xCCLTuner: Treating xCCL as Black-Box and Automatically Tuning

Chenxu Wang[1], Wentao Fan[2], Zhehao Lin[1], Peirui Cao[1] Xi Lin[1], Xiaohu Xu[2], Wanchun Dou[1], Guihai Chen[1], Chen Tian[1]
[1]State Key Laboratory for Novel Software Technology, Nanjing University
[2]China Mobile (Suzhou) Software Technology Co. Ltd., Suzhou, 215000, China

IEEE INFOCOM 2026 - IEEE Conference on Computer Communications | 979-8-3315-4961-9/26/$31.00 ©2026 IEEE | DOI: 10.1109/INFOCOM59046.2026.11571473

***Abstract*—The collective communication libraries (xCCL) are an important networking support for distributed training systems. Optimizing xCCL is crucial for reducing training costs while maintaining accuracy, but existing approaches often require extensive manual intervention, lack user-friendliness, and struggle to adapt to complex environments. Therefore, we introduce xCCLTuner, an out-of-the-box tuning system without needing to categorize or model parameters that automatically optimizes xCCL parameters for distributed training systems. xCCLTuner utilizes Bayesian optimization to navigate the high-dimensional parameter space of xCCL knobs in a black-box manner. It overcomes key challenges through three main designs: (1) Low-dimensional tuning to reduce the complexity of the parameter space, (2) xCCL-specific lookup table to identify and skip fatal parameter combinations, and (3) Bucketization techniques to conquer vast parameter domains. xCCLTuner rapidly delivers optimized training times across various topologies, showcasing its efficiency and ease of use. Our experiments demonstrate that xCCLTuner achieves up to a 5.87X improvement in throughput.**

***Index Terms*—Collective Communication Libraries, Tuning.**

I. Introduction

As contemporary DNN models continue to grow in size, distributed machine learning techniques have become widely applied. Distributed DNN training sequentially depends on deep learning training frameworks, collective communication libraries (xCCL [1]), network protocol stacks, and hardware resources, forming a critical path. Within this path, xCCL play a crucial role. These libraries define the implementation of collective communication primitives, supporting various parallel training paradigms in the upper layers while accommodating specific hardware and network features in the lower layers.

The primary objective of deep learning engineers is to minimize training costs while maintaining training accuracy within their specific training environments. Generally, minimizing training time within a given environment leads to greater cost efficiency. To this end, recent research has begun to focus on optimizing xCCL. Some studies aim to design improved algorithmic implementations for specific collective communication primitives [2]–[7], while others design topology-aware collective communication algorithms or optimize routing for communication traffic [8]–[11].

This paper is funded by the NSFC No. 62325205 and U25B2035, the Nanjing University-China Mobile Comunications Group Co.Ltd. Joint Institute. Corresponding author: Peirui Cao.

In practice, the aforementioned optimization approaches are time-consuming due to the high degree of human involvement, requiring engineers to intrusively modify xCCL code, alter critical path dependencies, and utilize experience or simulations to obtain topological details to achieve improvements in training performance. Given that existing methods lack user-friendliness and adaptivity to new training environments and specific training tasks, engineers often find themselves unable to leverage these optimization techniques effectively given the limited timeframe of available training resources. Furthermore, we posit that as upstream frameworks and strategies in the critical path continue to evolve, and downstream software and hardware systems upgrade, network speeds increase, the number of hosts and links increase [12], [13], and interconnect topologies advance, xCCL will become a bottleneck in the critical path if optimization approaches remain unchanged.

In summary, the primary contradiction lies between the increasing demand from deep learning engineers for out-of-the-box training optimization solutions in given environments and the inherent difficulty of optimization in intricate environments. We aim to provide users with a one-click solution that optimizes performance for their current training without requiring any code modifications to xCCL.

xCCL inherently maintains a series of tunable knobs that allow users to customize the implementation of communication primitives, allocate resources, and enable hardware features, thereby controlling various aspects of xCCL's behavior. Our case studies in §III-A demonstrate that fine-tuning these knobs can lead to performance improvements in specific training environments. However, simultaneously tuning these knobs has been largely overlooked in previous optimization efforts.

This paper introduces xCCLTuner, an out-of-the-box tuning system designed to adaptively optimize xCCL knobs for a given training environment. xCCLTuner employs Bayesian optimization to explore and fit the parameter space composed of all xCCL knobs in a black-box manner. The system ultimately outputs a set of knob combinations that users can configure with a single click, offering optimized performance for specific tasks in the current training environment.

To achieve its objectives, xCCLTuner's design faces three challenges: (1) The parameter space, comprising dozens of parameters, invokes the curse of dimensionality, proving over-

whelming not only for human engineers but also for algorithms and many models that struggle to cope effectively. (2) When considering multiple xCCL knobs that may have implicit interdependencies, an incorrect value for one knob can render the entire knob combination invalid, potentially affecting multiple iterations of sampling that include this fatal value. (3) Some numerical xCCL knobs have extremely vast domains, reaching up to billions of possible values. xCCLTuner overcomes these challenges through the following key design features.

Low Dimensional tuning for large space (§V-A). To address challenge 1, xCCLTuner employs the HeSBO dimensionality reduction method, which provides error bound guarantees. The process proceeds as follows: 1) It utilizes HeSBO to define a mapping from the original parameter space to an artificial low-dimensional parameter space. 2) Sampling is conducted within this low-dimensional parameter space. 3) The sampled points are inverse-mapped back to the original parameter space for testing, where corresponding performance data is collected.

xCCL-specific lookup table to tackle failure (§V-B). To prevent the optimizer from becoming trapped by challenge 2, we implement an online xCCL-specific lookup table mechanism, which continuously monitors the sampling process, promptly detecting and skipping options that contain fatal values through the xCCL-specific lookup table.

Bucketization to conquer vast domains. (§V-C). For challenge 3, we propose bucketization methods, which enhance xCCLTuner's performance across these extensive parameter domains based on our observations of these parameters.

xCCLTuner enhances throughput by up to 5.87× across various scenarios, showcasing its efficiency and ease of use in reducing training costs while maintaining accuracy.

II. Background

A. Collective communication in Distributed Training

The communication behavior in distributed machine learning applications is abstracted into a class of operations known as *collective communication primitives*, which describe how data is aggregated or scattered among multiple GPUs across multiple nodes. The selection of communication primitives is typically determined by a combination of factors, including the model architecture, training phase, and parallelism strategy. For instance, when employing DP (Data Parallelism) [14], the *AllReduce* primitive is used for the gradient averaging phase. In the case of TP (Tensor Parallelism) [15], the *ReduceScatter* primitive is employed in the gradient update phase. For DLRM (Deep Learning Recommendation Model) [16], the *AllToAll* primitive is used to shuffle the lookup table. It is important to note that a complete training process may involve multiple primitives, each serving different communication requirements at various stages of the distributed training workflow.

xCCL implements these communication primitives. NCCL (NVIDIA Collective Communication Library) is designed for communication between NVIDIA GPUs within single or multiple nodes, supporting hardware, e.g., PCIe, NVLink/NVSwitch, as well as networks using InfiniBand or TCP/IP. In addition to NCCL, there are various other collective communication libraries, including AMD's RCCL [17], Intel's oneCCL [18], Microsoft's MSCCL [19], and Alibaba's ACCL [20]. All of them adhere to NCCL's specifications, providing a series of tunable knobs that can be configured through environment variables. Next, we will use NCCL as an example to illustrate these concepts.

