AutoCCL: Automated Collective Communication Tuning for Accelerating Distributed and Parallel DNN Training

Guanbin Xu, Zhihao Le, Yinhe Chen, Zhiqi Lin, and Zewen Jin, *University of Science and Technology of China;* Youshan Miao, *Microsoft Research;* Cheng Li, *University of Science and Technology of China; Anhui Province Key Laboratory of Biomedical Imaging and Intelligent Processing; Institute of Artificial Intelligence, Hefei Comprehensive National Science Center*

https://www.usenix.org/conference/nsdi25/presentation/xu-guanbin

This paper is included in the
Proceedings of the 22nd USENIX Symposium on
Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

`AutoCCL`: Automated Collective Communication Tuning for Accelerating Distributed and Parallel DNN Training

Guanbin Xu[†] Zhihao Le[†] Yinhe Chen[†] Zhiqi Lin[†] Zewen Jin[†] Youshan Miao[‡]
Cheng Li[†,∘]
[†]*University of Science and Technology of China,* [‡]*Microsoft Research*
[∘]*Anhui Province Key Laboratory of Biomedical Imaging and Intelligent Processing*
Institute of Artificial Intelligence, Hefei Comprehensive National Science Center

Abstract

The collective communication libraries are pivotal in optimizing the performance of distributed and parallel deep neural network (DNN) training. Most network optimizations are under the assumption that these libraries are well-tuned, ignoring their low-level parameter selection. In this paper, we present a novel automated tuning method `AutoCCL` that significantly improves communication performance without incurring additional costs. One of the primary challenges we tackle is the state explosion in searching for the optimal configuration. To overcome this, we decouple implementation-related parameters from those sensitive to the search space size and propose a divide-and-conquer algorithm, minimizing the requirement for exhaustive trials. We further propose an online tuning approach that accounts for communication-computation interference to enhance accuracy in finding optimal configurations, while hiding tuning overhead within early iterations of training jobs. We implement `AutoCCL` atop NCCL, a leading and widely-used communication library provided by NVIDIA. Our evaluation on both a 2-node cluster (16 A40 GPUs, intra-node NVLink, inter-node 2× 400Gbps InfiniBand) and a 4-node cluster (32 A40 GPUs, intra-node PCIe, inter-node 100Gbps InfiniBand) demonstrates that `AutoCCL` achieves 1.24-1.29× and 1.15-1.22× speedups on microbenchmarks compared to NCCL and another SOTA NCCL tuner, respectively, and up to 1.80× and 1.49× with concurrent computation. End-to-end evaluations on three large language models and one vision model show 1.07-1.32× improvements in per-iteration training time.

1 Introduction

Training contemporary DNN models often distributes computation over a cluster of GPUs and relies on collective communication primitives to frequently exchange model data in a group-wise manner. However, many studies have pointed out that communication is a well-known bottleneck that throttles training performance [14, 26, 27, 33, 34, 36].

Communication optimization has become a research hotspot, garnering growing interest including communication-computation overlapping [13, 19, 22, 23, 31, 43, 46, 50, 59, 60, 67], message compression [9, 53, 56, 62], and new collective algorithm design [10, 12, 15, 16, 27–29, 32, 35, 42, 47, 57]. These proposals assume the off-the-shelf communication libraries such as NCCL [5] to be already well-tuned. Some works [12, 47] have attempted to provide empirical guidance, but in Section 3.1, we have found that the guidance is not always correct in practice. Besides, our study highlights that collective communication tasks can be further improved, e.g., up to 35% higher bandwidth, by tuning low-level performance-sensitive parameters. Unfortunately, these libraries are often used as black boxes, and the potential for tuning them across different hardware and communication tasks is overlooked.

In this paper, we propose `AutoCCL`, a tool for automatically tuning collective communication with accelerated end-to-end training performance. Compared to existing communication optimizations, `AutoCCL` aims to provide *a free lunch* and transparently support upper-level training jobs, maintaining model accuracy without requiring additional hardware investment. However, `AutoCCL`'s design faces the following challenges:

First, existing work lacks a comprehensive analysis of the mainstream collective communication libraries to identify key performance-sensitive parameters as tuning candidates, understand the joint impact of these parameters, and establish practical tuning guidelines. Second, computation and communication are executed concurrently during training. Therefore, tuning for communication also necessitates accounting for computational interference. This understanding needs to be further adapted to different hardware, as well as the dynamics in the concurrent combination of computation and communication due to runtime scheduling.

To address the above challenges, we propose the following innovations. First, we conduct a comprehensive tuning study (Section 2.3 and Section 3) to identify six low-level NCCL parameters that are performance-sensitive, forming a very large search space. Then, we experimentally analyze and theoretically model these parameters and the relationships among them and make some important and instructive

This research project began in June 2023.

observations. For example, the parameters can be classified into two categories, one defining the implementation method of communication primitives and the other determining the resource allocations. The former is a prerequisite for the latter, and the former has a small range of values while the latter has a large range of values. In addition, we analyze the resource parameters using the control variable method and find that their combination is characterized by a unimodal function on the communication bandwidth. These findings are far beyond a recent preliminary study [64].

Second, guided by the aforementioned analysis, we design an efficient tuning algorithm (Section 4) that avoids a brute-force search within the parameter space. This algorithm handles implementation parameters and resource parameters separately. Since the combinations of the former are relatively few and the analysis of resource parameters must begin with the determination of the communication task, the tuning algorithm first divides the implementation parameters into several subspaces, each representing an independent search task. Within any given subspace, we employ a coordinate descent method to search for the optimal combination of resource parameters. This approach is feasible because the characteristics of unimodal functions indicate that their combination to the communication bandwidth curve will initially increase, then reach a peak, and subsequently decrease gradually until reaching a steady state (at which point bandwidth becomes the bottleneck).

Finally, we propose an online tuning method (Section 5) that co-runs with training. We exploit the iterative nature of DNN training (where identical communication tasks are repeated throughout the training process) to integrate the tuning process into the initial training iterations. Once the optimal configuration is obtained, it is updated via atomic broadcast to all nodes participating in collective communication for consistent future usage. The advantage of this online approach is that we can directly capture the performance of communication primitives under the influence of computational interference and scheduling dynamics. After integrating this into the tuning algorithm, the performance analysis achieves higher accuracy, thus eliminating the need to model uncertain factors directly, such as hardware capabilities and the concurrency of tasks in the real training environment.

We implemented the tuning tool `AutoCCL` on top of NCCL, a leading and widely adopted collective communication library. `AutoCCL` maintains compatibility with the NCCL interface, allowing it to seamlessly integrate with training frameworks like PyTorch and transparently support any DNN model training task. We've made it open-source `AutoCCL` [2]. We conducted extensive testing on a 4-node cluster with 32 A40 GPUs and a 100Gbps network, as well as a 2-node cluster with 16 A40 GPUs and two 400Gbps links. Pure communication experiments with different communication primitives show that, compared to NCCL and AFNFA [64] (the most recent NCCL tuner), our communication bandwidth speedups are 1.24-1.29$\times$ and 1.15-1.22$\times$ higher, respectively. In microbenchmark experiments with computational interference, the results show that `AutoCCL` achieves even greater average gains, with improvements of 24.1% and 20.7%, respectively. Additionally, we conduct end-to-end training tests on three popular large-scale models and one computer vision model. Compared to NCCL and AFNFA, `AutoCCL` achieves average training speed improvements of 16.7% and 14.5%, respectively, with gains of up to 32% on some large models. Note that the training gains have taken tuning overhead into account, and since `AutoCCL` converges quickly, the tuning overhead is negligible.

In summary, we make the following contributions:

- A comprehensive study that uncovers the tuning guidelines of low-level performance parameters of collective communication primitives in NCCL, a leading communication library;
- An online tuning method that implements the parameter subspace division and intra-subspace coordinate descent search algorithms and leverages training iterativeness for accurately modeling the impacts of parameter assignment in the context of computational interference;
- The `AutoCCL` tuner that incorporates this tuning method and transparently supports training jobs;
- An in-depth evaluation of `AutoCCL` compared to NCCL and another state-of-the-art tuner on communication micro-benchmarks and representative training jobs.

2 Background

2.1 Distributed and Parallel DNN Training

It has been a common practice to distribute and parallelize DNN training jobs over a cluster of GPUs for fast model parameter updating [17, 20, 24, 25, 30, 37–39, 44, 45, 48, 49, 51, 55, 58, 66]. For instance, as shown in Table 1, when the model size does not exceed a single GPU's memory budget, *data parallelism* is often employed to let GPUs consume disjoint data partitions for collective model training. This requires GPUs to frequently synchronize gradients, with the transmission volume equal-sized with model parameters. When a model's memory consumption exceeds the GPU capacity, in addition to *data parallelism*, AI participants employ either *tensor parallelism* or *pipeline parallelism* to partition a model into smaller computational units, which are distributed across various GPUs. This results in more complex communication patterns among GPUs, which are often on the critical path of the training pipeline, compared to *data parallelism*. Recently, various studies have already proved that communication is the major bottleneck that limits the scalability of distributed and parallel DNN training [14, 26, 27, 33, 34, 36].

