

# CCLINSIGHT: Unveiling Insights in GPU Collective Communication Libraries via Primitive-Centric Analysis

ICSE 2026 Submission #350 – Confidential Draft – Do NOT Distribute!!

## Abstract

As distributed AI workloads scale in complexity, GPU-based collective communication libraries (xCCLs) such as NCCL are becoming increasingly critical for efficient data movement across heterogeneous hardware. Characterizing and optimizing the interactions between hardware architectures, communication algorithm designs, and low-level primitive configurations is essential to fully utilize GPU and interconnect resources while ensuring scalability. However, existing analysis methods often lack depth, limiting optimization insights, leaving significant performance gains untapped, or lack generality. To address these limitations, we propose a **novel primitive-based profiling and analysis method** that integrates architectural, algorithmic, and primitive-level insights to comprehensively **qualify and quantify performance-critical parameters** in GPU-based CCLs that we implement and test in our tool CCLINSIGHT. Evaluating three large-scale GPU clusters with diverse GPU architectures featuring up to 64 H100s, 64 RTX 5000s, and 256 A100s, CCLINSIGHT uncovers critical parameter influences on communication efficiency and scalability, as well as **exposing a new CCL performance scaling law**. Adjusting parameters based on this analysis, we demonstrate a 43.90% improvement in NCCL and a 40.64× speedup in MSCCL over default configurations in the collective communication microbenchmark, while also improving NCCL and MSCCL performance on LLM training workloads (GPT-3, LLaMA2, DeepSeek-R1) by up to 2.27× and 3.00×, respectively, on a 256-GPU H200 cluster. These contributions establish CCLINSIGHT as a useful tool for xCCL characterization and optimization in distributed AI environments.

## 1 Introduction

Large-scale Graphics Processing Unit (GPU)-based Artificial Intelligence (AI) training and inference workloads are increasingly dominating today’s datacenter and HPC environments. Underpinning these AI systems is a critical, often underappreciated infrastructure: the **Accelerated Collective Communication Library (xCCL)**. These libraries, such as NVIDIA’s NCCL, Microsoft’s MSCCL, and AMD’s RCCL, form the communication substrate on which distributed AI training and inference<sup>1</sup> is realized. xCCLs enable high-speed, large-scale coordination among the accelerators by providing core collective operations including AllReduce (critical for data parallelism), AllGather, ReduceScatter (both of which are needed in fully-sharded data parallelism), AllToAll (used in expert parallelism) as well as point-to-point operations [31, 32, 47]. Given that communication can account for over 50% of training time in

<sup>1</sup>The impact of xCCLs extends beyond AI; they are indispensable to many scientific workloads on exascale systems like Frontier and El Capitan which rely on xCCLs to orchestrate data movement with high performance.

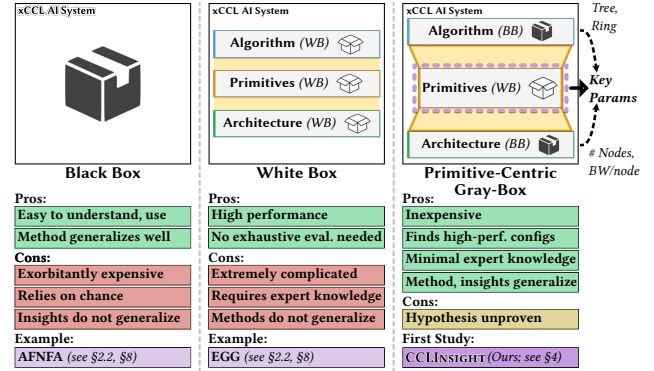


Figure 1: Black Box (BB), White Box (WB), and CCLINSIGHT compared.

large-scale distributed workloads [40], even modest improvements in xCCL performance can translate to meaningful reductions in time-to-convergence and computational costs. However, optimizing and scaling xCCLs remains challenging, despite numerous prior attempts employing diverse methodologies [25, 28–30, 33, 33, 41].

Black box approaches (Fig. 1 left) attempt to analyze the xCCL without knowledge of its internal mechanisms, focusing only on their configurable parameters and resulting performance. Beyond the choice of collective and number of accelerators involved in communication, each library exposes tens of low-level parameters. These include the algorithm, chunk size, number of channels, number of tiles, queue depth, and others. Parameters interact with the accelerator, network interface, host, and sometimes each other to affect performance in non-trivial ways. For a fixed 128-GPU cluster and three representative message sizes, reliably finding a global optimal using an exhaustive sweep exceeds **65,000 GPU-days**, or 512 real days, assuming a per-experiment runtime of 15 seconds (additional details are given in §8.2). Worse still, the optimal configuration frequently changes for different GPU scales and message sizes, e.g., the optimal configuration for a workload using 4 GPUs and 4 MB message sizes is rarely optimal for a workload using 128 GPUs and 256 MB messages. Including GPU scale and message size, the exploration time explodes to over 44 years. To illustrate, this task is analogous to trying to find a single golden needle in the ocean. Some works attempt to mitigate the effects of a large search space by using sampling. Unfortunately, such approaches lack rigor and leave the performance results entirely to chance.

White-box approaches (Fig. 1 middle) such as detailed performance modeling and cross-stack profiling leverage expert insights and known assumptions about the workload and hardware. In the right context, they can achieve high performance with little computational cost. However, expert knowledge is rare and difficult to acquire and it is unrealistic to expect AI practitioners to also be communication systems experts. Both the methods and the results of

white-box analysis rarely generalize to other clusters or workloads, as we elaborate on in §8.1. Moreover, accelerator-based execution makes performance observation challenging; standard profiling tools introduce prohibitive overhead while vendor-specific tools often lack the flexibility needed for systematic analysis and usually only apply to specific xCCLs. The heterogeneous nature of modern clusters, with varying combinations of accelerators, interconnects, and network topologies, means that optimizations and profiling tools must adapt to diverse and evolving hardware configurations.

## 1.1 Our Insights and Contributions

With the foregoing in mind, it is clear that *a new xCCL profiling and analysis method is needed*. We begin by challenging the assumption that a global optimal configuration needs to be found. Instead, **we center our approach around finding a “good enough” configuration that considerably improves xCCL performance while massively reducing the cost of finding it**. Next, we carefully consider the importance of choosing the right level of abstraction to focus on in our approach, as considering all layers of the stack is inherently expensive in either compute time (black box, as in Fig. 1 left) or in terms of required expertise (white box, as in Fig. 1 middle).

**We hypothesize that profiling only at the primitive level is sufficient** to understand xCCL performance, given certain basic information about the architecture and algorithm layers. Rather than treating xCCLs as monolithic black boxes or diving into hardware-specific implementation details, we focus only on the communication primitives that form the basis of all collective algorithms. Our key insight is that these primitives provide the optimal level of abstraction: they are universal enough to apply across different collective operations, hardware configurations, and algorithms; detailed enough to capture the performance impacts of parameter choices; limited to a fixed set of operations, which reduces complexity; and able to be profiled with low overhead. In summary, our contributions are as follows:

- ① We hypothesize that a **primitive-centric analysis approach** is sufficient to study xCCLs in detail, enabling the study of xCCL performance and discovery of high-performing parameter configurations with substantially reduced costs compared to state-of-the-art approaches. We compare to white box and black box approaches both qualitatively and quantitatively.
- ② We implement CCLINSIGHT, a tool guided by our primitive-centric gray box analysis methodology, which we use to test and support our hypothesis. To the best of our knowledge, this work is the first study of xCCLs using a primitive-centric approach.
- ③ We discover an important new scaling law using CCLINSIGHT, which we label the **collapsed optimal configuration scaling law**.

## 2 Background and Related Work

### 2.1 Accelerator-based CCLs

NCCL [5] orchestrates collective communications across NVIDIA GPUs using core primitives such as point-to-point communications, reduction operations, and memory transfers [47]. MSCCL [16, 18, 19, 24, 43] extends this primitive-based design by allowing users to develop custom collective communication algorithms through a

data-oriented domain-specific language (DSL). The MSCCL compiler generates an intermediate representation (IR) from DSL code, which is then executed within the MSCCL runtime. RCCL [12] builds upon NCCL’s architecture, extending support to AMD GPUs while integrating primitive designs from both NCCL and MSCCL.

### 2.2 Related Work

Although xCCLs are essential for distributed AI workloads, profiling and analyzing them at large scale is challenging due to their algorithmic complexity, high resource demands, and use of multiple types of hardware. Common issues include overwhelming trace-file I/O, out-of-memory errors, hangs, and other runtime failures. For example, Google Cloud suggests limiting NCCL debug logs on clusters of 64 nodes or more to prevent excessive log output [8]. In addition, using NVIDIA Nsight Systems and Nsight Compute can add a few microseconds of overhead to each trace event, which quickly adds up when profiling many small-data operations in large-scale collectives [7].

Existing xCCL analysis methods can be classified into two categories: black box and white box (left and middle columns of Fig. 1, respectively). The black-box approach treats xCCLs as opaque systems, offering *method generality*—applicable across contexts with minimal expertise [29, 30, 33, 41]. Due to their ease of use, they are widely adopted [13, 38, 42, 44]. However, this method often rely on exhaustive experiments across the full configuration space, yielding context-specific results, limited insight into xCCL internals, and high time and resource costs. AFNFA [46] reduces cost by randomly sampling 5% of the parameter space and using NCCL-test to identify high-performance configurations; we compare against this state-of-the-art black-box approach in §8.