B. Tunable Knobs in xCCL

TABLE I
Example Subset of NCCL Knobs

Knob	Domain
ALGO	Tree, Ring, NVLS, NVLSTree
PROTO	LL, LL128, Simple
P2P_DISABLE	Enable, Disable
P2P_LEVEL	LOC, NVL, PIX, PXB, PHB, SYS
SHM_DISABLE	Enable, Disable

xCCL exposes these knobs to users, allowing them to adjust various behaviors of the collective communication library. Table I presents a subset of NCCL knobs and their respective domains. The *NCCL_ALGO*[1] knob determines the implementation algorithm for primitives. For instance, a ring-based *AllReduce* algorithm implements data transfer between GPUs following a ring pattern. *PROTO* determines the encoding for data transfer between memory and GPUs. In *LL*, flag bits occupy 50% of the total size, allowing the receiver to recognize data reception immediately upon receiving the flag, ideal for small messages. However, for larger messages, half of the bandwidth would be used for transmitting flags. Similarly, *LL128* uses 5% of the total size for flag bits, making it suitable for medium-sized messages, while the *Simple* protocol is optimal for large messages. *P2P_DISABLE* and *P2P_LEVEL* control the use of PCIe, QPI, and NVLink to support CUDA direct access. *SHM_DISABLE* regulates the use of shared memory for communication when P2P is disabled.

The number of knobs varies slightly between versions. In the latest version, NCCL 2.23, there are 108 adjustable parameters. When training frameworks invoke xCCL to create a communication primitive process for communication, this process inherits pre-configured knobs (in the form of key-value pairs) from the system environment variables.

III. Motivation

A. Tuning xCCL Knobs Does Matter

NCCL provides a default knob configuration for the current training environment based on a cost model built on Nvidia machines. However, this cost model is not generalizable, and the configurations it provides are suboptimal in many training environments. To illustrate this point, we present two examples demonstrating performance improvement through tuning. We use a topology comprising two nodes, each equipped with 2 CPUs, 4 Nvidia P100 GPUs, and two 100 Gbps Mellanox NICs as well as run the NCCL Test to evaluate the performance of the *AllReduce* on a single machine. By adjusting the

[1]For simplicity, we omit the "NCCL" prefix from knob names in the following context.

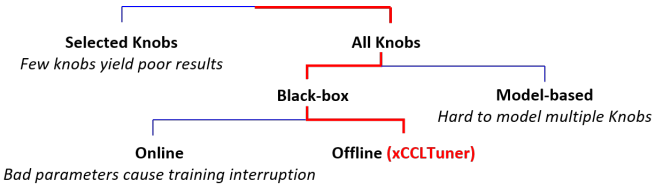

Fig. 1. Tuner methodology.

P2P_LEVEL to shift traffic from the default QPI bus (orange lines in the figure) to the RDMA NICs, we achieve a $1.54\times$ increase in bidirectional bandwidth. We then train DLRM on two machines. The default configuration provided by NCCL's cost model utilizes 4 tree channels and 4 ring channels. By adjusting the *MAX_NCHANNEL* knob to limit the number of channels to 1 ring channel, we achieve a $1.42\times$ increase in bandwidth while simultaneously reducing the training time to 89% of the original duration.

Tuning xCCL knobs holds significant potential for performance optimization. In fact, several intuition-based empirical guidelines exist, such as: *P2P* performance is superior to *SHM*, and it is beneficial to enable the highest possible level of *P2P*; increasing channels and threads generally yields better results; for large messages, it is advisable to use a larger chunk size; for large-scale topologies, tree algorithms are preferable. However, we need to address two critical questions: **Is tuning a trivial process, and are these empirical guidelines sufficient? If not, how should we approach the tuning process?** We answer these questions using specific examples, gradually elucidating our tuning methodology shown in Figure 1.

B. Black-box Tuning of All knobs vs. Model-based Tuning of Selected knobs

The aforementioned empirical rules represent a method of analyzing and modeling knobs to provide tuning choices, typically applicable only when dealing with a small number of knobs. The knobs mentioned in these rules are among the simplest and most intuitively meaningful of NCCL's tunable knobs. Their relationship with performance is approximately monotonic within a certain range. Furthermore, these knobs are relatively independent (i.e. the correlation among them is weak). Expanding the number of knobs to consider and modeling more complex knobs become challenging when attempting to identify overall patterns in a larger knob set. The following example demonstrates that tuning only a subset of knobs results in performance that still exhibits a significant gap compared to the optimal performance achievable.

We test the AllReduce performance on a topology of 4 nodes with 16 Nvidia V100 GPUs interconnected via 10Gbps links. Table II presents the performance of the default configuration S_0, our optimal configuration set S^* (containing 33 key-value pairs), and configurations derived from S^* by only keeping a subset of parameters. S_1 represents all parameter subsets derived from S^* by selecting a single parameter and its value (containing 1 key-value pair), while S_2 represents parameter subsets obtained by removing one parameter from S^* (containing 32 key-value pairs). That is, S_1 and S_2 each contains 33 different parameter subsets, and we test these parameter subsets separately to present the average and best performance of the subsets. The best performance of S_1 is only 85% of S^*, and the best performance of S_2 is 87% of S^*. Even missing just one parameter results in a significant performance decrease.

TABLE II
PERFORMANCE OF VARIOUS PARAMETER COMBINATIONS.

Knobs set	Average Bw (GB/s)	Best Bw (GB/s)
S_0	0.54	*
S^*	1.40	*
S_1	0.65	1.20
S_2	1.10	1.23

This shows that to achieve better results, we need to consider as many knobs as possible. Even if we could provide modeling and analysis for a larger number of parameters, we cannot guarantee that such modeling would be general. The training example presented in §III-A achieved better performance by limiting the number of channels to 1, contradicting general experience. NCCL's own cost model is another example of a non-generalizable approach. Moreover, modeling and tuning only a subset of parameters while disregarding the impact of remaining parameters on the external system can lead to errors. In the aforementioned example, S^* increased the *MIN_CHANNEL* number to 2 to enhance parallelism. However, further expansion results in excessive shared memory usage, leading to performance degradation. When *BUFFSIZE* uses the default value, an "out of memory" error occurs.

Finally, while modeling is beneficial for providing interpretability assurances for parameter tuning, for most ML engineers, completing the training process faster without crashes and reduced training accuracy is sufficient. The interpretability of given configurations is less important. Therefore, to tune all parameters, we opt for a non-model-based, black-box method.

C. Offline Tuning vs. Online Tuning

The distinction between offline and online tuning lies in whether tuning occurs before or during the training process. Although online tuning offers better dynamic adaptability, we consider it unreasonable to employ in the context of black-box tuning for all xCCL knobs. This is because a significant portion of the domain formed by all knobs is invalid, meaning that many knob combinations would lead to communication process crashes and consequently interrupt the training process. Such losses are unacceptable. Therefore, we opt for offline tuning before training, using a low-cost workload with characteristics similar to the training job.

