReduceSend`, `recvCopySend`, and `recvReduceCopySend`, along with their "direct" variants discussed in Section IV-A. Each primitive represents a distinct data movement or computation pattern, with naming conventions that clearly indicate the sequence of operations. For example, `recvReduceSend` denotes a step in which a GPU receives data from a peer, performs a reduction with its local buffer, and sends the result to the next GPU. During execution, the NCCL runtime dispatches these primitives iteratively across loop steps, enabling flexible coordination across different algorithms, topologies, and transport layers.

The behavior of each NCCL primitive is further shaped by the selected communication protocol. Synchronization, buffer management, and transfer granularity vary depending on whether the Simple, LL, or LL128 protocol is used. It is important to note that these low-level primitives are heavily optimized for collectives with a fixed, small number of sources and destinations, such as rings and trees, which typically involve one source and one destination (or up to three for certain tree topologies). While this approach enables high efficiency for many standard collective algorithms, it is less effective for patterns like all-to-all, which require handling N sources and N destinations.

C. Iterative Execution of NCCL Collectives

NCCL processes collective operations by first dividing the user's input data among the available communication channels, enabling parallelism at the channel level. Each channel is responsible for a contiguous segment of the input, determined by the total number of elements (`count`) and the number of channels. This partitioning is visualized in Figure 3, where the total data is split so that each channel, such as Channel 0 and Channel 1, operates independently on its assigned region. The start index for each channel's work is given by `workOffset`, and the size by `channelCount`.

To facilitate efficient data transfer and computation, NCCL allocates a fixed-size buffer for each channel, the capacity of which depends on the chosen communication protocol (Simple, LL, or LL128, as shown in Table IV). If a channel's data region is larger than its total buffer, NCCL breaks the data into several outer loop iterations. Each iteration processes a segment of data up to the size of the buffer (`loopCount`

TABLE III
SUPPORTED ALGORITHMS AND PROTOCOLS FOR NCCL COLLECTIVE OPERATIONS

	AllReduce			Broadcast			Reduce			ReduceScatter			AllGather		
Algorithm	Simple	LL	LL128	Simple	LL	LL128	Simple	LL	LL128	Simple	LL	LL128	Simple	LL	LL128
Ring	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Tree	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
CollNet Direct	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
CollNet Chain	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
NVLS	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓	✗	✗
NVLS Tree	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

Legend: ✓= Supported, ✗= Not supported.

TABLE IV
NCCL CHANNEL BUFFER SIZES FOR EACH PROTOCOL UNDER THE DEFAULT CONFIGURATION

Protocol	Total Channel Buffer Size	Buffer Capacity per Slot	Effective Data per Slot
Simple	4 MiB	512 KiB	512 KiB
LL	256 KiB	32 KiB	16 KiB
LL128	~4800 KiB	600 KiB	562.5 KiB

elements per iteration), and the channel cycles through as many loops as needed to cover all its assigned elements.

Within each outer loop iteration, NCCL implements pipelining by dividing the channel buffer into a fixed number of segments, known as **slots**. It is typically 8, and set by the `NCCL_STEPS` parameter. Each slot can independently advance through different stages of communication and computation, allowing the pipeline to overlap data transfers with reduction or copy. During each **elementary step**, which follows the communication primitives described in Section V-B, a chunk of data (`chunkCount` elements, or `lastChunkCount` for the final chunk in a loop) is processed and mapped to the buffer slots. This chunking mechanism allows NCCL to keep the communication channels busy, overlapping new chunks with ongoing operations for maximum throughput.

In NCCL, the basic unit of data movement is called an element, and its meaning depends on the collective operation. For `ncclAllGather` and `ncclBroadcast`, each element is a single byte, since these operations focus on efficiently moving and concatenating data. This byte-level granularity gives NCCL flexibility in packing and transferring data, independent of the underlying type. For `ncclAllReduce`, `ncclReduceScatter`, and `ncclReduce`, each element corresponds to the user-defined data type (e.g., `float` or `int`), because these operations require arithmetic reductions that are meaningful only at the data type level.

Figure 3 shows this process in action. Each cell in the figure represents one data element in `sendBuff`. For illustrative purposes, this example assumes that `channelCount` equals 2, `chunkCount` equals 2, and `loopCount` equals 4. Channel 0 starts at its `workOffset` and processes elements in loop iterations of `loopCount`, breaking them further