A wide range of optimizations aim to address communication bottlenecks, including scheduling [13, 19, 22, 23, 31,

Table 1: The parallelism and collective communication primitives used by training 4 DNN models on a cluster of 32 GPUs (four machines). *B* stands for billion model parameters. *TP*, *DP*, and *PP* stand for tensor parallelism [48], data parallelism [32], and pipeline parallelism [20], respectively. The numbers in parentheses following these parallelism approaches represent the corresponding number of GPU participants. For each collective communication primitive, its (*x_y_z*) value represents that an *x*-sized message is exchanged among a *y*-sized communication group for *z* times per training iteration.

Model	Phi-2-2B	Llama-3.1-8B	Yi-1.5-34B	VGG-19-0.14B
Parallelism[1]	TP (8) + DP (4)	TP (8) + DP (4)	TP (8) + PP (4)	DP (32)
AllGather	(80 MB_8_3,120)	(128 MB_8_6,240), (857 MB_4_1)	(56 MB_8_61,440)	—
ReduceScatter	(80 MB_8_2,064)	(128 MB_8_4,128), (1,710 MB_4_1)	(56 MB_8_93,184)	—
AllReduce	(632 MB_4_1)	—	—	(15-392 MB_32_6)

46, 50, 59, 60, 67], data compression [8, 9, 53, 56, 62, 63], sparsity exploration [10, 28], and topology-based algorithm design [12, 15, 16, 27, 29, 32, 35, 42, 42, 47, 57]. The vast majority of these approaches focus on improving communication performance based on the assumption that the underlying communication library is well-tuned and will effectively deliver their improvements. However, we have experimentally validated that the communication tuning matters a lot in Section 3 with significant bandwidth benefits.

2.2 Collective Communication

Distributed and parallel DNN training introduces a distinct class of communication patterns, known as *collective communication*, where data is aggregated or disseminated across multiple GPUs. The NVIDIA Collective Communication Library (NCCL) is designed to optimize multi-GPU and multi-node communication for NVIDIA GPUs and networking systems [5]. It offers high-performance primitives like *ReduceScatter*, *AllGather*, *AllReduce*, and so on. These APIs are optimized to provide high bandwidth and low latency communication over PCIe and NVLink interconnects within a single node, as well as over networks across multiple nodes. In addition, RCCL (ROCm Collective Communication Library) [1] is AMD's counterpart to NCCL. In this paper, we focus on NCCL since it is the most widely used and its design philosophy has influenced followers such as RCCL.

Table 1 summarizes the training parallelisms for the four deep neural network training tasks, including three large language models and a computer vision model, with typical message sizes and the collective communication primitives used. The detailed hardware and workload configurations can be found in Table 8. Of the four model training tasks, the VGG-19 model is the smallest, necessitating the use of only data parallelism. VGG-19 employs 32-way data parallelism, so every GPU needs to exchange its six bucketed gradients (15-392 MB in size) with any other GPU within the 32-GPU communication group via *AllReduce* in each iteration.

Unlike VGG-19, the other three large language models utilize a complex hybrid parallelism, combining 8-way tensor parallelism and 4-way data parallelism or 4-way pipeline parallelism. In such a configuration, for example, Yi-1.5-34B

Table 2: Primitive configuration parameters of NCCL

Parameter	Value Range
Algorithm (A)	Tree, Ring
Protocol (P)	LL, LL128, Simple
Transport (T)	peer-to-peer (P2P), shared memory (SHM)
Nchannel (NC)	$1 \le n \le 128, \quad n \in \mathbb{N}$
Nthread (NT)	$n = 32 \times i, \quad i \in \{1, 2, 3, \ldots, 20\}$
Chunk size (C)	$n = 256 \times i, \quad i \in \{1, 2, 3, \ldots, 8K\}$

executes *AllGather* 61,440 times in a single training iteration, each time transferring 56 MB of activation data among eight GPUs within a single machine. It also executes *ReduceScatter* 93,184 times, each time passing 56MB of activation data between eight GPUs to each other within a single machine. Phi-2-2B and Llama-3.1-8B transfer more data each iteration, up to 1,710 MB, as it also requires transferring the model gradient along the data-parallel dimension between four GPU servers with *AllGather*, *ReduceScatter*, and *AllReduce*, respectively, due to their four-way data parallelism.

In summary, multiple types of collective communication primitives are frequently used in distributed DNN training, with message sizes ranging from tens of MB to a few GB, varying communication group sizes, and hardware setups. Therefore, it is crucial to efficiently execute communication tasks despite their complexity and variety.

2.3 Low-Level Primitive Configurations

We conduct a systematic and comprehensive study of NCCL's parameter space, analyzing all 158 parameters, including 93 undocumented ones. Our findings reveal that NCCL has 28 performance-sensitive parameters, categorized as follows: 1 for *algorithm* (*A*), 3 for *protocol* (*P*), 3 for *transport* (*T*), 11 for *nchannel* (*NC*), 3 for *nthread* (*NT*), and 7 for *chunk size* (*C*). Other parameters will be discussed in Section 8. When users call a communication primitive, NCCL runtime will create a corresponding collective communication task, and assign it a *configuration* in the form of $< A, P, T, NC, NT, C >$. As shown in Table 2, the parameter *A* determines how data is distributed, combined, or aggregated across GPUs and has several possible values like ring or tree. Specifically, when a *ring* algorithm is used for the *allreduce* primitive, data is transferred sequentially between nodes in a ring-like fash-

Table 3: Various configurations of *AllGather* (80MB) on the 8-GPU node with intra-node **PCIe** w.r.t. varied NCCL configurations. C_{P0} refers to the default NCCL configuration.

Config	A	P	T	NC	NT	C
$C_{P0}{}^*$	Ring	Simple	SHM	2	256	2 MB
C_{P1}	Ring	Simple	P2P	2	96	32 KB
C_{P2}	Ring	Simple	P2P	8	256	32 KB
C_{P3}	Ring	Simple	P2P	8	96	32 KB

ion for both reduction and broadcast phases. *P* defines how data is moved across GPU memory hierarchies and managed within the GPUs, such as loading data into the GPU cores (SMs) and performing necessary operations (like reductions). Among its options, *LL* stands for the Low-Latency protocol, optimized for small message sizes to minimize latency. *T* refers to the mechanism used for physically transferring data between GPUs, either within the same machine or across different machines. For instance, its *P2P* (Peer-to-Peer) option defines a direct connection between GPUs bypassing CPU, typically using NVLink or PCIe. *SHM* refers to enabling data transmission between certain GPUs via shared memory.

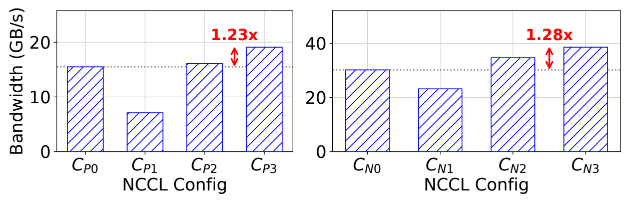
The remaining three parameters correspond to the allocation of resources or the degree of parallelism during communication. First, for a large message, NCCL will decompose data into *NC* partitions, with each partition bonded to a threadblock for independent transmission and processing. Each threadblock consists of a certain number of threads, defined by *NT*. In addition, each data partition can be further decomposed into several chunks, with the size defined by *C*. Therefore, *NC* threadblocks are transmitted chunks concurrently, and each threadblock processes its chunks sequentially.

3 NCCL Tuning Opportunities

However, we find that the default configuration of NCCL is often not optimal, and tuning the parameter values of the configuration can have a huge impact on communication performance during the DNN model training. Therefore, here, we will show the improvement of communication performance, as well as the positive impact during training performance through several examples.

3.1 Stand-alone Communication

Here, we consider communication tasks running alone without any interference. First, we take the $< AllGather, 80M, 8-A40-PCIe >$ communication task from the Phi-2-2B model in Table 1, i.e., *AllGather* aggregates 80MB data among 8 A40 GPUs in a single machine. Table 3 lists four primitive configurations that are used in the test. Figure 1 reports the communication bandwidth (higher is better). The best configuration C_{P3} achieves 2.69× the bandwidth of the worst configuration C_{P1}. The difference is that C_{P3} splits the message into more data partitions (larger *NC*), which are split into smaller chunks (smaller *C*), while adjusting the number of threads (*NT*) to align with the transmission granularity (*C*).



(a) PCIe (b) NVLink

Figure 1: Bandwidth comparison of using various NCCL configurations for <AllGather, 80M, 8-A40-*>

Table 4: Various configurations of *AllGather* (80MB) on the 8-GPU node with intra-node **NVLink** w.r.t. varied NCCL configurations. C_{N0} refers to the default NCCL configuration.