D. Opportunities with Bayesian Optimization

Bayesian optimization (BO) based methods have emerged as promising approaches for knob tuning in complex systems. These methods leverage previous results to guide subsequent searches, offering high sample efficiency suitable for high-dimensional spaces. They can simultaneously optimize continuous and discrete parameters, while also demonstrating robustness against noisy data points. Consequently, these approaches show potential in automatically searching for performance optimization points within the xCCL parameter space.

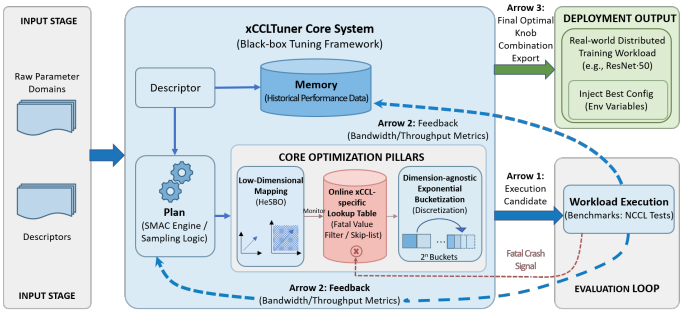

Fig. 2. Architecture of xCCLTuner.

BO leverages a surrogate model to model the function's distribution and utilizes an acquisition function to decide where to sample next. The conventional surrogate model for BO is the GP (Gaussian Process). SMAC (Sequential Model-Based Optimization for General Algorithm Configuration) [21] employs a RF (Random Forest), which better supports variables of continuous, discrete, and categorical types. The most recent update of SMAC further enhances support for features such as multi-objective, multi-fidelity, and multi-threading.

To the best of our knowledge, no prior work has addressed the automatic tuning of xCCL parameters. Although SMAC has demonstrated excellent performance in other parameter tuning tasks, such as hyperparameter tuning for reinforcement learning and database tuning, it is unfortunately insufficient for rapidly obtaining optimal configurations within xCCL context when applied directly for two reasons: (1) SMAC lacks optimizations tailored to the dimensional characteristics of xCCL. (2) SMAC's sampling process is not adapted to the specific properties of xCCL parameters. As our experiments in § VI-B3 demonstrate, the absence of these two adaptations can result in a search efficiency gap of several orders of magnitude. Consequently, we propose to enhance SMAC with appropriate designs to construct an automatic tuning system for xCCL.

IV. Overview

Figure 2 illustrates the architecture of xCCLTuner and its workflow for knobs tuning and finally boosting the training job. At a high level, the system takes the domain of xCCL parameters as input (①) and automatically completes the search of the parameter space, outputting a set of knob combinations that enhance performance (⑦). Specifically, the descriptor initializes a memory that stores historical performance based on the definition of parameters (②). The plan module then generates a set of parameters based on this memory (③). These parameters, after undergoing our design optimization, are configured (④). ⑤-⑥ represent the offline tuning process and the collection of performance data to update the memory. Note that the xCCLTuner system does not require interaction with training frameworks [15], [22]–[24]. The parameter combinations it outputs are applied to the training process through environment variable inheritance. Consequently, the models, training framework, and xCCL do not need to be modified.

Traditionally, engineers need to perform tuning based on their understanding of the current training environment and knobs, or spend time adapting the current model, topology, and training process to other works that requires code modifications. With xCCLTuner, manual optimization is no longer necessary. The entire process is simplified to waiting for xCCLTuner's output, configuring it in the system environment variables, and then running the training job.

V. Design

In this section, we introduce the key design features of xCCLTuner and describe how they address the challenges mentioned in §I. First of all we define the tuning problem. Suppose that we need to tune n knobs, $x_1, x_2, \cdots, x_n$, which may be continuous, discrete, or categorical, with each knob having its domain defined as $D_1, D_2, \cdots, D_n$. Then, the parameter space that we need to explore can be defined as $\mathcal{D} = D_1 \times D_2 \times \cdots D_n$. Without loss of generality, we assume that there exists a function $f : \mathcal{D} \to \mathbb{R}$ that captures the mapping from a set of parameters to an objective metric. Tuner aims to find a point X^* that satisfies $X^* = \text{argmax}_{X \in \mathcal{D}}\ f(X)$, i.e. maximizing or minimizing the metric.

A. Low Dimensional Tuning

If we use SMAC directly, the input space would be an n-dimensional space $\mathcal{D}$, with each dimension representing one knob, and a point X in the space indicating a configuration containing a specific value for each of the tuned knobs. The size of the parameter space is related to both the dimensionality (number of parameters) and the size of each dimension (the domain of each parameter). We first address the dimensionality aspect. Although SMAC can handle high-dimensional spaces, the number of iterations required for the search process grows with the number of parameters. Usually, NCCL has 30+ knobs available for tuning. With new hardware such as Infiniband or NVLink/NVSwitch, the number of knobs to consider increases. For instance, with support for IB/RoCE, an additional 16 parameters must be considered, bringing the total dimensionality of the parameter space to 50+. To obtain optimized knobs more rapidly and to preserve the potential for xCCLTuner to expand to multi-task scenarios, we decide to reduce the complexity introduced by dimensionality. As we have decided to tune all knobs in §III-B to avoid performance degradation, we approach this challenge by focusing on the representation of dimensions. We adopt a mapping method that aims to reduce the dimensionality of the SMAC's modeling space without decreasing the actual tuning space dimensionality, while ensuring performance gains.

1) Artificial Space Representation: Until now, we have assumed that each dimension of the space modeled by SMAC corresponds to a single knob. While intuitive, such mappings may not be the most conducive for SMAC to comprehend and extract information. Due to the interconnections and couplings between parameters, the original parameter space inherently contains redundancies. In statistical mathematics, techniques such as PCA (Principal Component Analysis) and LDA (Linear Discriminant Analysis) employ eigenvalue decomposition to identify the most significant dimensions and to obtain a low-dimensional space. In NLP, word embedding methods [25],

[26] represent textual data in an artificial space where semantically similar words are proximal to each other. This provides a dense representation of the vocabulary which, compared to the original sparse representation, can significantly enhance processing efficiency and model performance. Inspired by these approaches, we aim to decouple knobs and dimensions, enabling SMAC to model an artificial space $\mathcal{D}'$ instead of the original space $\mathcal{D}$, thereby achieving higher optimization efficiency. Specifically, a single knob in the artificial space can map to several knobs in the original space.

2) Low-dimensional BO: Recent work [27], [28] has explored the use of artificial spaces to assist BO. Studies [29], [30] have shown that when the dimensionality of the artificial space exceeds the **intrinsic dimensionality** of the original high-dimensional space, BO algorithms provide theoretical guarantees for finding the optimum point within a specifically defined artificial space. That is, as long as n', the size of the subspace $\mathcal{D}'$, is greater than the size of the intrinsic space $\mathcal{D}^I$, n^I, then there is a potential to identify a point X' in $\mathcal{D}'$ whose projection X^* in the original space $\mathcal{D}$ is the optimal or an approximate optimal point.