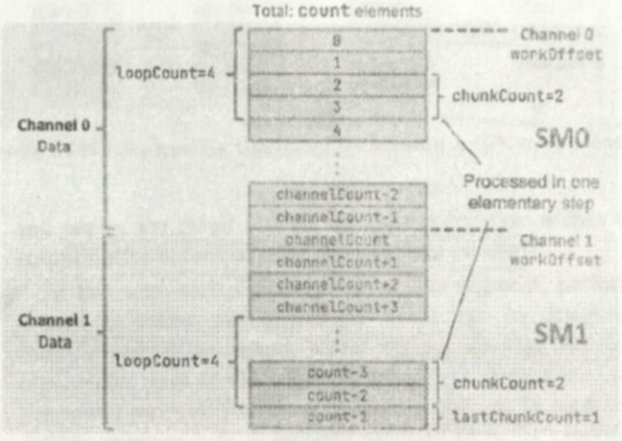

Fig. 3. Visualization of NCCL's data partitioning strategy across communication channels and loop iterations.

into chunks of `chunkCount`. Channel 1 follows the same logic for its own region. By coordinating this partitioning and pipelining, NCCL achieves efficient, parallel, and scalable collective operations across all participating GPUs.

D. Qualitative Algorithm Analysis

Now that we have established that all common NCCL collective algorithms follow an iterative processing model, an important difference lies in whether GPUs can pipeline consecutive loop iterations. Based on this characteristic, algorithms can be grouped into two categories: pipelined and non-pipelined. In the following sections, we organize the collective algorithms accordingly and provide a qualitative analysis of each. For every algorithm, we describe the specific sequence of elementary steps executed within each loop iteration.

While we initially considered a quantitative complexity analysis that would bound algorithm runtimes in terms of parameters such as data size and the alpha-beta model, we found this approach impractical because of the large number of factors that influence performance. Variables such as how GPUs are distributed across nodes have a major effect. For example, 4 GPUs on a single node experience very different bandwidth and latency compared to 4 GPUs placed on separate nodes. Including all these variables would make the model

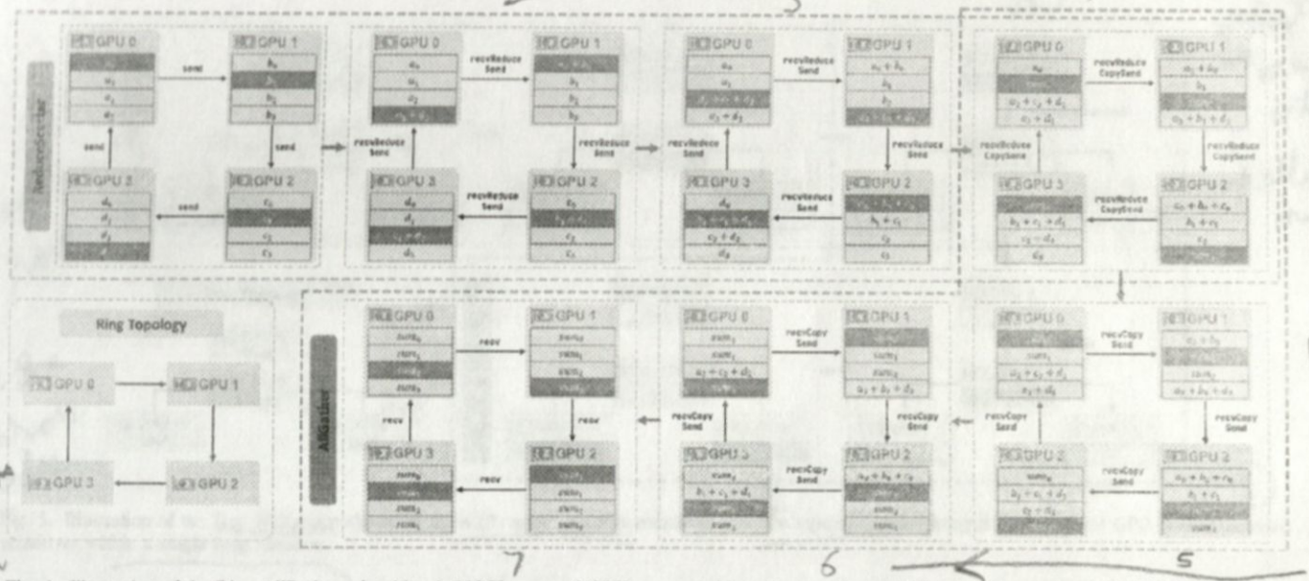

Fig. 4. Illustration of the Ring AllReduce algorithm in NCCL across 4 GPUs connected in a ring topology, highlighting the sequence of GPU communication primitives within a single loop iteration.

too complex and contradict the goal of keeping complexity bounds simple and useful. For this reason, our analysis remains qualitative and focuses on the essential behaviors rather than attempting to provide detailed theoretical runtime estimates.

1) Non-pipelined Pattern: In the non-pipelined pattern, each GPU must complete all tasks in one iteration before starting the next. Ring AllReduce, Ring AllGather, and Ring ReduceScatter follow this pattern. In the analysis below, k denotes the number of GPUs participating in the collective.

a) Ring AllReduce: The Ring AllReduce algorithm in NCCL combines a distributed reduction phase with a data dissemination phase to ensure all k participating GPUs receive the complete, element-wise reduced result. The operation is divided into $2k-1$ steps per loop, as detailed in Table V.