White-box approach provides deeper insights into xCCL performance by leveraging domain knowledge for tuning and optimization [25, 28, 33]. However, these methods rely on specific workload or hardware assumptions [20, 26, 27, 34, 36, 39], limiting both *method* and *insight generality*. For example, Kim et al. [27] use this approach for PCIe-based intra-node GPU optimizations, and the work “Exploring GPU-to-GPU Communication” (EGG) from De Sensi *et al.* achieve up to 6× speedup in AllReduce by tuning environment variables and analyzing their impact [21]. We also benchmark against this white-box approach in §8.

## 3 Motivation and Methodology

In this section, first we explain the key ideas and preliminaries of CCLINSIGHT, and then we will introduce the methodology of CCLINSIGHT.

### 3.1 Key Ideas Behind CCLINSIGHT

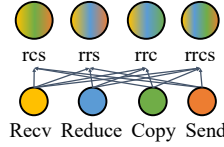
To explain why we choose to profile primitives in xCCLs, we must consider the most efficient way to analyze collective communication. xCCLs can be analyzed from multiple perspectives: the *GPU architecture level* (what low-level operations is the GPU performing?), the *network level* (from the perspective of the NIC and interconnect), and the *communication algorithm level* (which ranks communicate with which others, and when?). However, a closer examination of xCCLs reveals an underlying intuition: all of these

perspectives ultimately rely on a common foundation—the **primitive communication operations**. Architecture influences the performance of primitives, and the combination of primitive performance, types, and usage frequency within an algorithm offers a comprehensive view of collective communication performance. An additional advantage of primitive-level profiling is that the set of primitive types is both fixed and limited, which significantly reduces the profiling space compared to the vast variety of algorithmic and architectural variants. Based on this intuition, our central **hypothesis** is that primitive-level analysis provides an efficient (i.e., far lower overhead than prior methods) and effective (i.e., good-enough to capture the key performance factors) methodology for studying xCCLs.

### 3.2 Preliminaries of CCLINSIGHT

Before presenting our methodology, we first clarify primitives and explain how they expose the impacts of hardware architecture, algorithm design, and xCCL implementation.

We classify xCCL primitives into two categories: **multi-operation primitives** and **fundamental primitives**. **Fundamental primitives** consist of single logical operations, such as Send, Recv, Copy, Reduce, Scatter, and Gather. In contrast, **multi-operation primitives** are more complex and can be decomposed into fundamental primitives, as shown in Fig. 2. Examples include RecvCopySend (rcs), RecvReduceCopy (rrc), RecvReduceSend (rrs), and RecvReduceCopySend (rrcs).

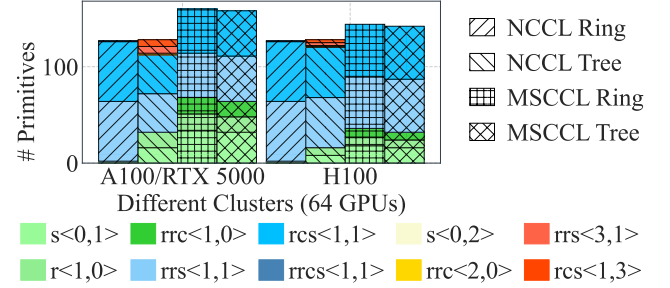


**Figure 2: Breakdown of Multi-Operation Primitives into Fundamental Primitives**

Although the set of primitive types is fixed, different architectures, algorithms, and xCCL implementations employ distinct combinations of these primitives. Fig. 3 highlights three key observations:

- (1) **Architecture impact.** The A100/RTX 5000 cluster has four GPUs per node (16 nodes total), whereas the H100 cluster has eight GPUs per node (8 nodes total). Although both experiments use 64 GPUs, the A100/RTX 5000 setup involves more inter-node communication. Consequently, the counts of  $s\langle 0, 1 \rangle$  and  $r\langle 1, 0 \rangle$  primitives in NCCL-Tree, MSCCL-Ring, and MSCCL-Tree are lower on the A100/RTX 5000 cluster than on the H100 cluster.
- (2) **Algorithm impact.** Comparing NCCL-Ring with NCCL-Tree shows that Ring rarely employs primitives such as  $s\langle 0, 1 \rangle$ ,  $r\langle 1, 0 \rangle$ ,  $rrs\langle 3, 1 \rangle$ , and  $rcs\langle 1, 3 \rangle$ , illustrating how algorithm choice directly shapes the primitive mix.
- (3) **Implementation effect.** Even with the same algorithm and architecture, different xCCLs may produce different primitives. For example, MSCCL-Tree lacks  $rrs\langle 3, 1 \rangle$  and  $rcs\langle 1, 3 \rangle$  because MSCCL primitives do not support sending to or receiving from multiple peers simultaneously, unlike NCCL-Tree.

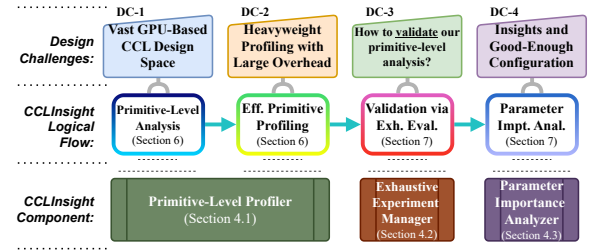
These differences demonstrate how architecture, algorithm, and xCCL implementation each leave a distinct “fingerprint” at the primitive level. Consequently, primitive-level profiling can capture these differences—at least to a meaningful extent.



**Figure 3: Numbers of primitives used by NCCL and by MSCCL (Ring / Tree algorithms). Each bar reports the count of every primitive type; the tuple  $\langle n, m \rangle$  indicates the number of peers sending ( $n$ ) and receiving ( $m$ ) data for that primitive.**

### 3.3 Our Methodology

Our methodology is to enable rapid analysis and optimization of xCCLs, we focus on identifying the factors that influence performance. In particular, we analyze parameters—the configurable components of xCCLs—as they are key contributors to performance variation. We contend that an ideal characterization and analysis framework for xCCLs should address the guiding questions outlined in Fig. 4.



**Figure 4: The challenges and methods of CCLINSIGHT, including how they each relate to one another. Associated sections in our paper are indicated within each box.**

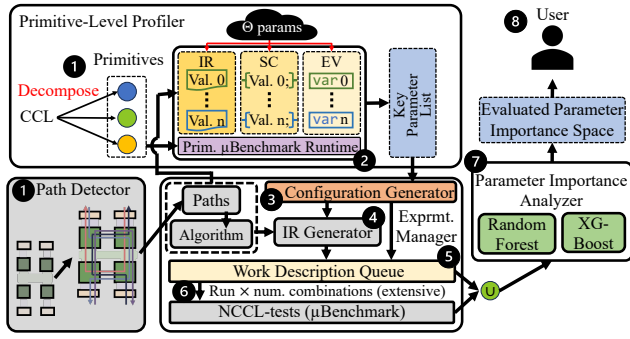
- ① Multiple collective operations, algorithms, and parameters—along with their interactions—create an extremely large configuration space. To reduce this complexity, we first conduct a primitive-level analysis by isolating and testing individual parameters. This approach reduces the search space by avoiding parameter interactions during initial evaluation. While this may not fully capture interactions, we compensate by later performing exhaustive exploration on the most important parameters identified. This yields a “good enough” understanding of interactions while significantly reducing the analysis cost (C1 in Fig. 4).
- ② Existing profiling tools such as NVIDIA Nsight [35] introduce non-negligible overhead. To address this, we design a lightweight profiling tool using GPU shared memory to minimize performance impact. This tool efficiently identifies the most important parameters (C2 in Fig. 4).
- ③ To validate our primitive-level analysis, we examine the values of all identified important parameters by automatically running collective-level experiments (C3 in Fig. 4). This targeted approach is faster than other methods because it only explores a narrowed parameter space.

4 Finally, CCLINSIGHT provides insights into parameter impacts and generates near-optimal configurations of GPU-based collective communication, while remaining lightweight (C4 in Fig. 4).

## 4 Design of CCLINSIGHT

### 4.1 Primitive-Level Profiling

Once a CCL determines the hardware topology and collective algorithm, our **primitive-level profiling** approach employs a customized, lightweight microbenchmark integrated with a co-designed, statically instrumented version of the CCL to collect detailed execution metrics for the involved primitives. These metrics include timestamps, durations, orders, and occurrence counts. Our profiling approach is designed to introduce minimal overhead by strategically leveraging low-latency GPU shared memory.<sup>2</sup> Com-



**Figure 5: Constituent components of CCLINSIGHT.** Gray background boxes represent prior work or off-the-shelf tools, while colored background boxes indicate new contributions. Here, *IR* denotes parameter configurations that modify an MSCCL intermediate representation, *SC* refers to configurations involving modifications and recompilation of CCL source code, and *EV* represents configurations set through pre-run environment variables.

pared to the most frequently used profiling tools such as NVIDIA’s Nsight Compute and Nsight Systems which have been observed to cause  $\geq 5\times$  and  $1.5\text{--}5\times$  overhead, respectively [1], CCLINSIGHT has significantly lower overhead of  $\approx 1\times$ .