REMBO [29] pioneered the use of random projections. Given the dimension of the low-dimensional space D' as n', and the dimension of the original space as n, it uses the normal distribution $N(0,1)$ to generate a projection matrix $A \in \mathbb{R}^{n \times n'}$, which is then used to inversely project points X' sampled from the low-dimensional space back into the original space, i.e. $X = AX'$. This method is straightforward and effective, achieving excellent results in a 47-dimensional linear programming problem. However, it does not adequately address issues occurring when inversely projected points exceed the boundaries of the original space. When $AX' \notin D$, REMBO forcibly clips the values to the domain boundaries, thus limiting exploration to boundary points and hindering the BO surrogate's capacity to model the target space accurately. HeSBO [31] improves this inverse projection boundary problem, preserving characteristics from the original space (e.g. pairwise point distances). Given a target dimension n', HeSBO defines a subspace $D' = [-1,1]^{n'}$ and a projection matrix $\mathbf{A} \in \mathbb{R}^{n \times n'}$, ensuring that each parameter combination in $\mathcal{D}$ is influenced by only one corresponding position in $\mathcal{D}'$. Conversely, an artificial position in $\mathcal{D}'$ can modulate multiple parameters in the original space $\mathcal{D}$.

3) Integrating HeSBO into Tuning: The dimensionality reduction and modeling processes are decoupled, allowing for the integrating HeSBO without requiring modifications to SMAC. The integration of HeSBO into tuning is shown in Algorithm 1, where changes compared to the standard SMAC routine are highlighted. Inputs from the user include the dimension n' of the desired low-dimensional space and the tuner's maximum iteration threshold, *Thre*. Initially, an instance of a projection matrix, $\mathbf{A}$, is generated. Then, the planning module samples a point by maximizing the current acquisition function within SMAC. This sampled point is then inversely projected back to the original space using the matrix $\mathbf{A}$, and its corresponding parameter setting is used to run the workload and obtain performance measures. The new sample, comprising the projected point and its performance, are subsequently fed to the surrogate model for updating.

Algorithm 1 SMAC with low-dimensional projections. Blue underlined text highlights differences to original searching algorithm.

Input: $n', Thre$
Output: Approximate optimizer $p*$
1: Generate a projection matrix $\mathbf{A} \in \mathbb{R}^{n \times n'}$
2: **for** $i = 1, ..., Thre$ **do**
3: Sample a point $X'_i \in \mathcal{D}'$ from the planning module
4: Conduct workload on the projected point $f(\mathbf{A}\mathbf{X}'_i)$
5: Update memory module with $\{(\mathbf{X}'_i, f(\mathbf{A}\mathbf{X}'_i))\}$
6: **end for**
7: **return** $\mathbf{A}\mathbf{X}^*$ for the best point $\mathbf{X}^* \in \mathcal{D}'$

4) Denormalization: Without loss of generality, we instruct HeSBO to inversely project points sampled by SMAC back into the normalized space $[-1,1]^n$. Before these points are ultimately applied to configure NCCL, a simple scaling operation is required, namely denormalization. The parameters of NCCL consist of continuous, discrete, and categorical types. For continuous numerical variables, assuming scaling is required from $[x_{\min}, x_{\max}]$ to $[y_{\min}, y_{\max}]$, a simple min-max transformation $y = y_{\min} + \frac{x - x_{\min}}{x_{\max} - x_{\min}}(y_{\max} - y_{\min})$ is applied. For discrete numerical variables, the results after denormalization are rounded to the nearest integer. For categorical variables, which are inherently uniformly distributed discrete numerical variables, let there be k categories, we directly divide the $[-1,1]$ interval into k equal segments, with each interval assigning a category.

B. xCCL-specific Lookup Table

We now focus on further optimizing the tuning process by leveraging our prior knowledge of xCCL, distributed ML training, and workload characteristics. In this subsection, We begin by addressing categorical knobs and certain discrete knobs that function similarly (enumeration types). The values of these knobs usually toggle certain mechanisms or selects a particular paradigm, leading to significant performance differences.

To ensure these values which may cause dramatic performance changes are sampled with a relatively higher probability, previous researches [32]–[34] have utilized the bias sampling method. This approach involves defining special values for these knobs in advance, based on prior knowledge, so that the optimizer explores these values with a higher likelihood—thereby quickly developing cognizance of the performance impacts associated with these special values, which often result in considerable performance enhancements.

In the NCCL context, which knob values lead to significant performance gains vary depending on the scenario. We choose to let SMAC explore these values, without the need to predefine certain values and sample them with biased probabilities. Instead, we believe that the focus should be on addressing the phenomenon where specific values of certain NCCL knobs cause crashes. For example, when tuning *AllToAll* primitive using NCCL Tests (§VI-A) in our V100 topology (§VI-A), any iterations that configure *COMM_BLOCKING* or *P2P_LEVEL*

to 0 result in crashes. We refer to these risky parameter values as **fatal values**. In the tuning process, the presence of multiple fatal values may lead to a situation where modifying one does not resolve the issue, trapping SMAC in such configurations for extended periods. When the maximum iteration threshold *Thre* is set to 1000, up to 90% of iterations might be ineffective due to these fatal values in above case. In the example illustrating the selection of tuning methods in §III-B, values for *MIN_CHANNEL* greater than 2 are also fatal values. If these values are not excluded, all knob combinations containing them result in runtime crashes. In fact, fatal values are so prevalent that NCCL developers strongly advise users against configuring parameters that they are not well acquainted with.

It is important to note that fatal values are environment-dependent. The two aforementioned fatal values do not cause crashes with the *AllReduce* primitive and can finish the tests normally. Instead of a predefined biased sampling strategy, this situation as well as the potentially infinite number of fatal values raise the need for an online processing mechanism. The phenomenon of fatal values is also present in other xCCLs.

1) Online Lookup: To reduce the overhead of tuning being trapped, we design an online lookup table for promptly detecting, locating, and mitigating these issues, as outlined in Algorithm 2. Our central philosophy involves identifying fatal values when errors occur frequently, and instructing the optimizer to bypass these values in subsequent sampling iterations. We initially establish a sliding window of size w along with an error rate threshold θ. This module actively monitors the number of iterations failed within the most recent w attempts. Should the error rate exceed θ, it is deduced that the optimizer has encountered fatal values and is unable to proceed, thereby triggering the fatal value detection process. Given the most recent combination of knobs causing crash, represented as $\{knob_1 = value_1, knob_2 = value_2, \ldots, knob_n = value_n\}$, we check each key-value pair by constructing a knob combination that includes only the current $\{knob_i = value_i\}$ for execution. Knobs not specified will assume their default values. If the execution fails, $value_i$ is considered a fatal value for $knob_i$ and is stored in a global lookup table. In subsequent iterations, the system checks whether the knob combination includes $\{knob_i = value_i\}$. If it does, the iteration is skipped and the workload is not executed. The lookup table is constructed and queried at runtime.