TABLE V
STEPS IN ONE LOOP ITERATION OF NCCL RING ALLREDUCE

Step Index	NCCL Primitive
0	`send`
1 to $k-2$	`recvReduceSend`
$k-1$	`recvReduceCopySend`
k to $2k-3$	`recvCopySend`
$2k-2$	`recv`

The Ring AllReduce algorithm begins with a ReduceScatter-like phase, illustrated in the upper portion of Figure 4. Initially, in Step 0, each GPU sends one segment of its local data to its neighbor. In the next $k-2$ steps, each GPU repeatedly executes a `recvReduceSend` operation: it receives a data segment from its preceding neighbor, performs an element-wise reduction with the corresponding segment of its local data, and forwards the reduced result to the subsequent GPU in the ring. This iterative reduction continues until Step $k-1$. At this step, each GPU receives a data segment, performs a final reduction, thereby producing the fully reduced segment, and copies the result into its designated location within the output buffer before sending this segment onward.

At this step, each GPU receives a data segment, performs a final reduction, and copies the result into its designated location within the output buffer before sending this fully reduced segment onward. For the next $k-2$ steps, each GPU executes a series of `recvCopySend` operations. In each step, a GPU receives a fully reduced segment from its preceding neighbor, copies it directly into the appropriate position in its output buffer, and forwards this segment unchanged to the next GPU. The Ring AllReduce operation concludes at Step $2k-2$, with each GPU performing a final `recv` to complete the collection of fully reduced data.

b) Ring AllGather: The Ring AllGather algorithm enables each of the k participating GPUs to collect a complete set of data blocks contributed by all ranks. The algorithm proceeds over $k-1$ communication steps using a logical ring topology that connects the GPUs.

In the initial step (Step 0 in Table VI), each GPU i prepares its local data block. If the operation is in-place, the block is already located in the ith segment of the output buffer. Otherwise, the GPU copies the data from its input buffer into that segment using the `copySend` primitive. After this setup, each GPU sends its local block to its right-hand neighbor.

Over the next $k-2$ steps, each GPU performs a sequence of `recvCopySend` operations. In each step, a GPU receives a block from its left-hand neighbor, stores it in the correct segment of the output buffer, and forwards it to the right-hand neighbor. The final step is a `recv` operation that delivers the last missing block. After this step, all GPUs hold a complete, ordered copy of the collective data.

c) Ring ReduceScatter: The Ring ReduceScatter algorithm performs an element-wise reduction across data blocks initially distributed over k GPUs, followed by scattering

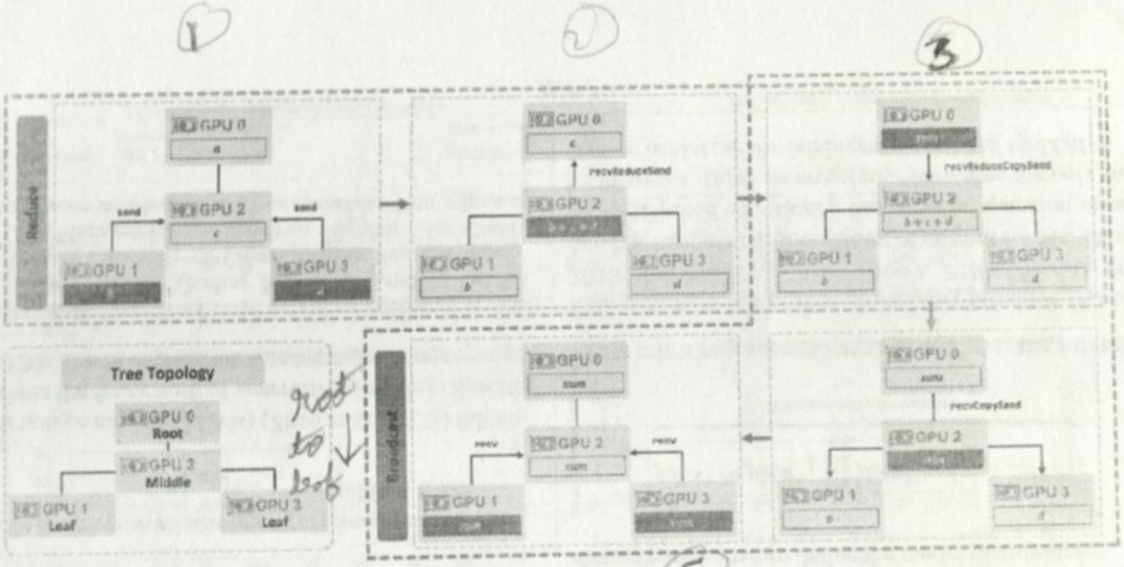

Fig. 5. Illustration of the Tree AllReduce algorithm in NCCL across 4 GPUs connected in a tree topology, highlighting the sequence of GPU communication primitives within a single loop iteration.