The detailed steps of the primitive profiling routine are presented in Alg. 1. The user starts by populating  $\Theta$  with the parameters under consideration, as well as the acceptable ranges for each parameter by adding them to a configuration file. The GETPARAMVALS routine acts as an iterator over the acceptable values for a given parameter  $\theta \in \Theta$ . For example, consider the “Chunk Size” parameter. The GETPARAMVALS routine would yield a new acceptable chunk size each time it is called until the acceptable values are exhausted. In the case of a parameter with continuous values, the GETPARAMVALS routine performs uniform discretization over the acceptable range using a user-specified number of sampling points. Alg. 1 then determines how the new “Chunk Size” will be updated. Depending on the parameter type, one of three approaches is applied: a) **for IR**, the algorithm generates a new intermediate representation (IR) configuration using LOWERING and benchmarks it with

<sup>2</sup>In our primitive-level experiments, CCLINSIGHT uses 5,360 bytes of shared memory per thread block when storing 64-bit timestamps. The available shared memory per SM is 228 KB on the NVIDIA H100 [3] and 164 KB on the A100 [2].

BENCHMARKIR; b) **for SC**, the algorithm updates the source code by applying a patch in which template values are substituted based on the current parameter set  $x$  using UPDATECCLSOURCE and recompiles with RECOMPILECCL, followed by benchmarking with BENCHMARKSC; c) **for EV**, it configures and benchmarks the parameter using BENCHMARKENV. Referencing Tab. 1, we can see that the parameter type of “Chunk Size” is SC, and thus the CCL source code will be patched and recompiled.

These flexible mechanisms ensure comprehensive coverage of parameter types, capturing their unique impacts on performance. Performance metrics are stored in a results matrix that maps primitives and parameters to their corresponding benchmark results. The algorithm then analyzes these results using REDUCEON, computing the standard deviation of the impact of each parameter on performance as a measure of importance. Alg. 1 produces “key parameters list”  $\mathcal{L}$  containing  $k$  different parameters that, through primitive-centric analysis, it has determined to be key. This is performed by computing the standard deviation in performance across all tested values, then selecting the top  $k$ . In addition to informing  $\mathcal{L}$ , the results matrix  $m$  and parameter impact hypothesis  $I$  can be reviewed by an expert to derive insights, such as those shown in §6.2, 6.3, and 6.4. As this last step involves manual review, we term our primitive-level profiling approach *semi-automated*.

For simplicity, Alg. 1 omits certain implementation details regarding the evaluation of  $\text{Is}(\text{IR}|\text{SC}|\text{EV})\text{Tweak}$ . Specifically, our tool internally maintains the previously tested configuration, allowing it to detect when a parameter modification—which we refer to as a “tweak”—occurs. This detection can be accomplished through a straightforward comparison when a new parameter configuration is loaded. Lastly, note that an explicit algorithm for GETPARAMVALS is not provided here. This omission is intentional as the routine itself is trivial to reconstruct, given that parameter ranges are predefined (see Tab. 1).

#### Algorithm 1 Primitive-Level Profiling Routine

**Given**  $\mathcal{P}$  is the set of primitives for all CCLs under consideration  
**Given**  $\Theta$  is the set of all parameters  
**Given**  $\mathcal{H}$  is the set of paths from path detection (constant for a given system)  
**Given**  $a$  is the algorithm under consideration

```

 $m \leftarrow \emptyset$  ▷ Matrix to store results by primitive and parameter
for  $(p, \theta) \in \mathcal{P} \times \Theta$  do ▷ Iterate over parameter values for  $\theta$ 
  for  $x \in \text{GETPARAMVALS}(\theta)$  do
    if  $\text{IsIRTWEAK}(\theta)$  then ▷ Generate IR using algorithm and paths
       $z \leftarrow \text{LOWERING}(a, \mathcal{H}, x)$ 
       $r \leftarrow \text{BENCHMARKIR}(a, p, z)$ 
    else if  $\text{IsSCTWEAK}(\theta)$  then
       $\text{UPDATECCLSOURCE}(x)$ 
       $\text{RECOMPILECCL}()$ 
       $r \leftarrow \text{BENCHMARKSC}(a, p)$ 
    else ▷ Assume environment variable tweak by default
       $e \leftarrow \text{CREATEENV}(x)$ 
       $r \leftarrow \text{BENCHMARKENV}(p, a, e)$ 
    end if
     $m \leftarrow \text{INSERT}(m, \{\text{Key}(p), \text{Key}(\theta) : \text{Value}(r)\})$ 
  end for
end for
 $I \leftarrow \text{REDUCEON}(m, \text{"parameter"}, \text{std})$ 
 $\mathcal{L} \leftarrow \text{SELECTTOP}(I, 2)$  ▷ Key Parameter List. We choose  $k = 2$  here.

```

## 4.2 Experiment Manager

CCLINSIGHT includes an experiment manager that systematically explores the key parameter space by running the collective-communication microbenchmark. It computes the Cartesian product of all combinations of values in the key parameter list.

The manager’s workflow starts with a configuration generator that produces work units from the key parameter list, e.g., channel count, chunk size, and tile count. These units are enqueued for benchmarking. Next, each unit is dequeued, and its configurations drive the execution of the corresponding microbenchmark. For relevant configurations, a code patch is applied or IR is generated using the MSCCL compiler. Building on NCCL-tests [9], the tool automates performance assessments by benchmarking MSCCL across multiple IR-generated configurations and NCCL. Each experiment is repeated multiple times in order to mitigate the impact of noise. After each experiment is completed, the results are combined with the work description data and stored in a database  $\mathcal{D}$  for later analysis. The values in  $\mathcal{D}$  are performance results for a given key, while the key is defined as a tuple containing algorithm  $a$ , a GPU scale  $s$ , and a message size  $m$ .

## 4.3 Parameter Importance Analyzer

Following the experiments, we analyze the collected data to evaluate each **parameter’s importance** in relation to collective latency. This analysis employs a random forest (RF) [14] regressor and an XGBoost [17] regressor to measure the parameter importance across various GPU scales and message sizes. Parameter importance quantifies the relative contribution of each parameter to the model’s prediction of overall collective performance.

We use parameter importance as a proxy for true parameter significance, as the parameter features are largely uncorrelated and do not have high cardinalities. The XGBoost models achieve a mean percent error of 2.6%, with the highest percent error at 5.9% across all models. In comparison, the RF models yield a mean percent error of 34.4%. Despite these differences, the parameter importance patterns identified by both models are nearly identical, reinforcing the reliability of our analysis.

The feature importance analysis routine is detailed in Alg. 2. Starting with the results of the experiments  $\mathcal{D}$ , the importance analyzer routine iterates through each unique algorithm-GPU scale-message size configuration and fits an RF and XGBoost model  $m$  to the associated samples  $\mathcal{W} \mapsto \mathcal{R}$  (retrieved via the GETALL call). The trained model is then used to compute importance values for each parameter. These parameter importance values, together with their corresponding models, are stored in a new database  $\mathcal{V}$  to facilitate future analysis.

## 5 Experimental Environment

We evaluated xCCL performance across clusters with diverse hardware configurations. This study conducts experiments on three clusters, providing a robust platform for performance analysis.

**Hopper H100 Cluster:** We utilized one Amazon Web Services (AWS)-based cluster with NVIDIA Hopper H100 GPUs. Each node contains 8 H100 GPUs (80 GiB HBM3 each) connected via 18 NVLink 4.0 links per GPU (900 GB/s). The inter-node communication uses 32 RDMA-capable Elastic Fabric Adapters (EFAs) per node, delivering

### Algorithm 2 Importance Analyzer Routine

---

**Given**  $\mathcal{D}$  is the experiments output DB from §4.2

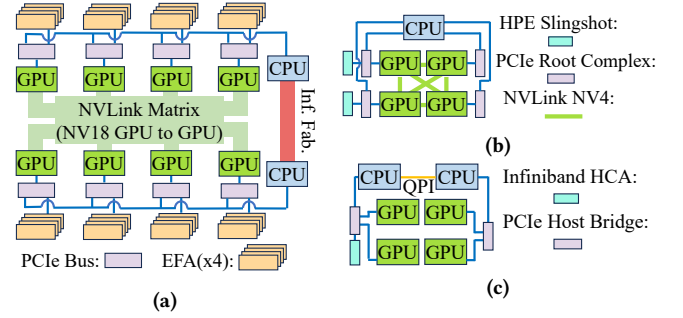
```

 $\mathcal{V} \leftarrow \emptyset$  ▷ Database to hold per-parameter importance values
for  $(a, s, m) \in \text{Keys}(\mathcal{D})$  do
   $\mathcal{W} \leftarrow \emptyset$  ▷ To hold set of parameter values
   $\mathcal{R} \leftarrow \emptyset$  ▷ To hold set of results
   $(\mathcal{W}, \mathcal{R}) \leftarrow \text{GETALL}(\mathcal{D}, (a, s, m))$  where  $\mathcal{W} \mapsto \mathcal{R}$ 
   $m \leftarrow \text{FITMODEL}(\mathcal{W}, \mathcal{R})$ 
   $\mathbf{i} \leftarrow \text{FEATUREIMPORTANCES}(m)$  ▷  $\mathbf{i}$  is a vector of importances
   $\mathcal{V} \leftarrow \text{INSERT}(\mathcal{V}, \{\text{Key}(a), \text{Key}(s), \text{Key}(m) : \text{Value}(\mathbf{i})\})$ 
end for
return  $\mathcal{V}$ 

```

---

3200 Gbps. The CPUs are dual AMD EPYC Milan 7R13 processors (96 physical cores, 2 TiB RAM). The environment was configured with the NVIDIA driver 550.54.15, CUDA 12.4, NCCL v2.20.5-1, MSCCL 2.8.4-1, and the AWS-OFI-NCCL plugin v1.8.1-aws. We deployed 64 GPUs across 8 nodes for our experiments.