Two more things are worth noting before finishing our discussion regarding online lookup. First, integrating the online pruning module requires no modifications to SMAC and HeSBO. SMAC combined with HeSBO can still generate knob combinations that may contain fatal values. When the lookup table matches a fatal value, it skips the execution of that iteration and returns a value that significantly deviates from the optimization objective. For example, if our goal is to maximize bandwidth, for failed and skipped iterations we can directly return a bandwidth that is exceedingly low. In other words, the correction occurs during the stage when SMAC learns from historical data, reducing the likelihood of sampling these values. Second, our assumption posits that a single fatal value is sufficient to induce runtime errors, and if execution using only that fatal value (with all other knobs set to defaults) would fail, we then presume that any knob combination containing that value would also result in crash. This naïve belief is rooted in our prior knowledge regarding the nature of fatal values. Although there may be crashes caused by multiple fatal values intertwining, these situations are preemptively matched by the lookup table and skipped. Our experiments have demonstrated that such online stragety is adequate to restore SMAC's ability to continue the normal tuning process and to efficiently locate knob combinations that significantly outperform the baseline.

Algorithm 2 Runtime Error Detection, Store and Mitigation.

Input: w, θ
Output: Table $\mathcal{T}$ of fatal values
Initialize $\mathcal{T}$ to $\emptyset$
Select the most recent knobs combination ES
for $k, v \in ES$ **do**
  Run workload with $\{k = v\}$
  if failed **then**
    Add $\{k = v\}$ to $\mathcal{T}$
  end if
end for
Continue to test new knobs combination S
for $k, v \in \mathcal{T}$ **do**
  if $\{k = v\} \in S$ **then**
    Skip and generate new S
  end if
end for
return $\mathcal{S}$

C. Dimension-agnostic Exponential Bucketization

Recall that in §V-A, we mentioned that the size of the parameter space is also determined by the domains of individual knobs. Now, we continue to leverage prior knowledge to handle the space contributed by vast domains. NCCL provides several numerical knobs, some of which have domain sizes in the billions. These knobs offer users the ability to fine-tune NCCL behavior with extreme granularity, but simultaneously greatly increase the burden of knobs tuning. During the tuning process, we observe that for these knobs, minor modifications do not markedly impact performance. For instance, increasing *BUFFSIZE* from 10,000 bytes to 10,100 bytes does not substantially alter performance. This phenomenon is not exclusive to NCCL; rather, it is commonplace among xCCLs.

Therefore, an intuitive approach is to further discretize the range of these parameters, specifically by further restricting the total number of possible values, denoted by N. Existing methods for discretization include equal-frequency binning, equal-width binning, discretization based on clustering, decision trees, etc. Among these methods, the simplest is equal-width discretization, which acts by uniformly dividing the range to $N - 1$ segments. At this point, we need to consider how to choose N, and whether the same N can be applied to parameters with different domains; these considerations require further validation. In addition, tuning with equal-width discretization resembles grid search, which can perform poorly in some situations as shown by past research [35].

Instead, we observe that when using heterogeneous topologies (such as mixing P100 with V100, or Titan Xp with 4090 (§ VI-B2), the selection of values for NCCL knobs becomes more stringent. Values that are not even or are not powers of two are very likely to cause NCCL crashes. Based on this observation, we decide to expand the interval between buckets exponentially, a method we refer to as **exponential bucketization**. Simultaneously, we retain the option for users to perform equal-width bucketization using hyperparameter N.

Similar to the online lookup mechanism, bucketization operates directly on the original knob space, oblivious to any prior dimensional reduction processes. SMAC with HeSBO remains unaware of the further discretized domain of the knobs, hence we refer to this approach as the Dimension-agnostic Exponential Bucketization. Specifically, the values sampled by SMAC with HeSBO are still from the original knob space, but before their final deployment, they are rounded to the nearest bucket by our bucketization mechanism. We illustrate through the following example that this implementation approach still helps SMAC more efficiently understand the properties of the discretized domain by learning from historical data. For *BUFFSIZE*, when SMAC with HeSBO samples several values within the original parameter space range of $[8192, 16384]$—such as 10,000 bytes and 10,100 bytes—these values are rounded to 8192. Since the performances associated with these values are almost the same, the optimizer incrementally increases the step size, understanding that performance gains on the *BUFFSIZE* dimension are unattainable without stepping out of the $[8192, 16384]$ range (in its lower-dimensional normalized interval).

Compared to directly modifying low-dimensional space representations by inserting discrete buckets into the low-dimensional space, our approach offers greater flexibility while ensuring boosting the tuning process. Switching from exponential bucketization to equal-width bucketization or other discretization methods merely requires changing the discretization logic at the end of the workflow, without the need to redefine a low-dimensional bucketized space for each knob.

VI. Evaluation

A. Experiment Setup

Topology setup. We mainly conduct experiments on 2 topologies. Topology A comprises 4 nodes interconnected via a 10Gbps Ethernet switch. Each node is equipped with 4 Tesla V100 GPUs (16GB DDR, PCIe), 8 RAM sticks (32GB DDR) and 2 Intel Xeon E5-2609 v4 processors, each featuring 8 physical cores with a base frequency of 1.70 GHz. Topology B comprises 3 nodes interconnected via a 1Gbps Ethernet switch. Each node is equipped with 4 Tesla P100 GPUs (16GB DDR, PCIe), with the remaining setup identical to Topology A.

Exploratory and supplementary experiments are conducted on 2 additional topologies. Topology C includes 2 nodes, each with 3 Titan Xp GPUs, while Topology D consists of 2 machines, each with 8 Tesla V100-SXM2 GPUs (32GB DDR).
Tuning setting. Most experiments are conducted using NCCL v2.18.3. On topologies C and D, we also test NCCL v2.15.1 and v2.20.5. From the tunable knobs available in these versions of NCCL, we select a set of 33 knobs common to all versions, excluding those unsupported by the current hardware (e.g. IB-related knobs), as well as parameters related to debugging or checkpointing. The tuning objective is set to high bandwidth. Each tuning scenario is run for 1000 iterations. For both the default knob combinations and the best knob combinations found through tuning, we repeat 100 times and take the average. For knob combinations that caused the workload to crash, we employ the approach mentioned in §V-B1 and return a exceedingly low value.
Workloads. NCCL Tests is an open-source test suite designed for checking the performance and correctness of NCCL primitives. It supports testing collective communication primitives across multiple nodes, and permits customization of various hyperparameters including the numbers of timed and warmup iterations, workload sizes, process counts, quantity of GPUs engaged, etc. We employ the *AllReduce* and *AlltoAll* tests from NCCL Tests as benchmarks for offline parameter tuning, and bus bandwidth as the performance metric for optimization.

We exclude test tasks whose sizes are too small for accurate measurement of elapsed time, testing workloads beginning at 128KB and increasing by powers of two up to 64MB. We collect measurements for 20 iterations after 10 warm-ups (the default setting). NCCL's default performances are the baseline.