TABLE VI
STEPS IN ONE LOOP ITERATION OF NCCL RING ALLGATHER

Step Index	Primitives
0	`send` (In-place operation) or `copySend`
1 to $k-2$	`recvCopySend`
$k-1$	`recv`

unique segments of the fully reduced result back to each GPU. At the beginning, each GPU's `sendbuff` contains k distinct data blocks, which are progressively reduced as they move around a logical ring topology.

Table VII summarizes the primitives executed in each step of a single loop iteration of Ring ReduceScatter. In the initial step, each GPU i sends one of its local data blocks to its immediate neighbor GPU $(i+1)\%k$, initiating data movement around the ring. During the subsequent $k-2$ steps, each GPU performs a series of `recvReduceSend` operations: it receives a partially reduced data block from its left neighbor (GPU $(i-1)\%k$), combines this block element-wise with its corresponding local block stored in `sendbuff`, and sends a different partially reduced block onward to its right neighbor. In the final step, each GPU receives one last data block from its left neighbor, applies the final reduction operation, and copies the fully reduced result directly into its own `recvbuff`.

TABLE VII
STEPS IN ONE LOOP ITERATION OF NCCL RING REDUCESCATTER

Step Index	Primitives
Step 0	`send`
Step 1 to Step $k-2$	`recvReduceSend`
Step $k-1$	`recvReduceCopy`

2) Pipelined Pattern: The Tree AllReduce, Ring Broadcast, and Ring Reduce algorithms in NCCL follow a pipelined execution pattern.

a) Tree AllReduce: The Tree AllReduce algorithm proceeds in two distinct phases within each loop iteration: a **Reduce** phase followed by a **Broadcast** phase. The data movement is illustrated by an example involving 4 GPUs in Figure 5. Although the illustration shows a complete tree over four ranks, it is important to note that the branching structure is built only across nodes. Inside each node, NCCL links the local GPUs in a simple chain. In an alternative implementation in NCCL, these two phases are often executed concurrently by partitioning the SMs into two uneven groups. One group handles the reduction toward the root, while the other simultaneously performs the broadcast from the root. This asymmetric allocation dedicates more threads to the bandwidth-intensive reduction phase, enabling better utilization of available resources.

In the Reduce phase, leaf GPUs initiate the reduction by sending their local data upward to their parent using a `send` operation. Middle GPUs receive data from one or more children using the `recvReduceSend` primitive, perform element-wise reduction with their own data, and pass the result upward. Finally, the root GPU performs a `recvReduceCopySend`, completing the reduction by combining the incoming data with its local buffer and copying the fully reduced result into the user-provided output buffer.

In the Broadcast phase, the fully reduced result is propagated back down the tree. The root sends the result to its children using a `recvCopySend` operation. Middle GPUs receive the data from their parent, copy it into their own output buffer, and forward it to their children using the same `recvCopySend` primitive. Leaf GPUs receive the data using a simple `recv` and copy it into their output buffer.

The sequence of device primitives used by each type of GPU role is summarized in Table VIII.

b) Ring Broadcast: The NCCL Ring Broadcast algorithm disseminates data from a user-specified root GPU to all other GPUs in the communicator. Although it uses a ring