Config	A	P	T	NC	NT	C
$C_{N0}{}^*$	Ring	Simple	P2P	8	512	2 MB
C_{N1}	Ring	Simple	P2P	64	512	108 KB
C_{N2}	Ring	Simple	SHM	8	512	108 KB
C_{N3}	Ring	Simple	SHM	64	512	108 KB

Table 5: Various configurations of *AllReduce* on the 8-GPU node with intra-node **PCIe**. Each task is shown in the format x, y, where x indicates the message size and y represents the communication group size. *P* and *T* are omitted as they are set to Simple and SHM, respectively, across all configurations. C_0 refers to the default NCCL configuration.

Task	Config	A	NC	NT	C	Bwd (GB/s)
64 MB, 16	$C_0{}^*$	Ring	2	256	512KB	4.0
64 MB, 16	C_1	Tree	2	256	512KB	5.4
64 MB, 16	C_2	Tree	8	160	59 KB	8.9
15 MB, 8	C_3	Tree	8	160	59 KB	8.1
15 MB, 8	C_4	Ring	10	128	27 KB	8.8

All these parameters are set to utilize the parallel computing power of GPUs and the network transmission bandwidth. Compared to C_{P3}, C_{P1} uses a smaller concurrency count *NC*, leading to significant performance degradation. The performance of C_{P2} is lower than C_{P3}, even with an increase in the number of threads. It is worth noting that C_{P0}, the NCCL default configuration determined by NCCL's built-in cost model, is not optimal; its bandwidth is only 81.2% of that of C_{P3}.

Second, we run the aforementioned *AllGather* task in a new hardware environment with GPUs interconnected via NVLink instead of PCIe. As shown in Table 4, we test 4 configurations, namely, C_{N0} to C_{N3}. Figure 1 shows that the best configuration C_{N3} reaches 1.28× the bandwidth of the worst configuration C_{N0}. By comparing C_{N0} (the default configuration selected by NCCL) with C_{N3}, we observe that *T* plays a significant role, as C_{N3} employs share-memory (SHM) communication with better performance. By comparing C_{N3} with the second best performing C_{N2}, using more channels leads to better parallel performance, consistent with our observation in the PCIe experiment. Also, comparing C_{N1}, C_{N0} and C_{N3}, we can find that increasing *NC* plays the exact opposite role when setting the *T* parameter as P2P.

Table 6: Bandwidth of *AllGather* (80MB) on the 8-**NVLink**-GPU node w.r.t. varied computational interference levels

Interference level	NCCL Bwd	Tuned Bwd
Communication-only	30.08 GB/s	38.62 GB/s
Light computation	26.21 GB/s	35.14 GB/s
Heavy computation	18.26 GB/s	32.44 GB/s

Finally, we investigate the joint effect of different communication tasks, different hardware environments, and different configuration parameter choices. To this end, we take the *AllReduce-64MB* and *AllReduce-15MB* communication tasks from the VGG19 model, as listed in Table 1, running on a single machine with 8 A40 GPUs and two machines with 16 A40 GPUs, with PCIe for intra-node network and 100 Gbps InfiniBand for inter-node network. We test five configurations, with performance shown in Table 5. Comparing C_0 and C_1, we find that the algorithm parameter A becomes the key factor to performance as NCCL's default algorithm misjudging that the Ring algorithm better fits current hardware, but Tree is better in practice. Besides, if applying the best configuration C_2 for task $< AllReduce, 64MB, 16 >$ to the task $< AllReduce, 15MB, 8 >$ as C_3, it will be approximately 10% lower than the performance reached by C_4, indicating that configuration rediscovery is needed, even for very similar tasks. Comparing C_3 and C_4, we find that Ring algorithm is a better choice, and increasing the degree of parallelism (NC) can indeed enhance performance. In addition, we need to shrink both the minimum granularity of the transmission C and the number of threads NT.

According to all the experiments above, we observe that the default configurations determined by NCCL are often not optimal, leaving significant performance potential to NCCL parameter tuning. Our experiments in Section 6 prove that this problem also exists in multiple nodes. For various tasks on the same hardware or even the same task on different hardware, different optimal configurations are also needed. Also, it is not clear whether there is a certain pattern in the selection of parameters and the interactions between those parameters. Therefore, one needs to add an auto-tuning function so that the training task running above NCCL freely enjoys the optimized fast communication.

3.2 Concurrent Computational Interference

Communication and computation tasks without data dependencies are usually executed concurrently, and it is infrequent for communication to occupy resources exclusively. The competition between computation and communication for GPU resources, especially SM, cache, and global bandwidth, will cause the performance of each to degrade. Therefore, to better study the configuration tuning of NCCL communication tasks, especially in the presence of resource contention caused by computation tasks, we design the following NCCL tuning experiments with computational interference.

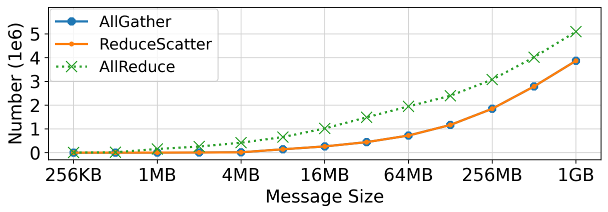


Figure 2: Combinations number of different communication

We introduce extra computational tasks over the *AllGather* communication task from Table 4, and take default configuration of NCCL as a baseline. The computational task we used is GEMM [4], which computes matrix multiplication A[m, k] × B[k, n] operation followed by a sigmoid [7] operation. Here, we choose different values of m, k, n in GEMM to represent the light and heavy computations. Table 6 shows that computational interference brings a significant communication performance degradation, even with light computation. For example, the *AllGather* bandwidth drops by 12.8% with interference by light computation and up to 39.3% by heavy computation. However, with carefully primitive configuration tuning, *AllGather*'s bandwidth has been improved by 34-78% for all settings. It's worth noting that, after configuration tuning, the cases with computational interference can be 7.8-16.8% faster than default NCCL without interference.

3.3 Challenges in NCCL Tuning

Tuning NCCL primitive configurations presents challenges that require addressing the following two questions.

How to quickly find the performant configuration in a large space? For specific communication types, message sizes, and communication group sizes, the configuration tuning space is huge. As shown in Figure 2, the number of possible combinations can reach up to millions. Traversing such a space and comparing the latencies and bandwidths of candidates is time-consuming. For example, testing all parameter combinations for *AllGather* with 80MB takes several hours on a single machine with 8 A40 GPUs connected by PCIe. With the increase of message size and communication group size, the time overhead expands significantly.

How to model computational interference w.r.t the training runtime dynamics? For parallel training tasks of a model, there are usually highly variable operators, dimensions of operator partitioning, and inputs to the operators, and consequently a diverse range of computational tasks are executed on the GPU. Modeling computational tasks and their impact on communication further increases the space of NCCL configuration. In addition, training frameworks add scheduling optimizations to the runtime, and introduce dynamics that can make the concurrent execution of computation and communication tasks unpredictable in advance. Exploring all potential combinations of these factors simultaneously results in an impractical search cost.

4 Communication Tuning Method

Next, we will design a unified, simple, and efficient configuration parameter tuning tool, `AutoCCL`, for collective communication primitives. We first consider communication modeling and tuning in an ideal state without computational interference. Recall that Table 2 presents six parameters that make up each configuration, and the selection of these parameter values forms a large search space. To address this challenge, we analyze NCCL's parameters and make several observations. First, we categorize the six parameters and apply different approaches to different types of parameters. We divide the parameters into two categories: those related to the implementation of primitives and those related to resource allocation, with a detailed analysis provided in Section 4.1.

Second, we find that resource allocation parameters are the primary cause of the large search space, and there are some patterns in the interactions between these parameters. By examining these interactions and their joint effects on performance, we propose a theoretical model to predict the performance of parameter combinations, thus avoiding exhaustive search in the vast parameter space (see Section 4.2). Finally, we integrate all these observations into a tuning algorithm, which is detailed in Section 4.3.

4.1 Parameter Division

First, algorithm, protocol, and transport are implementation-related and together determine the topology of the algorithm execution logic, the method of data transmission, and the software stack calls required to complete the primitive semantics. These parameters are crucial, but the search space is relatively small. For example, there are only two options for *algorithm*, three for *protocol*, and two for *transport*.

Some prior works have provided useful guidelines for selecting these implementation parameters [12,16,47]. However, in practice, we find that these guidelines are not always reliable. For instance, some works assume that bandwidth and latency between GPUs are fixed, allowing the time cost for Tree and Ring topology to be estimated as $log_2 N$ and $2 \times (N-1)$, respectively. However, in reality, bandwidth is affected by many factors, such as message size, cluster topology, congestion, and concurrency, which makes the cost model inaccurate. As a direct consequence, in the experiment corresponding to Table 5, NCCL incorrectly selected the algorithm.

Second, the parameters NC, NT, and C correspond to resource allocation. It is well-known that communication tasks during model training require both data transmission over the network and GPU cores for operations like accumulation and averaging. These three parameters jointly determine how to utilize the network bandwidth and GPU computational power. The search space for these parameters is relatively large, with options such as 8,192 for *Chunk Size* and 128 for *Nchannel*. We further observe that analyzing their impact on performance must be based on a prior selection of the above three implementation parameters. In other words, the choice of the first three parameters provides the foundation for modeling the analysis of the latter three. Although the first three parameters can be explored exhaustively, it is practically infeasible to search through the combinations of the latter three exhaustively. Therefore, we must build a performance model for these resource-related parameters.