We conduct performance tests using ResNet-50 [36] for real-world training scenarios after parameter tuning. Synthetic data are employed to test the model's inter-machine bandwidth during distributed training at four different batch sizes: 4, 8, 32, and 64. We record bandwidth information during the stable period of bandwidth usage midway through the training.
Optimizer. We employ SMAC as base optimizer of xCCLTuner, with the ratio for random search set to 0.1 and the number of estimators set to 100.

B. Microbenchmark: NCCL Tests

1) Results of Bandwidth Optimization: Figure 3 shows the results obtained by adjusting the number of nodes on Topologies A and B. The performance values obtained from repeated tests are presented as box plots. From top to bottom, the three horizontal lines within each box represent the third quartile, median, and first quartile, respectively. Green stands for means. The whiskers represent the upper and lower extremes, and outliers beyond the whiskers are indicated by circles. The results demonstrate that xCCLTuner can consistently find knob combinations that yield better average performance than the defaults. On Topology B, xCCLTuner observes up to $1.24\times$ and $1.8\times$ improvements on *AllReduce* and *AllToAll* primitives, respectively. On Topology A, xCCLTuner achieves up to $2.56\times$ and $5.87\times$ enhancements on *AllReduce* and *AllToAll*, respectively. Furthermore, it is important to emphasize that as the number of nodes increases, xCCLTuner continues to identify opportunities for performance enhancement. This indicates that the optimization effects of knob tuning is not confined to trivial cases involving one or two nodes, and that the tuning capability of xCCLTuner is scalable.

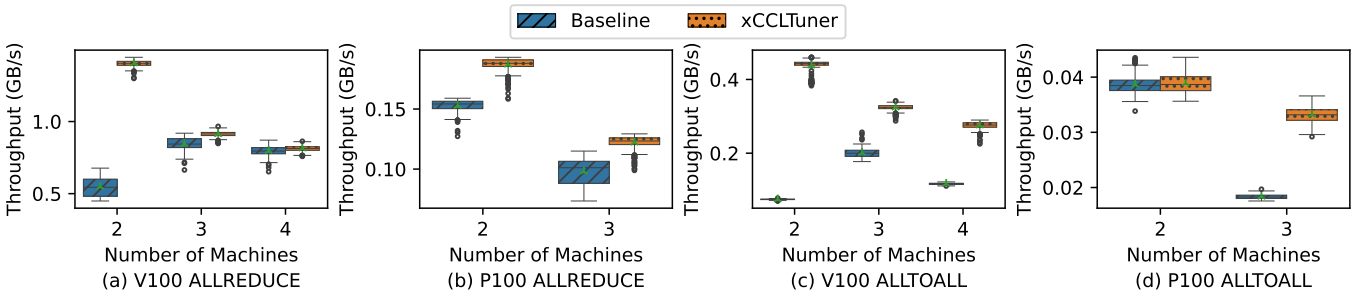

Fig. 3. Performance of xCCLTuner on NCCL Tests when using topologies A and B.

TABLE III
PERFORMANCE GAINS OF XCCLTUNER UNDER THE HETEROGENEOUS TOPOLOGY AND CONSUMER-GRADE GPUS.

Settings		Throughput Improvement	
Topologies	Primitives	[5%, 95%] CI	Average
1*4*V100 + 2*4*P100	*AllReduce*	[1.51×, 1.56×]	1.53×
1*4*V100 + 2*4*P100	*AllToAll*	[1.23×, 1.31×]	1.28×
2*2*Titan Xp	*AllReduce*	[1.42×, 1.44×]	1.43×
2*2*Titan Xp	*AllToAll*	[3.21×, 3.29×]	3.25×

2) Bandwidth Optimization Results on Heterogeneous Topology and Consumer-Grade GPUs: Many engineers conduct ML training using consumer-grade GPUs or lower-end GPUs intended for educational purposes, often in conjunction with slower network topologies. In these scenarios, engineers typically have a greater need for user-friendly, out-of-the-box optimization methods that can accelerate training without requiring extensive modification.

We construct a heterogeneous topology by assembling one node from Topology A with two nodes from Topology B, interconnected via a 1Gbps Ethernet switch. As illustrated in Table III, xCCLTuner achieved optimization results of $1.53\times$ for *AllReduce* and $1.28\times$ for *AllToAll* in terms of bandwidth. Additionally, tests are conducted on Topology C, where nodes are equipped with consumer-grade GPU, Titan Xp. xCCLTuner achieves impressive outcomes, securing bandwidth improvements of $1.43\times$ for *AllReduce* and $3.25\times$ for *AllToAll*, respectively. These supplementary experiments further demonstrate the broad applicability and effectiveness of xCCLTuner across diverse scenarios.

3) Ablation Experiments: We present the results of ablation experiments that contrast vanilla SMAC, SMAC enhanced with online lookup and bucketization, and xCCLTuner.

Figure 4 displays the highest performance achieved by each optimizer at each iteration across tested scenarios. Tests are conducted on Topology A (§VI-A), targeting the *AllReduce* and *AllToAll* primitives, for 2 and 3 machines. Figure 4 demonstrates that, across all tested scenarios, the performance of vanilla SMAC is consistently the poorest. This is especially evident in the 3-node *AllReduce* scenario, as depicted in Figure 4(b), where SMAC fell into the fatal values trap, failing to uncover better parameters even after 1000 iterations. With online lookup and bucketization, the optimizer's capability is enhanced, allowing it to avoid being trapped in suboptimal local points. In certain scenarios, as illustrated in Figure 4(a) and Figure 4(b), it is able to find optimal or near-optimal configurations, but its efficiency still fell short of xCCLTuner's, which benefited from a low-dimensional tuning mechanism.

C. Macrobenchmark: Real Distributed ML Workload

xCCLTuner shows scalability and accelerates training for real-world DNNs. As depicted in Figure 5, under default configuration, ResNet-50 is unable to fully utilize the 10Gbps inter-node bandwidth of Topology A across 4 different batch sizes. However, after applying the optimal knobs obtained from NCCL Tests for *AllReduce* on the corresponding number of machines using xCCLTuner, we achieve a maximum acceleration of up to $1.52\times$. Additionally, as the number of nodes increas, with the number of GPUs involved in the training reaching up to 16, we still observe performance improvements. These experiments demonstrate that the optimal knob combinations xCCLTuner derived from the NCCL Tests benchmarks remain effective in real-world training scenarios, and exhibit scalability with the expansion of the topology size.

D. Scalability

TABLE IV
NOTATION OF SCALABILITY ANALYSIS.

Notation	Description
E	Total Knob tuning time
I	Number of tuning iterations
e_i	Time per tuning iteration
A	Time per SMAC decision
R	Single test run time
B	Average bandwidth
n	Number of GPUs

Next, we present a theoretical analysis of the scaling behavior of xCCLTuner's knob tuning overhead as the topology expands, based on our experimental results. The notation required is presented in Table IV. First, we have $E = I \times e_i$. As the size of the parameter space does not change with the topology size, the modeling difficulty for SMAC does not increase, thus the number of tuning iterations I remains constant. For the time per iteration e_i, we have $e_i = A + R$. Under the hyper-parameters of SMAC and node specifications mentioned in §VI-A, A is negligible (less than 0.1s). We then analyze the runtime R. This is related to both primitives and their implementation, and cannot be generalized. Intuitively, R is inversely proportional to the B. Based on this rule, we present an analysis of R for two experiments on topology A.