**Figure 6: Topology of (a) the Hopper H100 cluster, (b) Ampere A100 cluster, (c) RTX 5000 cluster.**

**Ampere A100 Cluster:** For large-scale experiments, we utilize an A100 cluster. Each node features 4 NVIDIA A100 GPUs (40 GiB HBM2), connected via PCIe 4.0, with 4 NVLink 3.0 links per GPU (600 GB/s bandwidth). The nodes are interconnected through an HPE Slingshot 200 Gbps RDMA-capable network, organized in an 11-group dragonfly topology. Each node is powered by an AMD EPYC Milan 7543P processor (32 cores, 512 GiB RAM). We used 256 GPUs across 64 nodes for our experiments.

**RTX 5000 Cluster:** The third cluster consists of nodes with 4 NVIDIA Quadro RTX 5000 GPUs (PCIe 3.0, 16 GiB each), connected to Intel Xeon E5-2620 v4 processors (126 GiB RAM) via Intel Quick-Path Interconnect (QPI). Intra-node communication relies on PCIe as these GPUs lack NVLink. A 56 Gbps InfiniBand fat-tree network handles inter-node communication. We used 64 GPUs on 16 nodes.

**Model Information:** For each combination of  $\langle \text{GPU Scale, Message Size, Parameter Configuration} \rangle$ , between 36 and 90 experiments were conducted. The geometric mean of random forest model percent error across configurations was approximately 0.1% for the A100 cluster and 0.01% for both the H100 and RTX 5000 clusters, with XGBoost yielding similar results. We use default scikit-learn and XGBoost hyperparameters [4, 37].

## 6 Primitive-Level Analysis of CCLs

This section presents our analysis using a customized microbenchmark in CCLINSIGHT to evaluate the impact of parameters across

three types of configurations (*IR*, *SC*, and *EV*). To gain deeper insights, we perform a detailed, primitive-level examination to uncover the underlying factors driving each parameter's influence. This paper focuses on AllReduce, but our approach is also **applicable to other collective operations**, as discussed in §8.1.3.

## 6.1 Choosing Important Parameters for CCLs

In our microbenchmark, we categorize parameter configurations into three types based on their source. In the following subsections, we provide a case study on each category. For *EV*, we select the number of channels because its impact is closely tied to the physical architecture of the system, particularly its network configuration. For *SC*, we chose chunk size as a parameter due to its relevance to algorithm design and therefore the potential to provide valuable insights into algorithmic performance. For *IR*, we included the number of tiles, as this parameter is unique to MSCCL. We note that the methods introduced here can be easily adapted to evaluate additional parameters, such as those listed in Tab. 1, and those detailed in CCL documentation<sup>3</sup>.

**Table 1: List of Tuning Parameters in GPU-based CCLs. Those with a † are chosen for analysis in this work.**

Parameter	Default	Value Range	Source
chunk size †	128 KiB	64 : 256 : *2	<i>SC</i>
# channels †	1	1 : 32	<i>EV</i>
# tiles †	1	1 : 16	<i>IR</i>
# instances	1	$\mathbb{Z}^+$	<i>IR</i>
SOCKET_NTHREADS (SNT)	1	1 : 16	<i>EV</i>
NSOCKS_PERTHREAD	1	1 : 64 / SNT	<i>EV</i>
BUFFSIZE	4 MiB	$\mathbb{Z}^+$	<i>EV</i>
NTHREADS (NT)	512	64 : 512 : *2	<i>EV</i>
nthreadsSplit	358	32 : NT : +32	<i>SC</i>
IB_QPS_PER_CONNECTION	1	1 : 128	<i>EV</i>

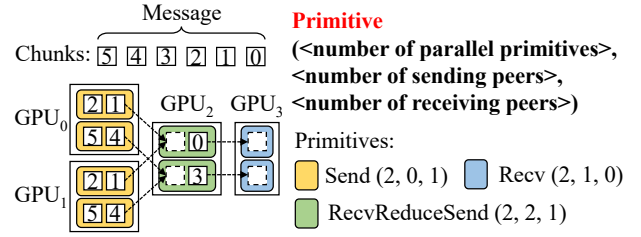
## 6.2 Case Study 1: Chunk Size

NCCL and MSCCL utilize a pipelined design to enhance performance. As shown in Fig. 7, NCCL's primitive divides the message into multiple chunks, with primitives processing each chunk sequentially. For instance, while the send primitive in GPU<sub>0</sub> sends chunk 1 to GPU<sub>2</sub>, the rrs primitive in GPU<sub>2</sub> simultaneously forwards the previously received chunk 0 to GPU<sub>3</sub>.

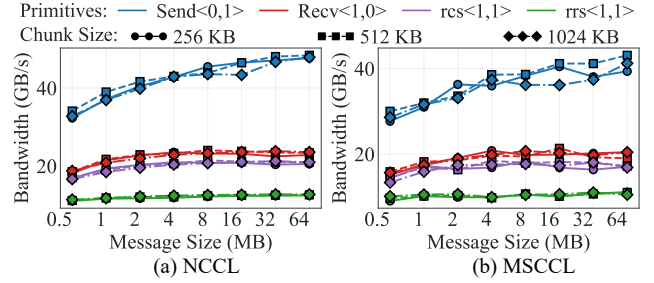
In our analysis, we define **chunk size** as the maximum size of each chunk, which segments the original message to facilitate communication pipelining across GPUs. In Fig. 8, adjusting the chunk size in the primitive microbenchmark runtime reveals that chunk size affects communication throughput for specific primitives, such as the send primitive. The difference in performance between the best and worst chunk sizes across various message sizes can result in up to a 7.24% improvement.

To further explore the impact of chunk size on collective communication, we analyze its role in different *algorithms*.

<sup>3</sup>NCCL documentation lists its available environment variables and value ranges at [docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html](https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html)



**Figure 7: Primitives in NCCL**



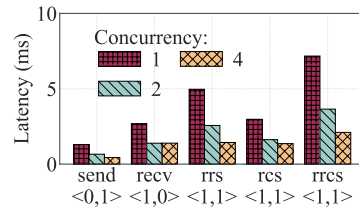
**Figure 8: Chunk Size Impact on Primitives<#receive peers>, #send peers> in A100 Cluster**

In the tree algorithm, the number of chunks is defined as  $\max\left(\frac{\text{message size}}{\text{chunk size}}, 1\right)$ , while in the ring algorithm, the **real chunk size** is  $\min\left(\frac{\text{message size}}{\text{number of GPUs}}, \text{chunk size}\right)$ . This means that in the ring algorithm, chunk size dynamically adjusts with message size and GPU count, making its importance vary across GPU scales.

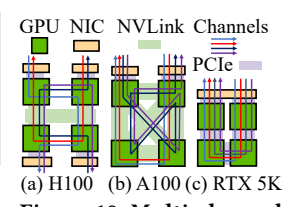
**Observation 1:** Chunk size is more important in the ring algorithm than in the tree algorithm, with varying importance across GPU scales.

## 6.3 Case Study 2: Number of Channels

Fig. 7 illustrates two concurrent primitives per GPU, where the message is evenly split between them. In Fig. 9, we demonstrate the impact of concurrency on primitives runtime. We found that increasing the number of concurrent primitives can significantly reduce the latency of the total data transfer workload.



**Figure 9: Concurrency Impact on NCCL Primitives in A100 Cluster**



**Figure 10: Multi-channel configs across clusters.**

In both NCCL and MSCCL, each channel is associated with a single thread block, making the channels concurrent. Each thread block is responsible for processing its assigned data block using the primitives, meaning that primitives across different channels run concurrently. We define the number of channels, often denoted as '# Channels', as the total number of concurrent communication channels used.

Building on the impact of channels at the primitive level, we illustrate how channel importance is influenced by the physical network topology. In Fig. 10a, which displays a rough analogue of the H100 cluster's topology, each GPU is directly connected to 4 EFA NICs. With 4 channels, there is only one channel in each communication direction of each NIC. In Fig. 10b, we explain the implementation of 4 channels in an A100 cluster node (Fig. 6b) where there are two NICs for each node, two channels share each direction of each NIC. For the RTX 5000 cluster node architecture (Fig. 10c) where each node has only one NIC, 4 channels share the same each direction of each NIC.

Due to the concurrency between channels and the ability for different channels to utilize separate NICs, the number of channels is crucial for fully leveraging all available network interfaces on a node. The impact of the number of channels becomes even more significant with larger message sizes, as smaller messages may not saturate the NICs, even when few NICs are used.

**Observation 2:** When the number of channels is fewer than the number of NICs per node, the number of channels becomes critically important, especially for larger message sizes.