Table 7: Notation table

Notation	Description
M	Message size
α_i	Initialization cost of phase i
$\overline{\beta i}$	Peak bandwidth of phase i
β_i	Bandwidth of phase i
$\gamma(NC,NT,C)$	Congestion of all phases

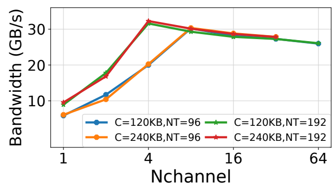
In summary, based on the classification of these parameters, we propose a subspace-based tuning method. First, we divide the entire search space into different subspaces $<A,P,T>$, each of which corresponds to a specific combination of *algorithm*, *protocol*, and *transport*. Then, within each subspace, we use a unified performance model for NC, NT, and C to determine the optimal combination of these parameters for that subspace. Finally, we compare the optimal combinations from all subspaces to identify the global optimal configuration. This approach mitigates the search space explosion for resource parameters while eliminating reliance on heuristic rules for the implementation parameters.

4.2 Modeling Resource Parameters

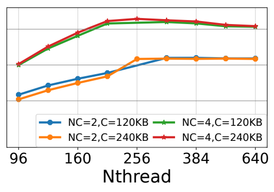
Given a subspace, we need to model three additional resource parameters. The notations required for modeling are provided in the table 7. Here, we evaluate the impact of resource parameter choices on communication bandwidth, denoted by $\beta(NC,NT,C)$. We provide qualitative analysis while avoiding difficult-to-achieve quantitative modeling. The execution of a collective primitive can be divided into two phases. In phase 0, the transport step is responsible for reading data from other GPUs and transferring it to the local buffer. Then, in phase 1, the protocol loads the data from the buffer to the SM, performs the reduction, and stores it back to the buffer. The bandwidths of these two phases are denoted as β_0 and β_1, respectively. Since the execution of the transport and protocol stages has data dependencies and is serial, as shown in Equation 1, $\beta(NC,NT,C)$ is equal to the minimum of the transport and protocol bandwidths.

$$\beta(NC,NT,C) = \min(\beta_0(NC,C), \beta_1(NC,NT)) \tag{1}$$

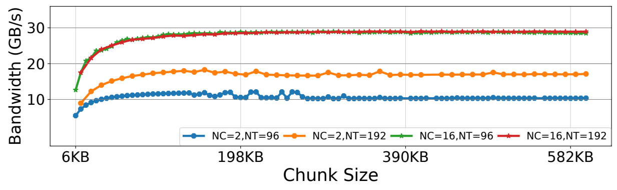
Next, we introduce the derivation process for the transport bandwidth formula in phase 0. The performance of transport depends on chunk size C and the number of concurrent messages NC, but not on NT since the transport does not include computation. We first compute the time t_0 required to transfer a message of size M across GPUs, as shown in Equation 2. $\frac{M}{NC \times C}$ represents how many serial steps the message will be



(a) Varied NC



(b) Varied NT



(c) Varied C

Figure 3: The trend of <AllGather, 80M, 8> bandwidth on a node with 8 A40 GPUs connected via NVLink

divided into, where each step consists of NC parallel transmission tasks, each of size C. The term in parentheses on the right side of the equation represents the time cost of one step. This cost consists of the initial latency α_0 and the transmission time for a step of size $NC \times C$, where $\overline{\beta_0} \times \gamma$ represents the bandwidth under the current congestion. γ is the congestion coefficient, which increases with the increase of NC, NT, and C. We only analyze it qualitatively here. Equation 3 defines bandwidth as the message size divided by the time. By solving Equations 2 and 3, we obtain the bandwidth formula for phase 0 as Equation 4.

Similar to the bandwidth derivation for phase 0, we can also obtain the estimated bandwidth formula for phase 1, as shown in Equation 5. The protocol already determines the data granularity and cache pattern, so processing performance is related to NC and NT, not to C. Due to space limitations, we omit the derivation process. According to the model, when any two parameters among NC, NT, and C are fixed and the third parameter is gradually increased, β_i will monotonically increase, approach its physical bandwidth upper limit $\overline{\beta_i}$, and then stabilize, or even fall, thereby affecting the overall bandwidth $\beta(NC, NT, C)$.

$$t_0(NC,C) = \frac{M}{NC \times C} \times (\alpha_0 + \frac{NC \times C}{\overline{\beta_0} \times \gamma}) \tag{2}$$

$$\beta_0(NC,C) = \frac{M}{t_0(NC,C)} \tag{3}$$

$$\beta_0(NC,C) = \frac{NC \times C}{\alpha_0 + \frac{NC \times C}{\overline{\beta_0} \times \gamma}} \tag{4}$$

$$\beta_1(NC,NT) = \frac{NC \times NT}{\alpha_1 + \frac{NC \times NT}{\overline{\beta_1} \times \gamma}} \tag{5}$$

To validate the aforementioned model and its characteristics, we designed a set of communication primitive experiments. By using a controlled variable method, we varied the values of different resource parameters and observed

Algorithm 1: Subspace-Directed Tuning

```
  Input: Task (w).
1 optimum ← nil
2 for subspace s ∈ $[A \times P \times T]$ do
3     config ← CoordinateDescentSearch(s)
4     if config.BwDelta(optimum) > 0 then
5         optimum ← config
6 return optimum
```

the changes in bandwidth of $< AllGather, 80M, 8-A40-NVLink >$. As expected, as shown in Figure 3, when any two parameters are fixed, the bandwidth increases first, then decreases or stabilizes as the third parameter increases. Furthermore, when the first two parameters change, the bandwidth peak (sweet point) corresponding to the third parameter also shifts accordingly. Figure 3a depicts that the optimal values of NC are 4 and 16 when $C = 80KB, NT = 96$ and $C = 20KB, NT = 96$, respectively, since a larger C results in a higher congestion factor γ, reducing the number of NC values required to reach the optimal point. Similarly, as shown in Figure 3b, the optimal values of NT are 320 and 196 when $NC = 2, C = 120KB$ and $NC = 4, C = 120KB$, respectively, since increasing either NC or NT increases γ, and with a larger NC, the NT curve reaches the sweet point earlier. Similar patterns are observed again in Figure 3c.

In summary, we find that the joint impact of NC, NT, and C on performance is not monotonic but exhibits the characteristics of a unimodal function with a sweet point. Based on this feature, we can naturally use the coordinate descent method [3] to search the resource parameter combination space. By abstracting NC, NT, and C as the three dimensions of the coordinate descent method, we can find the maximum value by continuously moving in the ascending direction along each dimension (see Section 4.3 for details).

4.3 Tuning Algorithm

Based on the parameter classification, resource-related parameter modeling, and subspace search method described above, we design an NCCL parameter tuning approach, which consists of Algorithm 1 and Algorithm 2.

Subspace-directed Tuning. The algorithm 1 uses a divide-and-conquer strategy to separately search each subspace for every communication task w. Specifically, we iterate through all the subspaces (lines 2-5). In line 3, we invoke the coordinate descent search (Algorithm 2) for subspace s to obtain the optimal configuration within that subspace. If the optimal configuration of subspace s outperforms the current best configuration, we update the best configuration (lines 4-5).

Coordinate descent search. For the input subspace s, Algorithm 2 sets M as the total number of resource parameters, representing the dimensions of tunable parameters within the subspace s (line 1). In our case, M is 3. Next, a configuration p is randomly generated, where the implementation parameters of p are determined by subspace s, and the values of the

Algorithm 2: Coordinate Descent Search

```
Input: Subspace (s).
1  M ← Number of resource parameters.
2  Randomly generate a config p from subspace s.
3  optimum ← p
4  dim, tuned_dim, lr ← 0, 0, 0.01
5  while tuned_dim ≤ M do
6      p.ProfileBw()
7      if p.BwDelta(optimum) > 0 then
8          lr ← $\frac{p.BwDelta(optimum)}{p.Bw()}$
9          optimum, tuned_dim ← p, 0
10     else
11         tuned_dim ← tuned_dim + 1
12         dim, lr ← dim + 1, 0.01
13     p ← optimum
14     p[dim] ← p[dim] + lr
15 return optimum
```

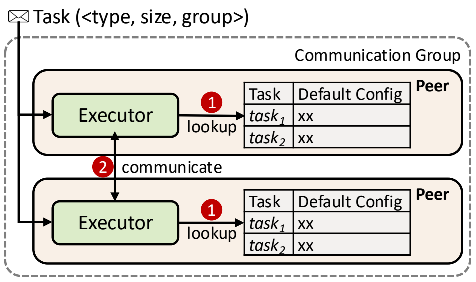
other resource parameters are randomly selected from their respective ranges (line 2). Line 4 initializes the index of the dimension to be tuned, the number of tuned dimensions, and the learning rate. Lines 5-14 execute a gradient descent search loop until all dimensions have been tuned (i.e., the number of tuned dimensions equals the total number of tunable dimensions). In line 6, configuration p is executed to profile its bandwidth. In line 7, the bandwidth of p is compared to the current *optimum*. Following this, two scenarios must be considered:

If p achieves higher bandwidth, then the configuration should be updated along the current tuning dimension. Line 8 calculates the percentage improvement in the bandwidth of p compared to the current *optimum*, using this percentage as the learning rate lr. Line 9 updates *optimum* to p and resets the tuned dimension counter, *tuned_dim*. If p does not improve bandwidth, then a new tuning dimension should be selected. Line 11 increments *tuned_dim*, and line 12 selects the next dimension to tune and resets the learning rate lr.