(1) When using primitives such as *AllReduce*, *AllGather*, and *ReduceScatter*, assuming the ring algorithm is employed, the inter-node bandwidth becomes the bottleneck. After increasing the number of nodes, disregarding the negligible time for establishing connections between nodes, the runtime R

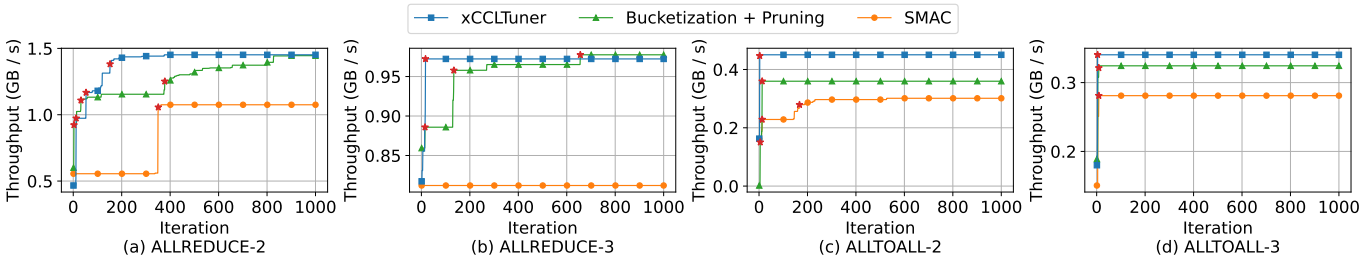

Fig. 4. Ablation experiments of xCCLTuner.

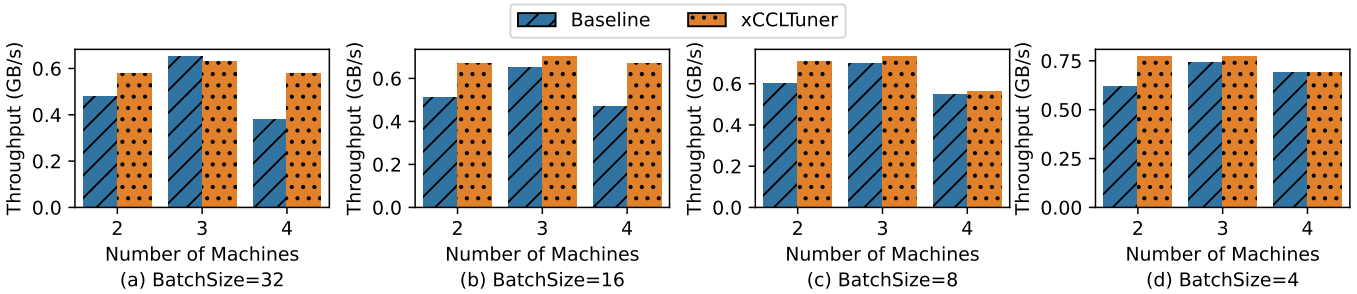

Fig. 5. Performance of xCCLTuner on ResNet-50 with varying batch size.

converges to the inverse of the bottleneck network bandwidth B (multiplied by a constant data size). In our scenario, the bottleneck is the 10G network, and the average baseline runtime for 2, 3, and 4 nodes is 27 ± 1s. This suggests that B should not differ significantly among 2, 3, and 4 nodes, which aligns with the results shown in Figure 3(a).

(2) When using *AllToAll*, R increases as the amount of data transmitted grows with the addition of nodes. In topology A, there is no NVLink within nodes, and the 4 V100 GPUs theoretically share 1/4 of the bandwidth. Therefore, the maximum bandwidth should be around 0.2-0.4 GB/s, which is consistent with the performance shown in Figure 3(c). It is important to note that, due to NCCL's default knobs not being optimal, the average bandwidth B of the baseline first increases and then decreases as the number of nodes increases. Correspondingly, R should first decrease and then increase, which aligns with our test results of baseline average runtimes of 119s, 30s, and 60s. In summary, R depends on the primitive, which may be constant or linearly increasing with the number of GPUs n, and is further influenced by the specific knobs chosen during the tuning process. As I remains constant, the overall complexity of the total tuning time E is $\mathcal{O}(n)$.

VII. Related Works

Communication Improvement. Recognizing communication as the scalability bottleneck in distributed machine learning, studies [37]–[46] aim to optimize the communication process during training. Works [37], [41]–[44] attempt to overlap communication with computation through measurement and scheduling techniques. Other works [38]–[40], [45], [46] seek to reduce communication costs through methods like gradient and state information compression. Again, while these approaches are currently orthogonal to xCCLTuner and lack the ability to fine-tune for specific training environments, xCCLTuner can be utilized to perform tuning on them once they are incorporated into xCCL as configurable mechanisms.

xCCL Knobs related Work. To the best of our knowledge, no rigorous work has been conducted on the automated tuning of xCCL knobs, making xCCLTuner the first. Previous studies [47]–[50] have only sporadically mentioned xCCL knobs. Study [47] examines the impact of two parameters, *NSOCKS_NTHREADS* and *SOCKET_NTHREADS*, on training performance in single-machine, multi-GPU cloud instances. [48] presents an empirical observation: better overlap between gradient update and gradient reduction can be achieved by adjusting two knobs in Horovod [51], *CYCLE_TIME* and *FUSION_BUFFSIZE*. Work [49] employs profiler to identify *CYCLE_TIME* and *FUSION_THRESHOLD* as the bottleneck knobs in Horovod, then utilizes grid-search to find the optimal configuration for these two parameters, which lacks scalability. In this issue [50], engineers call for the provision of tunable knobs API similar to xCCL, allowing for self-tuning to address the problem of multi-GPU training being slower than single-GPU training in the current training environment.

VIII. Conclusion

We introduce xCCLTuner, an automated tuning system that optimizes xCCL parameters for distributed training without requiring manual parameter categorization. Leveraging BO, xCCLTuner effectively manages high-dimensional parameter spaces through low-dimensional tuning, specialized lookup tables, and bucketization techniques. xCCLTuner does not require modifying any existing protocol stacks or system kernels. Instead, it serves as an out-of-the-box tuner for engineers who are not familiar with xCCL. Our experiments demonstrate that xCCLTuner enhances throughput by up to 5.87× across various scenarios, showcasing its efficiency and ease of use in reducing training costs while maintaining accuracy.

References

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

[2] T. T. Nguyen and M. Wahib, "An allreduce algorithm and network co-design for large-scale training of distributed deep learning," in *CCGrid*. IEEE, 2021, pp. 396–405.

[3] J. Fei, C.-Y. Ho, A. N. Sahu, M. Canini, and A. Sapio, "Efficient sparse collective communication and its application to accelerate distributed deep learning," in *SIGCOMM*, 2021, pp. 676–691.

[4] P. Sanders, J. Speck, and J. L. Träff, "Two-tree algorithms for full bandwidth broadcast, reduction and scan," *Parallel Comput.*, vol. 35, no. 12, pp. 581–594, 2009.