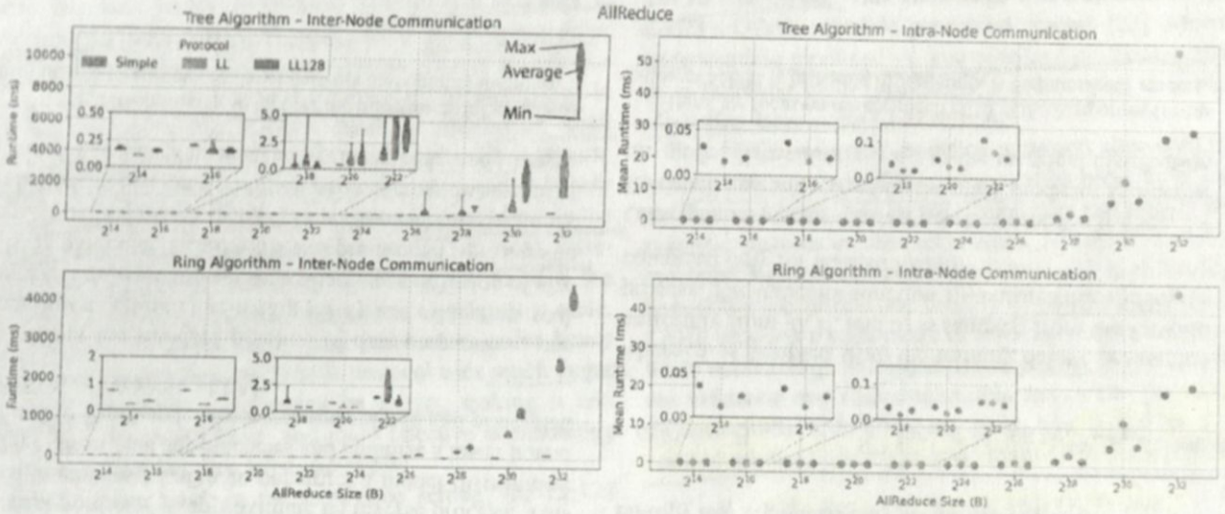

Fig. 6. Runtime comparison of protocols for Ring and Tree AllReduce when running inter- and intra-node. Each data point consists of 20 runs with a warm-up phase. For intra-node communication we report only the median value for readability as the variance is very low.

TABLE VIII
STEPS IN ONE LOOP ITERATION OF NCCL TREE ALLREDUCE

GPU Role	Primitives
Root	`recvReduceCopySend`
Middle	`recvReduceSend` and then `recvCopySend`
Leaf	`send` and then `recv`

topology, the communication pattern effectively forms a directed chain, beginning at the root and progressing sequentially through each GPU until the last one receives the data.

The operation begins with the root GPU. As indicated in Table IX, the root either performs an in-place `send` operation if its send buffer is also its receive buffer, or a `copySend` where data from its distinct send buffer is first copied to its receive buffer and then transmitted. In either case, the root sends its data block to its immediate successor in the ring. Each subsequent GPU in the middle of the chain executes a `recvCopySend` primitive: it receives the data block from its predecessor, copies it into its own receive buffer, and then forwards the data block to its successor. This process continues until the data reaches the last GPU in the chain. This last GPU simply performs a `recv` operation, copying the incoming data into its receive buffer, and does not send further, as all GPUs in the logical chain have now received the broadcast data.

TABLE IX
STEPS IN ONE LOOP ITERATION OF NCCL RING BROADCAST

GPU Role	Primitives
Root	`send` (in-place) or `copySend`
Middle	`recvCopySend`
Last	`recv`

c) Ring Reduce: The NCCL Ring Reduce algorithm performs an element-wise reduction of data distributed across multiple GPUs, aggregating the final result onto a user-defined root GPU. Like Ring Broadcast, the operation leverages a logical chain derived from the ring topology, along which data flows and accumulates toward the root.

As shown in Table X, the chain begins with the first GPU sending its local data block to the next GPU in the ring. Intermediate GPUs perform the `recvReduceSend` primitive: each receives a partially reduced block, applies an element-wise reduction using its own corresponding data, and forwards the updated result to the next GPU. This process repeats until the data reaches the destination root. The root GPU completes the operation with a `recvReduceCopy`: it receives the final partial result, reduces it with its local data, and stores the fully reduced output in its receive buffer.

TABLE X
STEPS IN ONE LOOP ITERATION OF NCCL RING REDUCE

GPU Role	Primitives
Initiator	`send`
Middle	`recvReduceSend`
Root	`recvReduceCopy`

E. Benchmarking

In this section, we present benchmarking results for NCCL collectives. Figure 6 highlights the runtime performance of the three NCCL communication protocols for AllReduce in both intra-node and inter-node settings, offering a representative view of their behavior. Experiments were conducted on the Alps supercomputing system at the Swiss National Supercomputing Center (CSCS), using 16 nodes equipped with NVIDIA Grace Hopper Superchips (GH200). Each node provides a 150GB/s high-bandwidth intra-node interconnect and connects to the Cray Slingshot interconnect via a 25GB/s per-direction network link [5], [22].

In the inter-node setting, for both Tree and Ring algorithms, LL and LL128 perform best for small messages (less than 64

KiB). However, as the AllReduce message size increases to the gigabyte range across 16 nodes, their performance drops sharply compared to the Simple protocol. This is mainly due to the overhead of fine-grained, flag-based synchronization in LL and LL128, which requires handling millions of small sync operations (one per 8 or 128 bytes) across the network. While LL128 benefits from larger buffer sizes and is highly efficient over NVLink, these advantages are outweighed by the cumulative synchronization cost over RoCE for large, inter-node transfers. LL128 can even lag behind LL because the extra cost per 128-byte operation becomes significant at scale, or because stalls affect larger data units more under heavy contention. By contrast, the Simple protocol uses much larger transfers with fewer synchronization events, making it less sensitive to network latency and more effective at sustaining high throughput for very large messages.

On the other hand, in the intra-node setting, the LL128 protocol shines with consistent performance across all the message sizes thanks to its ability to fully take advantage of the NVLink connection. In particular, for small messages, LL128 performs as well or only slightly worse than LL while almost matching the performance of Simple (it is 5% slower than Simple, as expected from Table I) at large messages. The remaining two protocols, LL and Simple perform their best at the opposite extremes, with Simple providing the best performance for large messages and LL for small messages. Finally, we observe that in both intra- and inter-node settings, the Ring algorithm excels for large messages, whereas the Tree algorithm performs best for smaller messages.