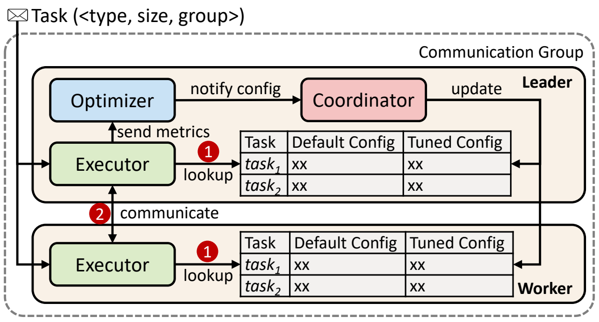
After analyzing p, lines 13 and 14 update the tuning dimension *dim* in the optimal configuration based on lr. If all dimensions have been tuned, line 15 returns the optimal configuration found within the subspace s.

5 AutoCCL: Design and Implementation

Next, we present the design and implementation details of the automated tuner, AutoCCL, focusing on how AutoCCL gathers the performance profiling data required for tuning from the training tasks and selects new configurations. To avoid the complexity of modeling computational interference and dynamic scheduling, we propose an online tuning approach. This approach leverages repetitive patterns of computation and communication cycles during the training process, using the tuning algorithm introduced in Section 4.3 for the same communication task until tuning is completed. It enables the



(a) NCCL



(b) AutoCCL

Figure 4: The architecture of NCCL and AutoCCL

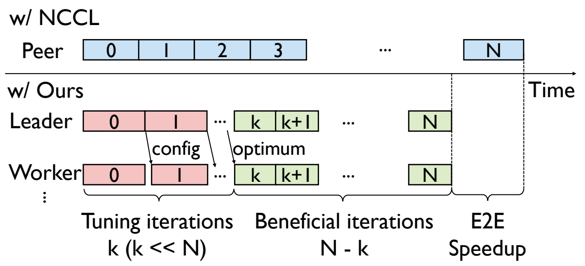


Figure 5: Online tuning workflow with AutoCCL (bottom). Rectangles with numbers represent training iterations.

collection of communication performance data affected by interference during the training process. This data is then incorporated into the tuning algorithm to capture accurately the impact of co-running computational tasks on communication. It could also amortise the cost of tuning over the early iterations.

5.1 Overall Architecture

Figure 4 illustrates the architecture of AutoCCL and its comparison with NCCL. Although AutoCCL evolved from NCCL, it exhibits significant differences from NCCL in several aspects. First, AutoCCL does not follow NCCL's peer-to-peer design. In NCCL, within each communication group executing a collective communication task, all GPUs are identical *Peers*, and each *Peer* independently generates a default configuration for the same communication task based on a deterministic cost model. Unlike this, in AutoCCL, one GPU is designated as the *Leader*, responsible for running the tuning algorithm, identifying the performant configuration, and updating the results to

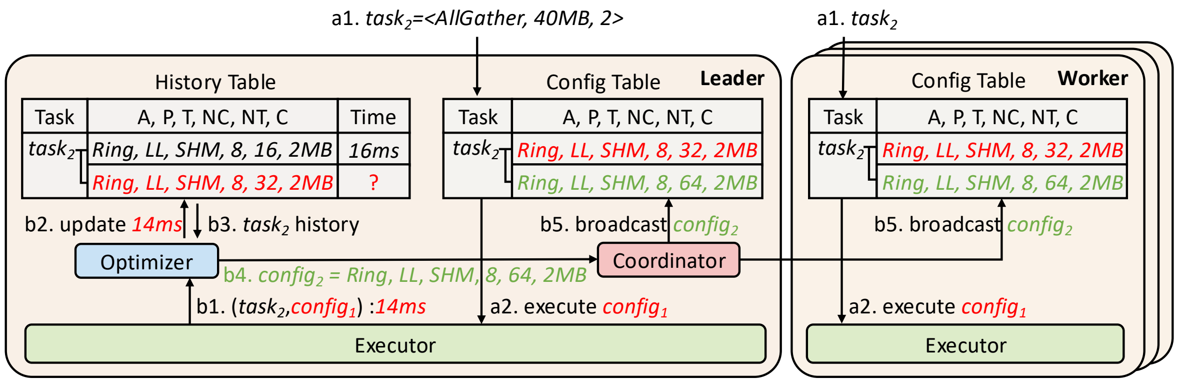


Figure 6: The online tuning workflow of AutoCCL. Colors distinguish different configurations.

the other GPUs, referred as *Workers*. Each AutoCCL *Worker* acts similarly as a *Peer* in NCCL, maintaining a configuration table. For any task, this table stores a default configuration (generated by NCCL's cost model) as well as a tuned configuration generated by the *Leader*. When a communication task is received, the *Worker*'s *Executor* checks the configuration table and selects the tuned configuration if available; otherwise, it uses the default configuration.

In contrast to the simple design of *Workers*, the logic of the *Leader* is more complex. The *Leader* introduces two additional system components: the *Optimizer* and the *Coordinator*. *Optimizer* collects performance profiling data from the *Executor*, determines whether tuning is necessary, and initiates the tuning process. Without requiring coordination with other nodes, it independently searches the performant configuration and decides whether to replace the current one. Notified by *Optimizer*, *Coordinator* then broadcasts the updated configuration to all nodes in the communication group, updating their tuned configuration. Upon success, all subsequent identical communication tasks will use the updated configuration.

5.2 Iterative Online Tuning

As shown in Figure 5, by placing the communication tuning process in the iterations of the pre-training period, we could cope with the diversity of workloads and hide tuning time cost.

We adapt the tuning algorithm for each subspace to the same communication task across the iterations. As shown in Table 1, due to the iterative nature of training and the use of microbatches, the same communication task is executed many times in each iteration. When performing a communication task, we run one step of the coordinate descent algorithm for the corresponding subspace. Upon completion, if the subspace has not reached its optimum, we save the corresponding state to facilitate the next search. Otherwise, we stop and switch to the next subspace. When the performant solution is found, the *Coordinator* broadcasts it to each node within the communication group, which guarantees the atomicity of the update and avoids the inconsistent state of the same communication task using different versions of the configuration.

Note that tuning introduces a slight overhead, which may extend the duration of each iteration. However, since the tuning process is highly efficient and the tuned iteration significantly improves performance, the end-to-end training acceleration remains substantial.

5.3 New Workflow

Here, we provide a walking-through example of the aforementioned online tuning method, illustrating the new workflow of AutoCCL, as shown in Figure 6. On the *Leader*, an additional *History Table* records the execution times of various tasks under different configurations, which the *Optimizer* uses to generate new configurations for tuning. Initially, the *Config Table* contains only the red *config_1* (<Ring, LL, SHM, 8, 32, 2MB>) for *task_2*, without green *config_2*, which will be generated during the tuning process of this example. $a1$ and $a2$ show the process of fetching and executing the current optimal configuration for the communication task, while $b1$ to $b5$ illustrate the online tuning and configuration update process.

Next, supposing both the *Leader* and *Worker* receive a new communication task with the same communication primitive, message size, and communication group members as the previously recorded *task_2*, the task is an *AllGather* operation with a message size of 40MB and a communication group size of 2. Both the *Leader* and *Worker* first need to query the *Config Table* for the target configuration (step $a1$). Since the *Config Table* is fully replicated between the *Leader* and *Worker*, both will retrieve the red *config_1* and submit *task_2* to the runtime for execution using *config_1* (step $a2$).

After a certain period, *task_2* completes execution, and the *Executor* on the *Leader* returns the execution time (14ms) to the *Optimizer* (step $b1$). The *Optimizer* updates the execution time in the *History Table* (step $b2$). The *Optimizer* then retrieves the historical execution times for *task_2* and initiates a new round of tuning (step $b3$). The *Optimizer* observes that increasing NT from 16 to 32 reduces the communication time from 16 ms to 14 ms. According to the coordinate

Table 8: DNN model statistics

Model	MBS	GBS	TP	PP	DP
Phi-2-2B	8	512	8	1	1-4
Llama-3.1-8B	2	256	8	1	1-4
Yi-1.5-34B	1	1,024	8	4	1
VGG-19-0.14B	32	32×[8,16,32]	1	1	8-32

descent method, the next attempt will also increase NT. Therefore, it selects $NT = 64$, generating *config_2*, and notifies the co-existed *Coordinator* (step *b*4). The *Coordinator* uses a broadcast operation with synchronous semantics to update *config_2* in both the *Leader* and *Worker*'s *Config Table*, replacing the original *config_1* (step *b*5). Subsequent *task_2* executions will use *config_2*, and the online tuning process will continue until the optimal configuration is found.