#### 6.4 Case Study 3: Number of Tiles

The multi-peer primitives in NCCL support communication with multiple peers using a single primitive. As shown in Fig. 7, the RecvReduceSend primitive can communicate with both GPU<sub>0</sub> and GPU<sub>1</sub> simultaneously. However, MSCCL primitives can only communicate with up to one peer in each direction. When a GPU in MSCCL needs to communicate with multiple sending peers, as shown in Fig. 11, GPU<sub>2</sub> needs to receive data from GPU<sub>0</sub> and GPU<sub>1</sub>. It first uses one rrc primitive to receive data from GPU<sub>0</sub>, then uses another rrc primitive to receive data from GPU<sub>1</sub>, and finally sends the data using a send primitive to GPU<sub>3</sub>. In situations where multiple single-peer primitives with dependent relationships are used to communicate with multiple peers on a single GPU, we set the dependency metric in the primitive parameters to 1; otherwise, it is set to 0.

**Primitive** (<number of parallel primitives>, <number of sending peers>, <number of receiving peers>, <number of tiles>, <dependency?>)

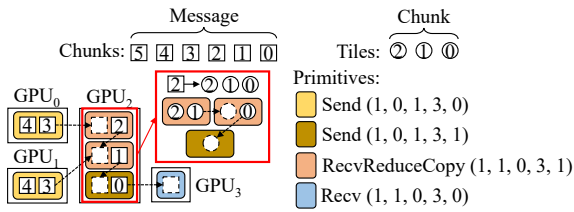


Figure 11: Chunk Tiling in MSCCL Primitives

The number of tiles, divided from a chunk, is used to optimize the communication pipeline across primitives. We frequently stylize this as '# Tiles'. MSCCL splits the chunk into multiple tiles, introducing a finer-grained pipeline between primitives within the same GPU. This is illustrated for GPU<sub>2</sub> in Fig. 11, where the two rrc and one send primitives form a pipeline.

In Fig. 12a, we illustrate the performance differences between NCCL's multi-peer and MSCCL's single-peer primitives. While the

combination of single-peer primitives can achieve the same communication task as multi-peer primitives, the absence of chunk tiling results in the latency of the combination of single-peer primitives ( $2 \cdot \text{send} + \text{recv}$ ) being  $2.52\times$  higher compared to the multi-peer primitive (rrs) when the message size is 32 MB. To evaluate the impact of chunk tiling on pipeline efficiency within a single GPU for multiple single-peer primitives, we used the microbenchmark runtime with different IR, varying the number of tiles across different message sizes. As shown in Fig. 12b, we observe an improvement of up to  $2.1\times$  between the optimal number of tiles and the least efficient configuration when the message size remains constant. This highlights the importance of selecting an appropriate number of tiles to enhance concurrent communication with multiple peers.

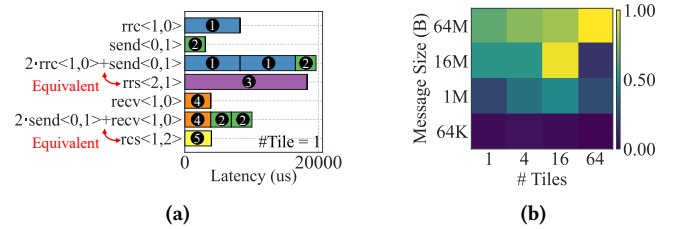


Figure 12: (a) Performance comparison between multi-peer and single-peer primitives for a message size of 64 KB and #Tiles = 1; (b) Heatmap showing the relative bandwidth of  $\frac{2 \cdot \text{rrc} \langle 1, 0 \rangle + \text{rrs} \langle 1, 1 \rangle}{\text{rrs} \langle 3, 1 \rangle}$  across different #Tiles and message sizes.

From our analysis of the MSCCL source code and the tree algorithm, we discovered that the tree algorithm in MSCCL depends heavily on numerous single-peer primitives with dependent relationships to complete multi-peer communication tasks—a characteristic absent in the ring algorithm.

**Observation 3:** The tile number parameter is more crucial in the tree algorithm than in the ring algorithm.

#### 7 Validation of the Analysis

In §6, we present our key observations based on microbenchmark experiments. In this section, we validate these observations through extensive testing using the experiment manager and analyze the collected data with the parameter importance analyzer.

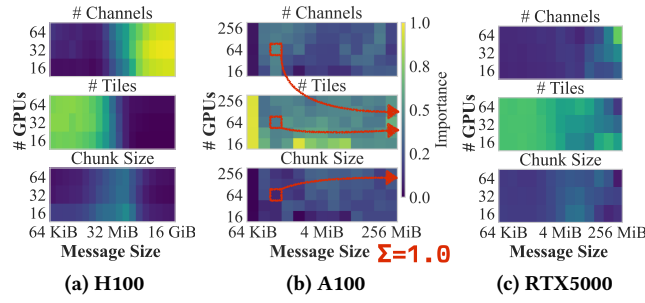
##### 7.1 Relative Parameter Importance to Latency

In this section, we evaluate the importance of each parameter in determining the overall collective latency using two case studies: the double binary tree algorithm [6] (Fig. 13) and the ring algorithm (Fig. 14). To perform this evaluation, we explore the entire parameter space and utilize a random forest regressor<sup>4</sup>, taking parameter configuration details as input features and the collective latency as the target output.

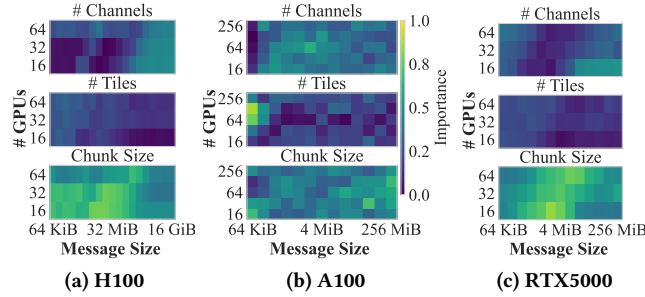
Each pixel in the figures represents a particular *parameter importance*<sup>5</sup> derived from the model for a corresponding GPU scale

<sup>4</sup>We chose not to present the XGBoost results here for reasons noted in §4.3, namely the fact that RF and XGBoost have nearly identical results.

<sup>5</sup>Our definition of "parameter importance" here corresponds exactly to the concept of "feature importance" commonly used in the Random Forest and XGBoost literature. However, we prefer the term parameter importance, as it better reflects the context of our specific application.



**Figure 13: Parameter importance to collective latency across message size-GPU scale combinations. (Double Binary Tree)** The charts for each parameter should be read by comparing values at identical pixel coordinates (visualized in red).



**Figure 14: Parameter importance to collective latency across message size-GPU scale combinations. (Ring)**

and message size ('coordinate'). Parameter importance is a relative scale, and the total parameter importance from all parameters at a given coordinate must always sum to 1.0. Intuitively, if the outcome is determined solely by a single parameter, with no influence from any others, that parameter would be assigned the maximum possible importance value of 1.0 and all others 0. In our experiments, we observe that the parameter importance values change with the GPU scale and message size. For example, in Fig. 13b, we observe that the boxed parameter importance values for the '# Channels' and '# Tiles' are near 0.5 and 0.4, respectively, and that the 'Chunk Size' is relatively unimportant at only approximately 0.1, which is not necessarily the case at other message sizes-GPU coordinates.

## 7.2 Validation of Observations

**7.2.1 Validation of the Chunk Size Observation.** Chunk size is more important in the ring algorithm (Fig. 14) across clusters than in the double binary tree algorithm (Fig. 13), validating observation 1. Moreover, an interesting diagonal pattern emerges in Fig. 14: the importance follows a diagonal pattern as both scale and message size increase. Overall, chunk size is the most important parameter in Ring, surpassing both channel count and tile size.

**7.2.2 Validation of the Number of Channels Observation.** We find that the most important parameter for large message sizes in the H100 cluster is the number of channels, as shown in Fig. 13a, compared to the A100 cluster (Fig. 13b) and the RTX 5000 cluster (Fig. 13c). This is due to the higher number of NICs. As shown

in Fig. 6a, each node in the H100 cluster has 32 NICs, compared to just 2 in the A100 cluster (Fig. 6b) and 1 in the RTX 5000 cluster (Fig. 6c), validating observation 2.

In our experiments, the minimum number of channels used was 2 for both the A100 and RTX 5000 clusters, indicating that the NICs were fully utilized even with the minimum number of channels. However, in the H100 cluster, the number of channels varied from 8 to 32, resulting in underutilized NICs when the number of channels was small. This factor becomes more critical with large message sizes.

**7.2.3 Validation of the Number of Tiles Observation.** As depicted in Fig. 13, tile number emerges as the most important parameter in most cases when using tree algorithm. This is due to the limitations of single-peer primitives with dependent relationships to complete multi-peer communication tasks, as illustrated in Fig. 12a, where single-peer primitive performance lags behind. Thus, using tile-based pipeline across thread blocks within a GPU is crucial to improve the efficiency of single-peer primitives for tree algorithm, while in Fig. 14, with the absence of the single-peer primitives with dependent relationships, number of tiles is not important in ring algorithm, validating observation 3.