There are three takeaways from this benchmarking experiment. First, the results confirm expectations: LL and LL128 are best suited for small messages, especially for inter-node communication, while the Simple protocol consistently outperforms the others for large, distributed transfers. Second, it is important to consider whether the communication is intra-node or inter-node, as different transport algorithms, particularly LL128, exhibit noticeably different performance across these configurations. Finally, while manual protocol selection can be useful for targeted tuning, in most cases it is beneficial to rely on NCCL's autotuning. Allowing NCCL to select the protocol based on workload characteristics generally provides robust performance and scalability across most use cases.

In addition to AllReduce, we also benchmarked the other collective algorithms. As their behavior follows the same trends observed in AllReduce, we present their runtime results in Figure 7 in the appendix.

VI. Integration into ATLAHS

Our deep analysis of NCCL's internal communication patterns, algorithms, and pipelined processing modes has significantly guided the design and capabilities of the ATLAHS toolchain [6]. By characterizing the primitives, data dependencies, and timing behaviors of NCCL operations down to their iterative execution across CUDA streams and communication channels, we were able to accurately decompose collective communication into fine-grained computation, send, and receive events. This knowledge was instrumental in ATLAHS's GOAL schedule generation process [23]. Moreover, understanding pipelined vs. non-pipelined collectives allowed us to faithfully model concurrency and overlap, essential for simulating large-scale LLM training.

This NCCL-informed modeling approach allows ATLAHS to accurately emulate GPU communication behavior in real AI training workloads. Unlike prior simulators that largely rely on synthetic patterns or abstract models, ATLAHS captures the execution logic of collective operations with high fidelity. By embedding this insight into the GOAL schedule generator, ATLAHS supports a wide range of topologies and configurations while maintaining simulation errors below 5%. As shown in our validation and case studies, this design enables ATLAHS to outperform state-of-the-art tools like AstraSim [24] in runtime prediction in large-scale multi-GPU environments.

VII. Related Work and Outlook

Recent studies have offered detailed analyses and performance evaluations of collective communication libraries such as NCCL, MPI, and Gloo in distributed deep learning and HPC environments. For instance, Lee and Lee [25] conducted an empirical study comparing these libraries under various training architectures and deployment settings, highlighting NCCL's clear advantage in intra-node GPU-to-GPU communication, particularly for large-scale All-Reduce operations, but also pointing out its performance degradation under virtualization or containerization overheads. Other works, such as the comprehensive survey by Weingram et al. [26] provide a broad perspective on the ecosystem of collective libraries, reviewing industry solutions like NCCL, RCCL, oneCCL, and Gloo, and noting that while NCCL remains the gold standard for GPU-centric collectives, alternative libraries are rapidly evolving to match its optimizations and hardware support. While studies provide valuable insights into performance and architectural choices, most focus on empirical benchmarks, high-level comparisons, or specific algorithmic innovations. Our work differs by providing an in-depth, systematic analysis of the internal iterative execution algorithms, communication protocols, and data dependencies in collective implementations.

Despite its wide adoption, NCCL faces pressure from recent advances emphasizing adaptability, topology awareness, and fault tolerance. Emerging libraries like Blink achieve notable speedups by dynamically building multiple trees and exploiting advanced network topologies, outperforming NCCL's ring and tree algorithms in large-scale and heterogeneous clusters [26], [27]. Automated frameworks such as SCCL [28] further push the frontier by synthesizing and tuning collectives for specific hardware, surpassing hand-optimized routines. As distributed AI workloads become more resource-intensive and long-running, fault tolerance and resilience have also become critical requirements [26]. To keep pace, we believe that future versions of NCCL will need to support automated algorithm selection, robust failure handling, and tighter integration with next-generation fabrics that offer features like in-network computation and smart NICs. Such enhancements are essential

for sustaining high performance, scalability, and reliability in ever-more demanding distributed training environments.

VIII. Conclusion

This paper presents a systematic and in-depth analysis of the NVIDIA Collective Communication Library (NCCL). Our investigation examined NCCL's communication protocols, emphasizing their design trade-offs and dynamic selection logic, as well as the data transfer mechanisms employed in both intra-node and inter-node settings. We also analyzed NCCL's widely used ring and tree-based collective algorithms, detailing their iterative execution models and the sequence of GPU communication primitives they employ. These insights go beyond academic interest as they are foundational to the development of ATLAHS, an application-trace-driven network simulation toolchain capable of accurately modeling the communication behavior of large-scale AI training workloads. By uncovering the internal mechanisms and performance-critical decisions in NCCL, this work provides system researchers, network architects, and performance engineers with the insights needed to diagnose bottlenecks, optimize communication patterns, and inform the design of future high-performance collective libraries.