5.4 Implementation Details

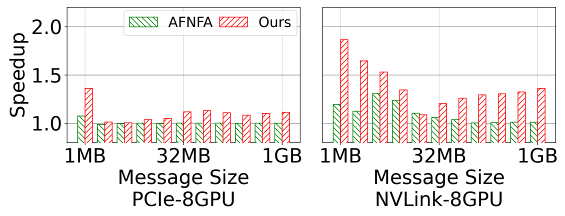
`AutoCCL` is implemented based on NCCL 2.18.3, with 9,176 lines of C++ code, supporting collective communication primitives such as *AllGather*, *Allreduce*, and *ReduceScatter*. We modified NCCL's configuration generation module and transport initialization module to allow more flexible use of various configurations. For any communication group, `AutoCCL` automatically launches an additional thread as the *Leader* on one node to perform communication task tuning. Due to the asynchronous execution characteristics of CUDA kernels, we start an additional thread to measure execution time.

`AutoCCL` is implemented based on the C++ standard library, the *Coordinator* is implemented based on Linux sockets, and no additional libraries are required. Thereby, users can seamlessly migrate from NCCL to `AutoCCL` by preloading the dynamic library without requiring any code modification. Since `AutoCCL` and NCCL share identical interfaces, training frameworks such as PyTorch [41] and MegatronLM [51] can directly integrate `AutoCCL` without modification, and training tasks running on these frameworks are unaware of the underlying switch. Our code is open-sourced at [2].

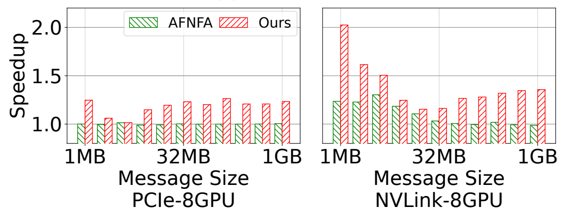
6 Evaluation

6.1 Experimental Setup

Cluster setup. We use two clusters to demonstrate the generality of our tuner. Cluster A has 2 nodes, interconnected via 2 pairs of 400 Gbps InfiniBand-PCIe 5.0 NIC. Each node has 8 NVIDIA Ampere A40 GPUs (48 GB GDDR6, PCIe 4.0), an AMD EPYC 9654 CPU, and 64GB of host memory. The 8 GPUs within a single node are divided into 4 pairs, with each pair of GPUs connected via 400 Gbps NVLink interconnects. Cluster B contains 4 machines in a single rack, connected via a 100 Gbps InfiniBand network. Each machine has 8 NVIDIA Ampere A40 GPUs (the intra-node GPUs are connected via PCIe 4.0), an Intel Xeon Gold 5320 CPU, and 504 GB of host memory. In Cluster B, due to hardware limitations, peer-to-peer communication is not available between some GPUs. Both clusters operate under the same environment with CUDA v12.1 and NVIDIA driver v470.63.01.



(a) *AllGather*



(b) *ReduceScatter*

Figure 7: Bandwidth speedups for the *AllGather* and *ReduceScatter* communication of different A40 clusters

DNN training jobs. We evaluate four models from different application scenarios, including Phi-2-2B [21], Llama-3.1-8B [54], Yi-1.5-34B [65], and VGG-19 [52] shown in Table 8, where three are large language models with up to 32 billion parameters and one computer vision model. Model training is conducted using PyTorch version 2.1.0. Various parallelisms are managed by MegatronLM. Due to limited GPU memory, Llama-3.1-8B combines *distributed optimizer* [45] along with data parallelism. We set the micro-batch size as large as possible under the GPU memory limitation for better hardware utilization. The global batch size follows the setting of the GPT series with a similar model scale [11].

Baseline systems. We compare against NCCL v2.18.3-1, almost the latest version provided by NVIDIA, and internally performing tuning to some extent. We always enable its tuning function. We also include an academic collective communication tuning system, AFNFA [64]. We use the random forest algorithm to train 1% of the offline sampled data and use environment variables to set the configuration for different communication to reproduce the results of AFNFA. All data represent the average results from multiple experiments.

6.2 Communication Micro-Benchmarks

We explore how `AutoCCL` improves different collective communication primitives, considering various hardware setups and message transmission sizes. We first evaluate the communication performance without computational interference, followed by experiments with interference.

Communication without computational interference. We first consider *AllGather* and *ReduceScatter* primitives, which are heavily used in *Tensor Parallelism*. We test them with 8 GPUs within a single machine, connected via PCIe 4.0 or NVLink. Figure 7 summarizes the bandwidth speedups of `AutoCCL` and AFNFA over the native NCCL.

For A40-PCIe, AFNFA's results are almost identical

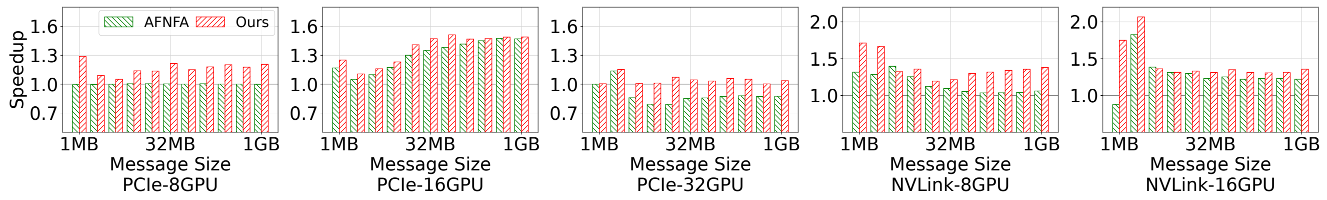


Figure 8: Bandwidth speedups for the *AllReduce* communication of A40 clusters with different setups

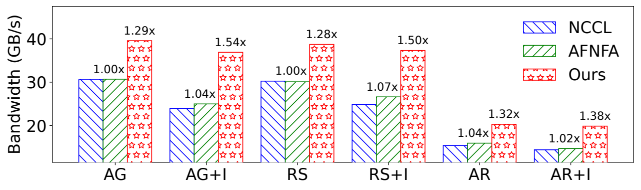


Figure 9: Comparison of bandwidth with and without computation **interference** for different communication. The message size is 128 MB. '+I' refers to the communication that interfered with computation. The speedup numbers are annotated in the figure. *AG*, *RS* and *AR* stand for *AllGather*, *ReduceScatter* and *AllReduce* respectively.

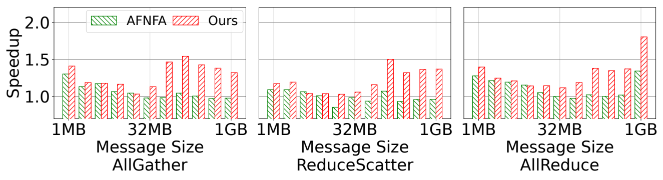


Figure 10: Bandwidth speedups with increasing message sizes and computation **interference** for different communication types of a cluster with 8 A40 connected with NVLink

to NCCL, offering little optimization. However, AutoCCL achieved an average bandwidth improvement of 22.66% and 27.52% for *AllGather* and *ReduceScatter*, respectively, compared to NCCL. This is especially evident for larger message sizes, where AutoCCL outperforms both baselines.

It is widely acknowledged that NVLink is highly efficient, and NCCL has been heavily optimized for it. Despite this, AutoCCL demonstrates even greater advantages on NVLink compared to PCIe. For instance, for various message sizes, the average bandwidth speedups are 1.38× and 1.39×, respectively. In contrast, AFNFA shows far fewer benefits, particularly for large message sizes, where its tuning results still align with those of NCCL since it uses global configuration and cannot adapt to different message sizes.

We observe that in a few cases, AutoCCL's tuning results align with those of NCCL. This occurs because, for these particular message sizes, NCCL's configuration is already optimal. In other words, its standard α-β network model can accurately predict performance at these points, although this model does not generalize to all cases.

Next, we analyze *AllReduce*, which is commonly used in data parallelism to exchange gradient information. As shown in Figure 8, similar to the results above, AutoCCL achieved average bandwidth speedups of 1.28× and 1.15× compared to NCCL and AFNFA, respectively. In the PCIe-32GPU cluster, where AFNFA even results in negative optimization, AutoCCL slightly outperforms NCCL since the default configuration of NCCL is effective. However, NCCL becomes less suitable under computational interference.

Communication with computational interference. We test the behavior of AutoCCL when communication and computation are co-executed. We evaluate the above three primitives with a GEMM operation running concurrently alongside each primitive. The matrices of GEMM are A[m, k] and B[k, n], where m, k, and n are 3,456, 128, and 3,456. Figure 9 compares the bandwidth of these primitives with and without computational interference for a fixed message size.