[5] X. Jia *et al.*, "Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. corr abs/1807.11205 (2018)," *arXiv preprint arXiv:1807.11205*, 2018.

[6] H. Mikami *et al.*, "Imagenet/resnet-50 training in 224 seconds," *arXiv preprint arXiv:1811.05233*, pp. 770–778, 2018.

[7] C. Ying, S. Kumar, D. Chen, T. Wang, and Y. Cheng, "Image classification at supercomputer scale," *arXiv preprint arXiv:1811.06992*, 2018.

[8] H. Kim, J. Ryu, and J. Lee, "Tccl: Discovering better communication paths for pcie gpu clusters," in *ASPLOS*, 2024, pp. 999–1015.

[9] A. Shah *et al.*, "{TACCL}: Guiding collective algorithm synthesis using communication sketches," in *NSDI*, 2023, pp. 593–612.

[10] Z. Cai *et al.*, "Synthesizing optimal collective algorithms," in *PPoPP*, 2021, pp. 62–75.

[11] D. De Sensi, T. Bonato, D. Saam, and T. Hoefler, "Swing: Short-cutting rings for higher bandwidth allreduce," in *NSDI*, 2024, pp. 1445–1462.

[12] Z. Jiang *et al.*, "{MegaScale}: Scaling large language model training to more than 10,000 {GPUs}," in *NSDI*, 2024, pp. 745–760.

[13] K. Qian *et al.*, "Alibaba hpn: a data center network for large language model training," in *SIGCOMM*, 2024, pp. 691–706.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, vol. 25, 2012.

[15] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[16] M. Naumov *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *CoRR*, vol. abs/1906.00091, 2019. [Online]. Available: https://arxiv.org/abs/1906.00091

[17] Advanced Micro Devices, Inc., "Rocm communication collectives library," 2024, https://github.com/ROCm/rccl.

[18] Intel, "oneccl," 2024, https://github.com/ROCm/rccl.

[19] Microsoft, "Microsoft collective communication library (msccl)," 2021, https://github.com/microsoft/msccl.

[20] J. Dong *et al.*, "Accl: Architecting highly scalable distributed training systems with highly efficient collective communication library," *IEEE Micro*, vol. 41, no. 5, pp. 85–92, 2021.

[21] M. Lindauer *et al.*, "Smac3: A versatile bayesian optimization package for hyperparameter optimization," *J. Mach. Learn. Res.*, vol. 23, no. 54, pp. 1–9, 2022.

[22] P. Moritz *et al.*, "Ray: a distributed framework for emerging ai applications," in *OSDI*, 2018, p. 561–577.

[23] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *KDD*, 2020, pp. 3505–3506.

[24] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *NIPS*, 2019.

[25] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[26] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *EMNLP*, 2014, pp. 1532–1543.

[27] M. Binois and N. Wycoff, "A survey on high-dimensional gaussian process modeling with application to bayesian optimization," *ACM Trans. Evol. Learn. Optim.*, vol. 2, no. 2, pp. 1–26, 2022.

[28] B. Bischl *et al.*, "Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges," *WIREs Data Mining Knowl. Discovery*, vol. 13, no. 2, p. e1484, 2023.

[29] Z. Wang, F. Hutter, M. Zoghi, D. Matheson, and N. De Feitas, "Bayesian optimization in a billion dimensions via random embeddings," *J. Artif. Intell. Res.*, vol. 55, pp. 361–387, 2016.

[30] D. Eriksson and M. Jankowiak, "High-dimensional bayesian optimization with sparse axis-aligned subspaces," in *UAI*. PMLR, 2021, pp. 493–503.

[31] A. Nayebi, A. Munteanu, and M. Poloczek, "A framework for bayesian optimization in embedded subspaces," in *ICML*. PMLR, 2019, pp. 4752–4761.

[32] X. Zhang *et al.*, "An efficient transfer learning based configuration adviser for database tuning," *Proc. VLDB Endow.*, vol. 17, no. 3, pp. 539–552, 2023.

[33] D. D. Lewis and J. Catlett, "Heterogeneous uncertainty sampling for supervised learning," in *Mach. Learn. Proc.* Elsevier, 1994, pp. 148–156.

[34] J. Altmann, "Observational study of behavior: sampling methods," *Behaviour*, vol. 49, no. 3-4, pp. 227–266, 1974.

[35] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *J. Mach. Learn. Res.*, vol. 13, no. 2, 2012.

[36] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016, pp. 770–778.

[37] S. H. Hashemi, S. Abdu Jyothi, and R. Campbell, "Tictac: Accelerating distributed deep learning with communication scheduling," in *MLSys*, vol. 1, 2019, pp. 418–430.

[38] Y. Bai *et al.*, "Gradient compression supercharged high-performance data parallel dnn training," in *SOSP*, 2021, pp. 359–375.

[39] Y. Ma *et al.*, "Eliminating data processing bottlenecks in gnn training over large graphs via two-level feature compression," *Proc. VLDB Endow.*, vol. 17, no. 11, pp. 2854–2866, 2024.

[40] H. Wu *et al.*, "A generic, high-performance, compression-aware framework for data parallel dnn training," *IEEE Trans. Parallel Distrib. Syst.*, 2023.

[41] A. Jangda *et al.*, "Breaking the computation and communication abstraction barrier in distributed machine learning workloads," in *ASPLOS*, 2022, pp. 402–416.

[42] L. Chang *et al.*, "Flux: Fast software-based communication overlap on gpus through kernel fusion," *arXiv preprint arXiv:2406.06858*, 2024.

[43] S. Wang *et al.*, "Overlap communication with dependent computation via decomposition in large deep learning models," in *ASPLOS*, 2022, pp. 93–106.

[44] K. Mahajan, C.-H. Chu, S. Sridharan, and A. Akella, "Better together: Jointly optimizing {ML} collective scheduling and execution planning using {SYNDICATE}," in *NSDI*, 2023, pp. 809–824.

[45] X. Zhu, J. Li, Y. Liu, C. Ma, and W. Wang, "A survey on model compression for large language models," *arXiv preprint arXiv:2308.07633*, 2023.

[46] J. Huang *et al.*, "gzccl: Compression-accelerated collective communication framework for gpu clusters," in *ICS*, 2024, pp. 437–448.

[47] E. Begoli, S.-H. Lim, and S. Srinivasan, "Performance profile of transformer fine-tuning in multi-gpu cloud environments," in *BigData*. IEEE, 2021, pp. 3095–3100.

[48] J. Yin *et al.*, "Strategies to deploy and scale deep learning on the summit supercomputer," in *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, 2019, pp. 84–94.

[49] Q. Anthony, A. A. Awan, A. Jain, H. Subramoni, and D. K. D. Panda, "Efficient training of semantic image segmentation on summit using horovod and mvapich2-gdr," in *IPDPSW*. IEEE, 2020, pp. 1015–1023.

[50] "torchrec: Super slow allreduce in multi-node multi-gpu setting." [Online]. Available: https://github.com/facebookresearch/dlrm/issues/356

[51] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.