IX. Acknowledgment

The authors would like to thank Tiancheng Chen for his helpful suggestions. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 program (grant agreement PSAP, No. 101002047). We also thank the Swiss National Supercomputing Center (CSCS) for supporting this project and providing the computational resources used in this work. The authors used ChatGPT-4o and 4.5 to assist with light editing and proofreading throughout the manuscript. All content and ideas are the original work of the authors.

References

[1] NVIDIA, "Nccl (nvidia collective communications library)." https://github.com/NVIDIA/nccl, 2025.

[2] NVIDIA Corporation, *NVIDIA Collective Communications Library (NCCL) Documentation*, 2025. Accessed: 2025-01-28.

[3] M. P. Forum, "Mpi: A message-passing interface standard," tech. rep., USA, 1994.

[4] D. De Sensi, L. Pichetti, F. Vella, T. De Matteis, Z. Ren, L. Fusco, M. Turisini, D. Cesarini, K. Lust, A. Trivedi, D. Roweth, F. Spiga, S. Di Girolamo, and T. Hoefler, "Exploring gpu-to-gpu communication: Insights into supercomputer interconnects," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 1–15, IEEE, Nov. 2024.

[5] L. Fusco, M. Khalilov, M. Chrapek, G. Chukkapalli, T. Schulthess, and T. Hoefler, "Understanding data movement in tightly coupled heterogeneous systems: A case study with the grace hopper superchip," 2024.

[6] S. Shen, T. Bonato, Z. Hu, P. Jordan, T. Chen, and T. Hoefler, "Atlahs: An application-centric network simulator toolchain for ai, hpc, and distributed storage," 2025.

[7] NVIDIA, "NCCL GitHub Issue #578: Ring AllReduce performance discrepancy." https://github.com/NVIDIA/nccl/issues/578.

[8] NVIDIA, "NCCL GitHub Issue #1302: Broadcast logic and topology insights." https://github.com/NVIDIA/nccl/issues/1302.

[9] P. Sanders, J. Speck, and J. L. Träff, "Full bandwidth broadcast, reduction and scan with only two trees," in *Proceedings of the 14th European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, PVM/MPI'07, (Berlin, Heidelberg), p. 17–26, Springer-Verlag, 2007.

[10] T. Hoefler and D. Moor, "Energy, Memory, and Runtime Tradeoffs for Implementing Collective Communication Operations," *Journal of Supercomputing Frontiers and Innovations*, vol. 1, pp. 58–75, Oct. 2014.

[11] S. Jeaugey, "Distributed deep neural network training: Nccl on summit." https://www.olcf.ornl.gov/wp-content/uploads/2019/12/Summit-NCCL.pdf, 2019. Presentation slides.

[12] zegao96, "NCCL GitHub Issue # 609: why uses rdma write for default ib traffic." https://github.com/NVIDIA/nccl/issues/609, Dec. 2021. Issue #609. Accessed: 2025-05-24.

[13] NVIDIA Corporation, *Developing a Linux Kernel Module using GPUDirect RDMA*. NVIDIA Corporation, 2025. Last updated May 1, 2025; accessed May 24, 2025.

[14] Linux Kernel Documentation Project, *Buffer Sharing and Synchronization (dma-buf)*. kernel.org, 2025. Accessed: 2025-05-24.

[15] R. Rabenseifner, "Optimization of collective reduction operations," in *Computational Science - ICCS 2004* (M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, eds.), (Berlin, Heidelberg), pp. 1–9, Springer Berlin Heidelberg, 2004.

[16] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.

[17] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[18] NVIDIA, "NCCL Issue #320: NVLS and CollNet Support." https://github.com/NVIDIA/nccl/issues/320, 2021. Accessed: 2025-05-19.

[19] NVIDIA, "Nccl github issue #919: Question: Nccl tree algorithm behaviour." https://github.com/NVIDIA/nccl/issues/919, 2023. Accessed: 2025-06-16.

[20] S. Jeaugey, G. Congiu, T. Gillis, B. Williams, and F. Oh, "New scaling algorithm and initialization with nvidia collective communications library 2.23," Jan. 2025. Accessed: 2025-05-19.

[21] S. Jeaugey, "Pat: a new algorithm for all-gather and reduce-scatter operations at scale," 2025.

[22] CSCS, "New research infrastructure: 'alps' supercomputer inaugurated," *Swiss National Supercomputing Center*.

[23] T. Hoefler, C. Siebert, and A. Lumsdaine, "Group operation assembly language - a flexible way to express collective communication," in *2009 International Conference on Parallel Processing*, pp. 574–581, 2009.

[24] W. Won, T. Heo, S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, "Astra-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale," 2023.