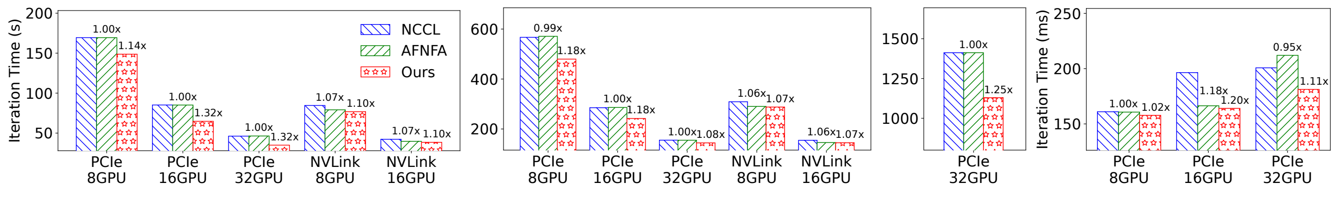
Similar to previous studies, the bandwidth of the three primitives decreases significantly under computational interference. With AutoCCL tuning, the bandwidth speedups compared to NCCL are 1.29×, 1.50×, and 1.38×, respectively. In contrast, AFNFA demonstrates poor performance, showing no improvements for *AllGather* and *ReduceScatter* and merely a 1.02× improvement for *AllReduce*. Interestingly, after AutoCCL tuning, the optimal bandwidth under interference closely approximates the optimal bandwidth without interference. This indicates that AutoCCL can accurately predict performance even with interference.

Figure 10 differs from Figure 9 by increasing the message size for primitives while simultaneously scaling up the corresponding co-executed GEMM to simulate varying degrees of resource sharing and interference.

For *AllGather* and *AllReduce*, AFNFA only shows tuning benefits for message sizes below 16 MB, compared to NCCL. In contrast, AutoCCL not only matches or slightly outperforms AFNFA for small message sizes, but also achieves significant bandwidth improvements of 1.11-1.76× and 1.16-1.39× for larger message sizes (e.g., those exceeding 32 MB) compared to NCCL, respectively. For *ReduceScatter*, AFNFA either performs on par with NCCL or results in negative optimization. This indicates that in more competitive scenarios, both NCCL and AFNFA fail to deliver satisfactory performance, as their modeling approaches are insufficient to handle complex environments and dynamic changes. In contrast, AutoCCL shows a distinct performance advantage in these situations.

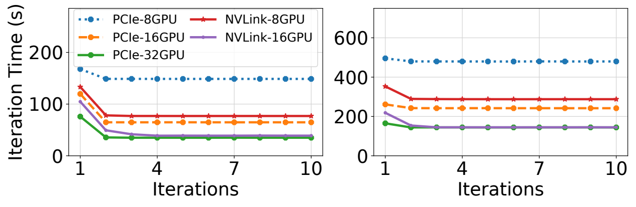
6.3 End-to-End Performance

To show how AutoCCL accelerates the training progress for real-world models, we conduct experiments to train models



(a) Phi-2-2B (b) Llama-3.1-8B (c) Yi-1.5-34B (d) VGG-19-0.14B

Figure 11: Training iteration time for different models between NCCL, AFNFA, and `AutoCCL`, with speedup annotated



(a) Phi-2-2B (b) Llama-3.1-8B

Figure 12: Rapid convergence of `AutoCCL` tuning at early iterations of end-to-end LLM training

on different communication systems under various hardware setups. Figure 11 shows the training iteration time of different models. Among all the hardware setups, `AutoCCL` outperforms both NCCL and AFNFA in training these models, while AFNFA is slower than NCCL in some cases as its offline-tuned configurations degrade performance under computational interference. On the PCIe machine, we observed a greater performance improvement compared to microbenchmark results because our online tuner exhibits stronger resilience to such interference than NCCL and AFNFA, whose performances are significantly degraded. The end-to-end gains on NVLink machines are modest because in scenarios where communication overlaps with computation and computation is dominant, excessive optimization of communication can slow down computation and reduce overall performance. For best cases, `AutoCCL` can improve the training iteration time by more than 32% compared to both NCCL and AFNFA.

6.4 Efficiency of Iterative Online Tuning

To show the efficiency of the `AutoCCL` to find the performant configuration with iterative online tuning, we measure how the iteration time changes when training the models, as shown in Figure 12. As noted in Table 1, because of the repetition of layers in the model and the large number of repeated micro batch sizes in each global batch size, the Transformer models like Llama-3.1-8B, Phi-2-2B, and Yi-1.5-34B can generate thousands to tens of thousands of repetitive communication operations per iteration. For these models, only several iterations are required for `AutoCCL` to find the performant configurations. For smaller models like VGG-19, although less repetitive communication operations are involved in each iteration, the time consumed per iteration is relatively short, e.g., 150-200 ms per iteration for VGG-19 in Figure 11d, which is around 1,000 times shorter than that of Llama-3.1-8B. In our evaluation, it only takes no more than 10 minutes for `AutoCCL` to find the performant configuration in VGG-19. In summary, for these four models of different areas, `AutoCCL` can rapidly identify a sufficiently performant configuration within just a few iterations or minutes through online tuning, highlighting its ability to adapt quickly to new environments, which is infeasible for offline tuning strategies.

7 Related Work

Computation-communication scheduling. At the application level, many studies [13, 19, 22, 23, 31, 46, 50, 59, 60, 67] aim to reduce communication overhead by overlapping computation and communication, thereby enhancing training efficiency. For example, Horovod [46] overlaps all-reduce communication with backward computation to improve performance. `AutoCCL` complements these methods by additionally considering the interference between computation and communication during the tuning process, making it more effective for tasks with specific scheduling requirements.

Communication compression. Other approaches [8, 9, 53, 56, 62, 63] improve communication efficiency by exploiting data sparsity. HiPress [9] introduces a framework to efficiently compress gradient data in *AllReduce*, reducing latency. ZeRO++ [56] combines quantization with collectives of *AllReduce* and *AllGather* in the ZeRO optimization to improve efficiency. `AutoCCL` is complementary to them, as it can be used to further enhance the performance of underlying communication collectives for these compression algorithms.

Collective algorithm generation. At the algorithm level, several studies focus on developing topology-aware [12, 15, 16, 27, 29, 32, 35, 42, 42, 47, 57] or sparse-aware [10, 28] communication algorithms. For instance, Swing [16] proposes a new *AllReduce* algorithm to minimize hops in torus networks, while MCCLang [15] offers a domain-specific programming interface and efficient compiler for easier algorithm customization. Unlike these works, `AutoCCL` does not introduce new algorithms but treats the algorithm choice as a tunable parameter. In the future, `AutoCCL` can be integrated with these approaches to further enhance runtime performance.

Network tuning. At the hardware level, to achieve higher network communication bandwidth, NCCL [5] uses the Alpha-Beta model to predict communication efficiency based on hardware topology and bandwidth. AFNFA [64] improves this prediction using machine learning with offline profiling.

However, all these approaches share a global configuration (i.e., through NCCL environment variables [6]) and do not consider computational interference in real-world workloads. In contrast, `AutoCCL` employs online profiling to capture the interference effects and dynamically selects a performant configuration for each collective communication operation, improving performance across diverse workloads.

8 Discussion

8.1 Parameter Selection and Extensibility

Once the implementation-wise parameter T is determined, NCCL allows further optimization of transport-specific parameters, including *Ethernet*, *IB*, *COLL_SHARP*, *NVLS*, *SHM*, and *P2P*. AFNFA considers *NCCL_SOCKET_NTHREADS* for Ethernet transport and *NCCL_NET_GDR_LEVEL* for IB transport. However, our study does not include these parameters, as they introduce a risk of failures while providing limited performance gains (Section 8.2). AFNFA attempts to optimize NCCL including *NCCL_ALGO* for A, *NCCL_SHM_DISABLE* and *NCCL_P2P_LEVEL* for T, NCCL_MAX/MIN_NCHANNEL for NC, and *NCCL_BUFFSIZE* for C. These tunable combinations are incorporated into our search space. Our search space also includes P and NT, whose importance is demonstrated in Section 3.1.

The subspace coordinate descent method is inherently scalable, allowing users to optimize transport-specific parameters. Introducing transport-specific parameters only increases the number of implementation-wise parameter combinations, effectively subdividing the existing subspaces into finer-grained subspaces. The scalability of the subspace coordinate descent method enables users to further optimize transport-specific parameters as needed.

Our findings suggest that using default or fixed values for transport-specific parameters already yields robust performance. For example, we attempted to optimize IB-specific parameters and found that the default values of *NCCL_IB_SPLIT_DATA_ON_QPS* and *NCCL_IB_QPS_PER_CONNECTION* are already optimal. Similarly, the optimal value for *NCCL_NET_GDR_LEVEL* is typically determined based on the NIC and GPU topology, making it effective across different communication workloads. Since the optimal values for transport-specific parameters are often hardware-dependent, further exploration of their optimization is left for future work.