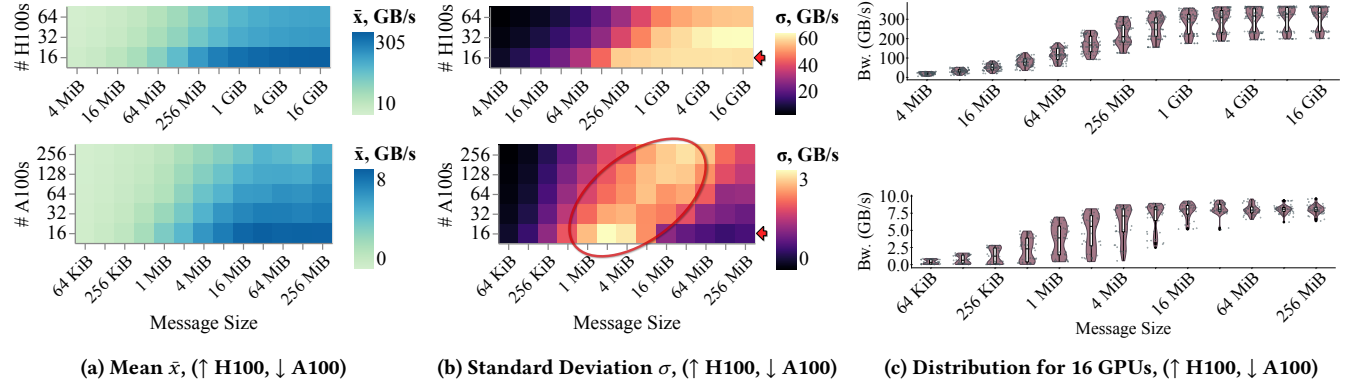
## 7.3 Takeaways

**7.3.1 Configuration Impact Until BW Saturation.** Fig. 15 presents the mean and the standard deviation of the collective bus bandwidth (BW)<sup>6</sup>, averaged across all measured parameter configurations and experiment iterations. The mean collective bus bandwidth, shown in Fig. 15a, varies significantly based on message size and the number of GPUs. An interesting pattern appears in the standard deviation (Fig. 15b): the largest deviation occurs with large message sizes in the H100 cluster, while it occurs with medium message sizes in the A100 cluster. Combinations of large message sizes and small GPU scales exhibit a drop in variance, resulting in a "diagonal" hot streak across the space in the A100 cluster. At the corresponding point in Fig. 15a, bus bandwidth approaches saturation at around 8 GB/s. This is further illustrated in Fig. 15c, where the saturation point and reduced variance are highlighted. When bandwidth saturation occurs, the parameter configuration has less impact on performance, reducing the gap between optimal and suboptimal configurations. This behavior is not observed in the H100 cluster because certain parameter configurations cannot saturate the bandwidth, even when the message size is sufficiently large.

**Takeaway 1:** Parameter configurations impact performance until bandwidth saturation, beyond which their effect diminishes.

**7.3.2 Configuration Impact Across Message Sizes.** We observe that the performance distribution across all parameter configurations is multi-modal for large message sizes. As shown in Fig. 15c, the number of channels is the most important parameter in the H100 cluster when using the tree algorithm for large message sizes. The group of suboptimal parameter configurations arises when the number of channels is smaller than the number of NICs in the H100 system, as discussed in §7.2.2. In contrast, in the A100 cluster

<sup>6</sup>Further information about bus bandwidth in the context of GPU-accelerated communication libraries is available at [github.com/NVIDIA/ncccl-tests/blob/master/doc/PERFORMANCE.md](https://github.com/NVIDIA/ncccl-tests/blob/master/doc/PERFORMANCE.md)



**Figure 15: Comparison of the mean, standard deviation, and distribution of bus bandwidth across different message size-GPU configurations using the Double Binary Tree algorithm.** Top row corresponds to H100s, bottom row corresponds to A100s. Each pixel in (a) and (b) represents a specific message size-GPU scale combination. Violin plots in (c) represent the distribution of bus bandwidths for the slice indicated with the red arrow across all parameter configurations and + marks represent individual samples.

(Fig. 15c) where the number of channels has less influence, all parameter configurations achieve peak bandwidth, resulting in only a small performance gap between them. This is also indicated in Fig. 13b.

**Takeaway 2:** As message size increases, the impact of parameter configuration on performance remains significant when the number of channels is a key factor.

## 8 Benefits Evaluation

### 8.1 Performance Benefits of CCLINSIGHT

To demonstrate the effectiveness of CCLINSIGHT, we implemented the same double binary tree algorithm used in NCCL, as well as the ring algorithm, in MSCCL. Using CCLINSIGHT, we identified optimal configurations and compared CCL performance with and without our optimizations in Fig. 16. The experiments were conducted with 64 GPUs in the H100 cluster, 256 GPUs in the A100 cluster, and 64 GPUs in the RTX 5000 cluster.

**8.1.1 NCCL vs. NCCL + CCLINSIGHT.** Although NCCL is already highly efficient, we found that CCLINSIGHT can still help enhance performance by identifying optimal parameter configurations. As shown in Fig. 16d and Fig. 16j, using the optimal number of channels identified by CCLINSIGHT—rather than the default settings—yielded an up to 43.71 % improvement in NCCL’s performance, especially for large message sizes. This aligns with our previous analysis. Additionally, in Fig. 16i and Fig. 16h, we observed up to 43.90 % improvement in performance when applying the optimal chunk size for small message sizes. However, no significant improvement was observed in Fig. 16f and Fig. 16e, as no parameter was identified as specifically important in this specific context.

**8.1.2 MSCCL vs. MSCCL + CCLINSIGHT.** For MSCCL, CCLINSIGHT plays a more critical role. Identifying the optimal number of tiles is particularly crucial for MSCCL’s performance. CCLINSIGHT enhanced the basic implementation in MSCCL, improving performance by up to 40.64%, 3.02%, and 2.81% on the H100, A100, and RTX 5000 clusters, respectively. The largest improvement stems

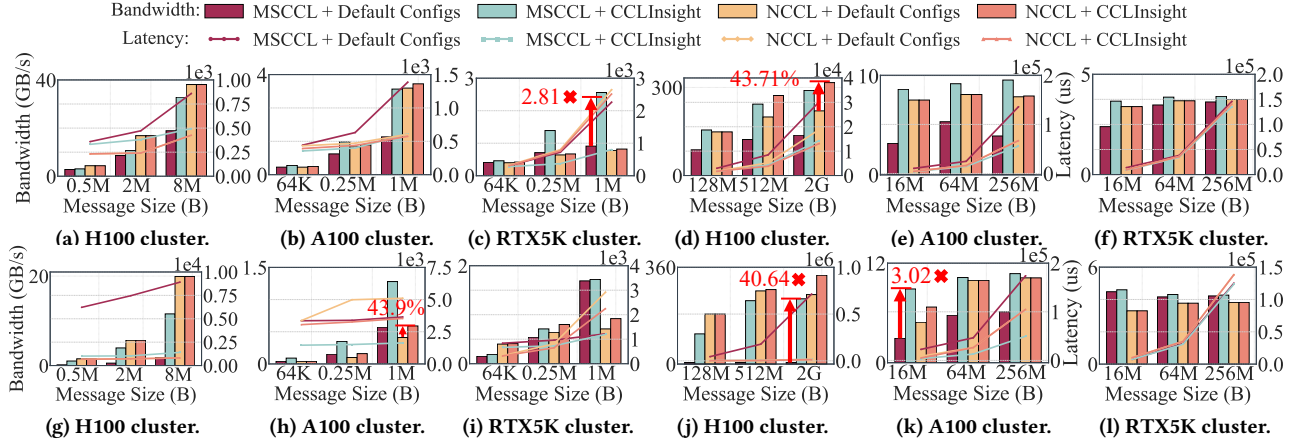
from the basic implementation’s failure to fully utilize the high bandwidth of the NICs in H100 cluster.

**8.1.3 LLM Application Performance Evaluation.** To evaluate CCL performance, we used CCLINSIGHT with three language model proxy workloads: GPT-3 175B [15], LLaMA2-7B [23, 45] and DeepSeek-R1 (LLaMA-70B) [22]. Our evaluation focuses on the AllReduce communication component, as it has been a primary focus of our previous discussions on collective communication. We conducted these evaluations on a 32-node H200 cluster (256 GPUs total, topology similar to Fig. 6a), each node offering 3200 Gbps of bandwidth. Fig. 17 shows performance for two algorithms—Ring (in NCCL) and Tree (in MSCCL). We use Tree in MSCCL because its multi-peer communication pattern matches MSCCL’s bottleneck (see §6.4), while NCCL relies on the Ring algorithm by default.

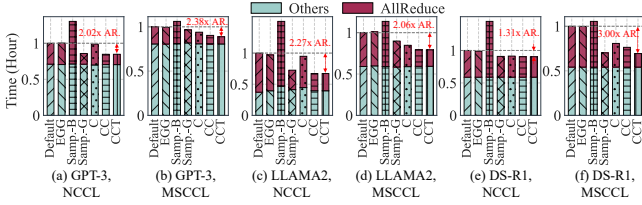
For a fair comparison, we normalize per-iteration runtimes so that the default configuration (using its iteration count as the baseline) corresponds to one hour; each other bar then shows how many hours it would take to complete that same number of iterations under the given configuration.

**CCLINSIGHT’s Benefits over Default Settings.** For the Ring algorithm in NCCL, we see that on both GPT-3 and LLaMA2—each of which runs AllReduce across 64 H200 GPUs—the largest gain comes from tuning the number of channels. With 3200 Gbps per node, increasing channel count is key to fully saturating the network. By jointly optimizing chunk size, channel count, CCLINSIGHT achieves up to 2.02× AllReduce speedup on GPT-3 and 2.27× on LLaMA2. On DeepSeek-R1 (32 GPUs AllReduce), where the default configuration is already close to optimal due to the smaller scale, CCLINSIGHT still delivers a 1.31× speedup.