[25] S. Lee and J. Lee, "Collective communication performance evaluation for distributed deep learning training," *Applied Sciences*, vol. 14, no. 12, 2024.

[26] A. Weingram, Y. Li, H. Qi, D. Ng, L. Dai, and X. Lu, "xccl: A survey of industry-led collective communication libraries for deep learning," *Journal of Computer Science and Technology*, vol. 38, no. 1, pp. 166–195, 2023.

[27] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. R. Devanur, and I. Stoica, "Blink: Fast and generic collectives for distributed ML," *CoRR*, vol. abs/1910.04940, 2019.

[28] Z. Cai, Z. Liu, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, and O. Saarikivi, "Synthesizing optimal collective algorithms," *CoRR*, vol. abs/2008.08708, 2020.

Appendix

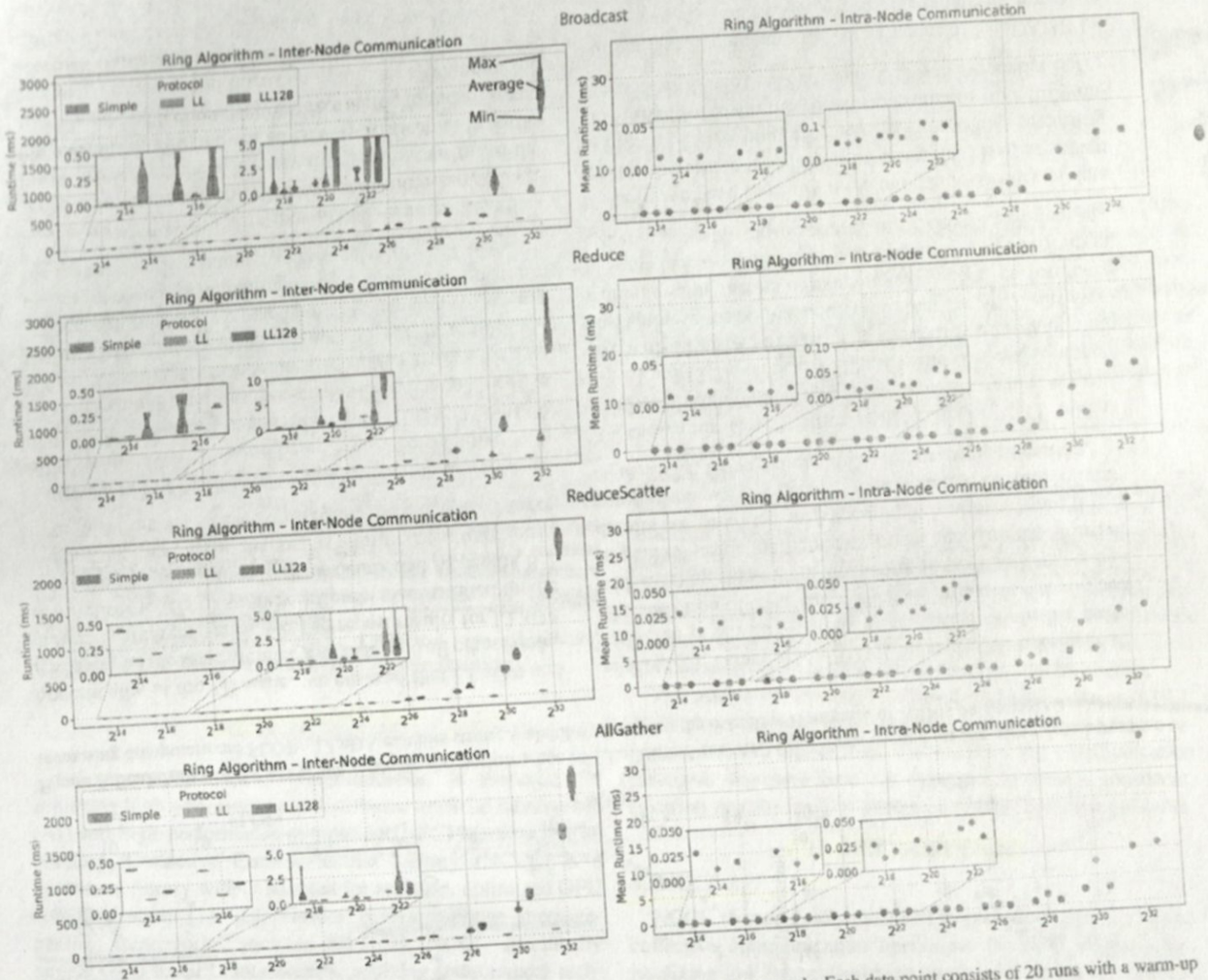

Fig. 7. Runtime comparison of protocols for various NCCL collectives when running inter- and intra-node. Each data point consists of 20 runs with a warm-up phase. For intra-node communication we report only the median value for readability, as the variance is very low.