8.2 Task Failures and Valid Configurations

Tuning NCCL parameters can potentially lead to training task failures. This issue has been reported in related studies [40, 64] and confirmed by our experiments. As previously mentioned, once an implement-wise parameter such as T is determined, collective communication will employ a specific transport mechanism, such as P2P transport. NCCL allows further optimizations for specific transport mechanisms—for instance, enabling *P2P_USE_CUDA_MEMCPY* for P2P transport can improve bandwidth. However, in end-to-end training tasks, this setting may result in deadlocks. Additionally, we observed another type of failure: for resource-allocation parameters such as NC, NT, and C, excessively large values can lead to resource saturation, ultimately causing program crashes.

Nevertheless, failures induced by tuning have minimal impact on our system for two reasons. First, transport-specific optimizations are beyond our scope; therefore, we inherently avoid failures associated with them. In the future, we may explore further optimizations under failure recovery support. Second, for NC, NT, and C, due to the unimodal nature of the performance function, excessively large values do not yield performance gains. Therefore, we can impose upper bounds on resource usage without sacrificing performance, effectively preventing failures.

Besides, if a failure occurs, online tuning introduces additional overhead for loading checkpoints compared to offline tuning. However, given the numerous recent optimizations in checkpointing [18, 26, 61], this overhead has been significantly reduced and is negligible. The dominant cost of failure recovery remains container rebooting, which is required for both offline and online tuning.

Finally, failure handling and recovery are orthogonal to tuning. We can explore this topic in future work.

9 Conclusion

`AutoCCL` is an automated tuning tool that optimizes low-level NCCL parameter selection. By decoupling implementation-related parameters from those that inflate the search space, our divide-and-conquer approach reduces the necessity for extensive trial runs. Furthermore, its online tuning strategy efficiently handles communication-computation interference. Evaluation across multiple clusters and models demonstrates that `AutoCCL` significantly enhances both communication and end-to-end training performance compared to NCCL and another SOTA tuner.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Rui Miao, for their insightful comments. This work is supported in part by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No.: XDB0660101, XDB0660000, and XDB0660100, the National Key R&D Program of China under Grant No. 2024YFB4505701, and the University Synergy Innovation Program of Anhui Province under Grant No. 2024YFB4505701. We thank the computing resources and technical support from Institute of Artificial Intelligence, Hefei Comprehensive National Science Center. We thank the computing resources and technical support from China Telecom Co., Ltd. Hefei Branch. Cheng Li is the corresponding author.

References

[1] AMD. ROCm communication collectives library (RCCL). https://github.com/ROCm/rccl, 2024. [Accessed 19-09-2024].

[2] Automated tunable collective communication library (AutoCCL). https://github.com/gbxu/autoccl, 2024. [Accessed 19-09-2024].

[3] Coordinate descent. https://en.wikipedia.org/wiki/Coordinate_descent, 2024. [Accessed 19-09-2024].

[4] General Matrix Multiplication. https://en.wikipedia.org/wiki/Matrix_multiplication, 2024. [Accessed 19-09-2024].

[5] NVIDIA Collective Communication Library (NCCL). https://github.com/nvidia/nccl, 2024. [Accessed 19-09-2024].

[6] NVIDIA Collective Communication Library (NCCL) Documentation. https://docs.nvidia.com/deeplearning/nccl/archives/nccl_2183/user-guide/docs/env.html, 2024. [Accessed 19-09-2024].

[7] Sigmoid Function. https://en.wikipedia.org/wiki/Sigmoid_function, 2024. [Accessed 19-09-2024].

[8] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. On the utility of gradient compression in distributed training systems. *Proceedings of Machine Learning and Systems*, 4:652–672, 2022.

[9] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. Gradient compression supercharged high-performance data parallel dnn training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 359–375, 2021.

[10] Charles Block, Gerasimos Gerogiannis, Charith Mendis, Ariful Azad, and Josep Torrellas. Two-face: Combining collective and one-sided communication for efficient distributed spmm. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1200–1217, 2024.

[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[12] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 62–75, 2021.

[13] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 178–191, 2024.

[14] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[15] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. Mscclang: Microsoft collective communication language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 502–514, 2023.

[16] Daniele De Sensi, Tommaso Bonato, David Saam, and Torsten Hoefler. Swing: Short-cutting rings for higher bandwidth allreduce. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1445–1462, 2024.

[17] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[18] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. Just-in-time checkpointing: Low cost error recovery from deep learning training failures. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1110–1125, 2024.

[19] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *Proceedings of Machine Learning and Systems*, 1:418–430, 2019.

[20] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[21] Alyssa Hughes. Phi-2: The surprising power of small language models — microsoft.com. https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models/. [Accessed 19-09-2024].

[22] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 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*, ASPLOS '22, page 402–416, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. *Proceedings of Machine Learning and Systems*, 1:132–145, 2019.

[24] Xianyan Jia, Le Jiang, Ang Wang, Jie Zhang, Xinyuan Li, Wencong Xiao, Yong Li, Zhen Zheng, Xiaoyong Liu, Wei Lin, et al. Whale: Scaling deep learning model training to the trillions. *arXiv preprint arXiv:2011.09208*, 2020.

[25] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[26] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. Megascale: Scaling large language model training to more than 10,000 gpus. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

[27] Heehoon Kim, Junyeol Ryu, and Jaejin Lee. 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*, pages 999–1015, 2024.

[28] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[29] Sabuj Laskar, Pranati Majhi, Sungkeun Kim, Farabi Mahmud, Abdullah Muzahid, and Eun Jung Kim. Enhancing collective communication in mcm accelerators for deep learning training. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–16. IEEE, 2024.

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

[31] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. Accelerating distributed moe training and inference with lina. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 945–959, 2023.

[32] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, 2014.

[33] Shengwei Li, Kai Lu, Zhiquan Lai, Weijie Liu, Keshi Ge, and Dongsheng Li. A multidimensional communication scheduling method for hybrid parallel dnn training. *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[34] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, et al. nnscaler: Constraint-guided parallelization plan generation for deep learning training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 347–363, 2024.

[35] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud. *Proceedings of Machine Learning and Systems*, 2:82–97, 2020.

[36] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 937–954, 2020.

[37] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[38] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.

[39] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[40] Lichen Pan, Juncheng Liu, Jinhui Yuan, Rongkai Zhang, Pengze Li, and Zhen Xiao. Occl: a deadlock-free library for gpu collective communication. *arXiv preprint arXiv:2303.06324*, 2023.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[42] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

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

[44] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International conference on machine learning*, pages 18332–18346. PMLR, 2022.

[45] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[46] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[47] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. Taccl: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 593–612, 2023.

[48] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

[49] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[50] Shaohuai Shi, Xinglin Pan, Qiang Wang, Chengjian Liu, Xiaozhe Ren, Zhongzhe Hu, Yu Yang, Bo Li, and Xiaowen Chu. Schemoe: An extensible mixture-of-experts distributed training system with tasks scheduling. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 236–249, 2024.

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

[52] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[53] Jaeyong Song, Jinkyu Yim, Jaewon Jung, Hongsun Jang, Hyung-Jin Kim, Youngsok Kim, and Jinho Lee. Optimus-cc: Efficient large nlp model training with 3d parallelism aware communication compression. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 560–573, 2023.

[54] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[55] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. Unity: Accelerating dnn training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, 2022.

[56] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Xiaoxia Wu, Connor Holmes, Zhewei Yao, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, et al. Zero++: Extremely efficient collective communication for large model training. In *The Twelfth International Conference on Learning Representations*, 2023.

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

[58] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[59] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, et al. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 93–106, 2022.

[60] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. Topoopt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, 2023.

[61] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 364–381, 2023.

[62] Zhuang Wang, Haibin Lin, Yibo Zhu, and TS Ng. Espresso: Revisiting gradient compression from the system perspective. *arXiv preprint arXiv:2205.14465*, 2022.

[63] Zhuang Wang, Haibin Lin, Yibo Zhu, and TS Eugene Ng. Hi-speed dnn training with espresso: Unleashing the full potential of gradient compression with near-optimal usage strategies. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 867–882, 2023.

[64] Zibo Wang, Yuhang Zhou, Chen Tian, Xiaoliang Wang, and Xianping Chen. Afnfa: An approach to automate nccl configuration exploration. In *Proceedings of the 7th Asia-Pacific Workshop on Networking*, pages 204–205, 2023.

[65] Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Guoyin Wang, Heng Li, Jiangcheng Zhu, Jianqun Chen, et al. Yi: Open foundation models by 01.ai, 2024.

[66] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. *arXiv preprint arXiv:2201.12023*, 2022.

[67] Yonghao Zhuang, Lianmin Zheng, Zhuohan Li, Eric Xing, Qirong Ho, Joseph Gonzalez, Ion Stoica, Hao Zhang, and Hexu Zhao. On optimizing the communication of model parallelism. *Proceedings of Machine Learning and Systems*, 5, 2023.