For the Tree algorithm in MSCCL, the default settings are far from optimal. CCLINSIGHT uncovers and exploits the true performance drivers, yielding substantial improvements—up to 2.37× on GPT-3, 2.06× on LLaMA2, and 3.00× on DeepSeek-R1. The biggest improvement occurs in DeepSeek-R1 due to its small message size. As shown in Fig. 13, tile size greatly affects Tree performance in this case, so tuning it brings large gains.



**Figure 16: Performance comparison: MSCCL + Default configs vs. MSCCL + CCLINSIGHT and NCCL + Default configs vs. NCCL + CCLINSIGHT.** Subfigures (a)-(f) use the tree algorithm, while subfigures (g)-(l) use the ring algorithm. Subfigures (a)-(c) and (g)-(i) present results for small message sizes, whereas (d)-(f) and (j)-(l) show results for large message sizes.



**Figure 17: Time breakdown in distributed LLM training across different methods.** ‘Default’ uses the default settings. ‘C’, ‘CC’, and ‘CCT’ denote progressively optimized chunk size, channels, and tiles in CCLINSIGHT. ‘EGG’ applies the expert-tuned settings from [21]. ‘Samp-G’ and ‘Samp-B’ indicate the good (G) and bad (B) configurations via sampling (Samp) approaches, as discussed in AFNFA [46].

*Comparing CCLINSIGHT and EGG.* The EGG (white-box) method [21] achieves up to 6 $\times$  speedup on AllReduce when scaling from two nodes on LUMI (AMD MI250X GPUs with HPE Cray Slingshot interconnect) and Alps (NVIDIA H100 GPUs with HPE Cray Slingshot interconnect) [10, 11]. While this expert-driven tuning can be very effective given deep knowledge of the architecture and xCCL, its generality is limited—in our experiments, the EGG configurations failed to transfer effectively to the H200 cluster.

*Comparing CCLINSIGHT and AFNFA.* AFNFA, a black-box method, tunes parameters by randomly sampling a subset of the full configuration space and brute-forcing tests on those samples. We observe a large difference between its best and worst configurations. On NCCL AllReduce for GPT-3, AFNFA’s worst configuration runs 4.12 $\times$  slower than CCLINSIGHT, and on MSCCL AllReduce for DeepSeek-R1 it runs 3.37 $\times$  slower. Even against its own best configuration, CCLINSIGHT still outperforms AFNFA—achieving a 1.13 $\times$  speedup on LLAMA2 with NCCL and a 1.80 $\times$  speedup on GPT-3 with MSCCL.

*Discussion on Other Collective Operations.* Other collective operations, such as AllGather and Broadcast, use send, rcs, and recv, while ReduceScatter and Reduce also include rrs and rrc. As all these primitives were illustrated in AllReduce, our method supports all collectives, but due to page limits, detailed are omitted.

## 8.2 Cost Benefits of CCLINSIGHT

CCLINSIGHT is designed to minimize the resource and time overhead of analyzing xCCLs. In this section, we quantify the cost of each component of our approach and compare CCLINSIGHT with the two other methods: the sampling (black-box) approach and the expert-driven (white-box) analysis approach, in the context of the LLM application described in §8.1.3.

**8.2.1 AFNFA Cost.** AFNFA identifies a high-performing configuration by randomly sampling 5% of the full parameter space and evaluating each sample with the NCCL-test microbenchmark. The total experiment time is  $T_{\text{exp}} = |A \times B \times C \times D \times E \times F| \times 0.05 \times t_{\text{itr}}$ , where  $A$  is the number of chunk sizes ( $|A| = 3$ ),  $B$  is the number of channels ( $|B| = 5$ ),  $C$  is the number of NTHREADS values ( $|C| = 4$ ),  $D$  is the number of queue-pairs per connection ( $|D| = 128$ ),  $E$  is the number of (SOCKET\_NTHREADS, NSOCKS\_PERTHREAD) pairs ( $|E| = 211$ ), and  $F$  is the number of tile sizes ( $|F| = 5$ ). Thus  $T_{\text{exp}} = (A \cdot B \cdot C \cdot D \cdot E \cdot F) \times 0.05 \times t_{\text{itr}}$ . Since each iteration takes about 30 s, the full sweep would require roughly 140 days. To make this feasible, we sample just 0.05 % of the space, reducing the total experimental cost to about 35.1 hours.

Considering the AllReduce scale of 64 GPUs in the LLM application, this equates to more than **2,240 GPU-hours**. Once the NCCL-test data are collected, one can train a random-forest regression model and then use a randomized selection procedure to identify the configuration with the highest predicted performance. Because this approach requires both fitting the model and searching across the full configuration space, its analysis time exceeds that of our ML-based Parameter Importance Analyzer, which only explores a much smaller subset of configurations.

**8.2.2 EGG Cost.** Compared to the sampling method, expert-driven method [21] evaluates NCCL and RCCL performance across message sizes from 1 B to 1 GiB. This methodology requires over **1,000 GPU-hours** to run tests on a 4096-GPU cluster, about 30 minutes to analyze the collected data, and significant manual effort and domain expertise to interpret and guide the tuning process.

**8.2.3 CCLINSIGHT Cost.** Here, we analyze the cost of our proposed approach, CCLINSIGHT. We start by analyzing the primitive-centric analyzer, then move to the evaluation of key parameters, followed by the ML-based parameter importance analyzer.

**Primitive Analyzer Cost.** In primitive-level profiling, we vary each parameter independently over all its possible settings. Hence the total runtime is  $T_{\text{prims}} = \sum_{X \in \{A, B, C, D, E, F\}} |X| \cdot t_{\text{itr}}$ . Each primitive micro-benchmark iteration completes in about 15 seconds. Thus,  $T_{\text{prims}} \approx (3 + 5 + 4 + 128 + 211 + 5) \times 15 \text{ s} \approx 1.48 \text{ h}$ . Considering the GPU scale (4 GPUs) used in the primitive-level profiling, the total cost amounts to **5.92 GPU-hours**.

**Key Parameter Experiment Manager Cost.** Our design achieves significant cost savings by running the collective-level microbenchmark (NCCL-Test) over only the key parameters. Specifically, we select  $A$ ,  $B$ , and  $F$  (as defined in §8.2.1) as our key parameters. Thus, the total cost of the Key Parameter Experiment Manager is  $T_{\text{KP}} = |A \times B \times F| \cdot t_{\text{itr}} = (A \cdot B \cdot F) t_{\text{itr}} \approx 0.625 \text{ h}$ . Taking into account the AllReduce scale of 64 GPUs in the LLM application, the total cost is **40 GPU-hours**.

**Combined Cost and Comparison.** Now we describe the combined cost of using CCLINSIGHT to find high-performing xCCL configurations. The two key components are the primitive-centric analyzer and the key parameter experiment manager, and their respective costs are  $T_{\text{prim}}$  and  $T_{\text{KP}}$ . Although it is part of CCLINSIGHT, the ML-based parameter importance analyzer is primarily a tool used to validate the important parameters determined through the primitive-centric analysis. Thus, we do not include  $T_{\text{ML}}$  in the cost of using CCLINSIGHT. The combined cost of the primitive-centric analyzer and key parameter experiment manager cost is:  $T_{\text{combined}} = T_{\text{prims}} + T_{\text{KP}}$ .

To compare the cost of CCLINSIGHT with AFNFA and EGG, we exclude the analysis overhead of the latter two methods. Thus, the reported cost reflects only the execution time for all three approaches.

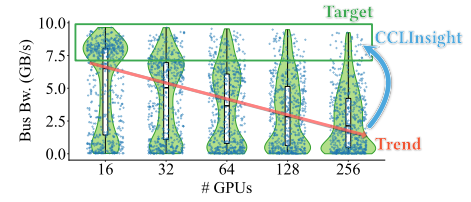
**Table 2: Comparison of the Execution Costs of AFNFA, EGG, and CCLINSIGHT for the LLM Application in §8.1.3**

Approach	AFNFA	EGG	CCLINSIGHT
Cost (GPU-hours)	2240	1000	45.92

### 8.3 A Scaling Law Found by CCLINSIGHT

We observe an important scaling law that arises with scale. Specifically, the performance distribution across all parameter configurations is often bi- or multi-modal and shifts as the number of accelerators increases. We refer to this phenomena as the **Collapsed Optimal Configuration Scaling Law**. The law is demonstrated in Fig. 18, where the scale drastically affects the distribution. Critically,

many parameter configurations exhibit relatively good performance at small scales. As the scale increases, however, the distribution shifts downwards, indicating that most parameter configurations show markedly diminished performance compared with the optimal and even the same configurations performance at a smaller scale. Nevertheless, specific configurations still perform nearly optimally despite the relative sparsity at the top of the performance range. Therefore, it is clear that choosing the correct parameter configuration becomes increasingly difficult and important with scale. We find that CCLINSIGHT assists in quickly reaching these configurations with low cost. Note that while not shown in Fig. 18, we see the same trend when maintaining constant message sizes and examining GPU scaling behavior. Similar patterns emerge for the H100 and RTX5000 clusters for both algorithms.



**Figure 18: Collapsed Optimal Configuration Scaling Law. Area and + mark meaning is similar to Fig. 15c.**

## 9 Conclusion and Future Work

In this study, we introduce a primitive-based analytical approach that simplifies and unifies the assessment of GPU-based collective communication with low overhead. We also develop a tool to conduct experiments and collect empirical performance information at the primitive and collective levels. We combine our extensive empirical results with our primitive analytical approach to provide an in-depth analysis of GPU-based collective communication performance. Through extensive evaluations of NCCL and MSCCL across three clusters, we show that CCLINSIGHT can drive substantial performance improvements, achieving up to a 43.90% improvement in NCCL and a 40.64× speedup in MSCCL compared to default configurations. In addition, we show that CCLINSIGHT improves performance by up to 2.27× on NCCL and 3.00× on MSCCL for LLM applications. Through our CCLINSIGHT, we discover and present the Collapsed Optimal Configuration Scaling Law, showing that although the number of high-performing configurations drops rapidly with GPU scale, finding them is feasible with the correct tool. In future work, we plan to release CCLINSIGHT as an open source tool to benefit the broader community.

## References

- [1] [n. d.]. Nsight. <https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html>.
- [2] [n. d.]. NVIDIA A100 Tensor Core GPU Architecture. <https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf>.
- [3] [n. d.]. NVIDIA H100 Tensor Core GPU Architecture. <https://resources.nvidia.com/en-us-tensor-core>.
- [4] [n. d.]. XGBoost Python.
- [5] 2017. NCCL. <https://github.com/NVIDIA/nccl>.
- [6] 2019. Massively Scale Your Deep Learning Training with NCCL 2.4. <https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/>.

- [7] 2020. Understanding the Visualization of Overhead and Latency in NVIDIA Nsight Systems. <https://developer.nvidia.com/blog/understanding-the-visualization-of-overhead-and-latency-in-nsight-systems/>.
- [8] 2024. GPU Bandwidth and GPUDirect TCP/UDP on Google Kubernetes Engine. <https://cloud.google.com/kubernetes-engine/docs/how-to/gpu-bandwidth-gpudirect-tcp>.
- [9] 2024. NVIDIA/Nccl-Tests. NVIDIA Corporation.
- [10] 2025. Alps Research Infrastructure. <https://www.cscs.ch/computers/alps>.
- [11] 2025. LUMI Supercomputer. <https://lumi-supercomputer.eu/>.
- [12] AMD. 2022. ROCm Communication Collectives Library. <https://github.com/ROCmSoftwarePlatform/rccl>.
- [13] Felix Berkenkamp, Andreas Krause, and Angela P. Schoellig. 2023. Bayesian Optimization with Safety Constraints: Safe and Automatic Parameter Tuning in Robotics. *Machine Learning* 112, 10 (Oct. 2023), 3713–3747. <https://doi.org/10.1007/s10994-021-06019-1>
- [14] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32. <https://doi.org/10.1023/A:1010933404324>
- [15] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models Are Few-Shot Learners. <https://doi.org/10.48550/arXiv.2005.14165> arXiv:2005.14165 [cs]
- [16] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2021. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) (PPoPP '21). Association for Computing Machinery, New York, NY, USA, 62–75. <https://doi.org/10.1145/3437801.3441620>
- [17] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. <https://doi.org/10.1145/2939672.2939785>
- [18] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. 2022. GC3: An Optimizing Compiler for GPU Collective Communication. *arXiv preprint arXiv:2201.11840* (2022).
- [19] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. 2023. MSCCLang: Microsoft Collective Communication Language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 502–514. <https://doi.org/10.1145/3575693.3575724>
- [20] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoeftler. 2020. An In-Depth Analysis of the Slingshot Interconnect. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. IEEE Press, Atlanta, Georgia, 1–14.
- [21] Daniele De Sensi, Lorenzo Pichetti, Flavio Vella, Tiziano De Matteis, Zebin Ren, Luigi Fusco, Matteo Turisini, Daniele Cesarini, Kurt Lust, Animesh Trivedi, Duncan Roweth, Filippo Spiga, Salvatore Di Girolamo, and Torsten Hoeftler. 2024. Exploring GPU-to-GPU Communication: Insights into Supercomputer Interconnects. <https://doi.org/10.48550/arXiv.2408.14090> arXiv:2408.14090 [cs]
- [22] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948* (2025).
- [23] HuggingLlama. [n. d.]. Llama-7B model on Hugging Face. <https://huggingface.co/huggyllama/llama-7b>. Accessed: 2025-01-16.
- [24] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 2022. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) (ASPLOS '22). Association for Computing Machinery, New York, NY, USA, 402–416. <https://doi.org/10.1145/3503222.3507778>
- [25] 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.
- [26] Kawthar Shafie Khorassani, Chen-Chun Chen, Bharath Ramesh, Aamir Shafi, Hari Subramoni, and Dhavalaswar K. Panda. 2023. High Performance MPI over the Slingshot Interconnect. *Journal of Computer Science and Technology* 38, 1 (Feb. 2023), 128–145. <https://doi.org/10.1007/s11390-023-2907-5>
- [27] Heehoon Kim, Junyeol Ryu, and Jaemin Lee. 2024. TCCL: Discovering Better Communication Paths for PCIe GPU Clusters. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (ASPLOS '24, Vol. 3). Association for Computing Machinery, New York, NY, USA, 999–1015. <https://doi.org/10.1145/3620666.3651362>
- [28] Robin Kobus, Daniel Jünger, Christian Hundt, and Bertil Schmidt. 2019. Gossip: Efficient Communication Primitives for Multi-GPU Systems. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP '19)*. Association for Computing Machinery, New York, NY, USA, 1–10. <https://doi.org/10.1145/3337821.3337889>
- [29] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (Jan. 2020), 94–110. <https://doi.org/10.1109/TPDS.2019.2928289>
- [30] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. 2018. Tartan: Evaluating Modern GPU Interconnect via a Multi-GPU Benchmark Suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 191–202. <https://doi.org/10.1109/IISWC.2018.8573483>
- [31] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. 2023. Colossal-AI: A Unified Deep Learning System For Large-Scale Parallel Training. In *Proceedings of the 52nd International Conference on Parallel Processing*. 766–775.
- [32] Xinyi Liu, Yujie Wang, Fangcheng Fu, Xupeng Miao, Shenhan Zhu, Xiaonan Nie, and Bin Cui. 2025. NetMoE: Accelerating MoE Training through Dynamic Sample Placement. In *The Thirteenth International Conference on Learning Representations*.
- [33] Octavio Loyola-González. 2019. Black-Box vs. White-Box: Understanding Their Advantages and Weaknesses From a Practical Point of View. *IEEE Access* 7 (2019), 154096–154113. <https://doi.org/10.1109/ACCESS.2019.2949286>
- [34] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1633–1649. <https://doi.org/10.1145/3318464.3389705>
- [35] NVIDIA. [n. d.]. NVIDIA Nsight Systems. NVIDIA Corporation.
- [36] Carl Pearson. 2023. Interconnect Bandwidth Heterogeneity on AMD MI250x and Infinity Fabric. <https://doi.org/10.48550/arXiv.2302.14827> arXiv:2302.14827 [cs]
- [37] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research* 12, 85 (2011), 2825–2830.
- [38] Simone Pellegrini, Thomas Fahringer, Herbert Jordan, and Hans Moritsch. 2010. Automatic Tuning of MPI Runtime Parameter Settings by Using Machine Learning. In *Proceedings of the 7th ACM International Conference on Computing Frontiers (CF '10)*. Association for Computing Machinery, New York, NY, USA, 115–116. <https://doi.org/10.1145/1787275.1787310>
- [39] Simone Pellegrini, Radu Prodan, and Thomas Fahringer. 2012. Tuning MPI Runtime Parameter Setting for High Performance Computing. In *2012 IEEE International Conference on Cluster Computing Workshops*. 213–221. <https://doi.org/10.1109/ClusterW.2012.15>
- [40] Hao Qi, Liuyao Dai, Weicong Chen, Zhen Jia, and Xiaoyi Lu. 2023. Performance Characterization of Large Language Models on High-Speed Interconnects. In *2023 IEEE Symposium on High-Performance Interconnects (HOTI)*. 53–60. <https://doi.org/10.1109/HOTI59126.2023.00022>
- [41] Redstone Software. 2008. Black-Box vs White-Box Testing.
- [42] Ibai Roman, Josu Ceberio, Alexander Mendiburu, and Jose A. Lozano. 2016. Bayesian Optimization for Parameter Tuning in Evolutionary Algorithms. In *2016 IEEE Congress on Evolutionary Computation (CEC)*. 4839–4845. <https://doi.org/10.1109/CEC.2016.7744410>
- [43] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. 2023. TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 593–612. <https://www.usenix.org/conference/nsdi23/presentation/shah>
- [44] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems*, Vol. 25. Curran Associates, Inc.
- [45] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajwal Bhargava, Shrutu Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
- [46] Zibo Wang, Yuhang Zhou, Chen Tian, Xiaoliang Wang, and Xianping Chen. 2023. AFNFA: An Approach to Automate NCCL Configuration Exploration. In *Proceedings of the 7th Asia-Pacific Workshop on Networking (Hong Kong, China) (APNet '23)*. Association for Computing Machinery, New York, NY, USA, 204–205. <https://doi.org/10.1145/3600061.3600068>

- [47] Adam Weingram, Yuke Li, Hao Qi, Darren Ng, Liuyao Dai, and Xiaoyi Lu. 2023. xCCL: A Survey of Industry-Led Collective Communication Libraries for Deep Learning. *Journal of Computer Science and Technology* 38, 1 (Feb. 2023), 166–195. <https://doi.org/10.1007/s11390-023-2894-6>