TuCCL: Tailored and Unified Configuration Optimizations for High-Performance Collective Communication Library

Ziming Li[†], Chenyang Hei[†✉], Fuliang Li[†✉], Tongrui Liu[†], Chengxi Gao[£], Xiuzhu Sha[†], Xingwei Wang[†]
[†]Northeastern University, China
[£]Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences
Email: 20226541@stu.neu.edu.cn, 2310726@stu.neu.edu.cn, lifuliang@cse.neu.edu.cn, 2301916@stu.neu.edu.cn, chengxi.gao@siat.ac.cn, 20225843@stu.neu.edu.cn, wangxw@mail.neu.edu.cn

***Abstract*—Modern distributed training systems face escalating communication bottlenecks as GPU clusters scale to accommodate large models. While collective communication libraries and automated synthesizers address algorithmic efficiency, they suffer from three critical limitations including labor-intensive manual intervention requirement, overreliance on predefined input optimization, and suboptimal isolated configuration optimization. To solve these problems, we present TuCCL, a systematic framework that co-optimizes communication algorithms and runtime parameters through three innovations: Topology-Aware Sketch Generation that automatically produces high-performance primitives, Hierarchical Configuration Optimization modeling nonlinear parameter-performance relationships, and Multi-phase Resource-Aware Configuration Optimization enabling joint configuration tuning with adaptive search space pruning. Evaluations demonstrate TuCCL's superiority over state-of-the-art systems with 1.75x–11.49x bandwidth improvements for ALLGATHER/ALLREDUCE on NVIDIA V100/A100 clusters, 90.2% faster configuration search than grid methods, and 1.22x-2.52x end-to-end training speedups across diverse model scales.**

***Index Terms*—Collective communication, distributed training, automated synthesizer, configuration tuning**

I. INTRODUCTION

Modern large-scale deep learning systems increasingly rely on distributed parallel training [1] across GPU clusters to address the growing computational demands of foundation models. Central to this process is collective communication [2], [3], a critical mechanism that facilitates coordinated data exchange between distributed workers, including gradients, parameters, and optimizer states. As GPU cluster sizes expand exponentially - with recent deployments exceeding tens of thousands of devices - communication efficiency has emerged as a primary performance bottleneck. For instance, DeepLight [4] reports that when handling parameter sizes reaching 2GB, synchronization-related communication overhead can consume up to 79% of total training time in large-scale systems. This significant time allocation underscores the urgent need for optimized communication algorithms.

Existing efforts to address this challenge fall into three categories. First, Open-source collective communication libraries (CCLs), such as NCCL [5] and RCCL [6], provide high-performance implementations of standardized algorithms (e.g., ring-based ALLREDUCE, double binary trees [7]), yet these static solutions lack adaptability to heterogeneous[1] or hierarchical network topologies prevalent in modern data centers. Second, domain experts manually design topology-specific communication strategies [8]–[10], but such approaches demand prohibitive human effort and fail to generalize. Third, to address various collectives and topologies, recent automated synthesizers (e.g., TACCL [3], TECCL [11], and SCCL [12]) generate near-optimal communication schedules by combining topology-aware routing with resource allocation. While these tools show promise, they face three critical limitations:

Limitation 1: Labor-intensive manual intervention requirement. Existing efforts prioritize algorithmic optimization while neglecting end-to-end automation. Current implementations require extensive manual intervention, demanding that operators: (1) possess detailed hardware expertise to complete topology sketches, (2) conduct iterative testing to identify viable low-level configurations, and (3) navigate interdependent trade-offs between communication sketches and their search spaces. This process is inherently labor-intensive, as optimized sketches may inadvertently restrict search spaces, while distinct sketches yield algorithm sets with divergent performance characteristics (refer to Insight 1 in Section II-C for more details).

Limitation 2: Overreliance on predefined input optimization. These systems fundamentally assume that input communication sketches (abstract specifications of data movement patterns) and low-level configurations (e.g., chunk size, buffer allocation) are already optimized. However, even minor variations in sketch design or configuration parameters significantly impact synthesized algorithm performance, leaving substantial optimization potential untapped. Current synthesizers operate as black boxes, offering neither guidance nor transparency for refining these critical inputs (refer to Insight 2 in Section II-C for more details).

Limitation 3: Suboptimal isolated configuration optimization. While AFANA [13] advances configuration optimization for collective communication libraries, its scope remains constrained to backend primitives of communication libraries. However, the synthesizer configuration - a pivotal

✉Co-corresponding authors.

979-8-3315-0376-5/25/$31.00 ©2025 IEEE

[1]'Heterogeneous' denotes network heterogeneity (bandwidth and topology), not GPU performance.

factor in algorithm performance, particularly for network link utilization - receives no systematic optimization. Current single-dimensional optimization approaches yield only marginal performance improvements due to the inherent interdependencies between system components (refer to Insight 3 in Section II-C for more details).

To solve these problems, we present *TuCCL*, a systematic tuning framework that bridges this gap by co-optimizing communication sketches and runtime configurations for automated synthesizers. The core challenge lies in identifying performance-sensitive sketch patterns and configurations across diverse hardware topologies while avoiding combinatorial explosion in the search space. Our solution introduces an innovative framework with three interconnected components: **(1) Topology-Aware Sketch Generation:** Automatically generates high-performance communication primitives by empirically characterizing interactions between communication sketches and synthesizers. **(2) Hierarchical Configuration Optimization:** Models nonlinear relationships between low-level parameters (e.g., transmission parallelism, buffer allocation) and bandwidth utilization to guide systematic tuning. **(3) Multi-phase Resource-Aware Configuration Optimization (MRACO):** Coordinates multidimensional configuration adjustments while strategically narrowing the search space through adaptive resource constraints.

To summarize, our main contributions are listed as follows:

- We first apply experiments and analysis to discuss about the limitations in existing configuration optimization works, revealing the synergistic effects of synthesizer configurations and backend primitives on network utilization.
- Then, we propose *TuCCL*, a tailored and unified configuration optimization scheme for efficient collective communications, with three key components, in order to generate topology-aware sketch to mitigate manual intervention, optimize hierarchical configuration to guide systematic tuning, and propose multi-phase resource-aware optimization schemes to adjust multidimensional configuration.
- Finally, we conduct extensive experiments to evaluate the performance of *TuCCL* and comparison results show that *TuCCL* achieves 1.75x–11.49x higher bandwidth than MSCCL [14] on NVIDIA V100/A100 clusters for critical operators (ALLGATHER, ALLREDUCE), reduces configuration search time by 90.2% compared to grid search, and delivers 1.22x–2.52x end-to-end training speedups across diverse model scales.

II. BACKGROUND AND MOTIVATION

A. Collective Communication

Collective communication refers to coordinated data exchange patterns among multiple processing units to achieve system-wide synchronization or data aggregation. In modern distributed deep learning, GPUs within a computing cluster must frequently transport intermediate data (e.g., gradients, model parameters) across nodes. This involves two key aspects: 1) *inter-GPU data movement* via high-speed interconnects (e.g., NVLink, InfiniBand) and 2) *collaborative communication patterns* among workers in a compute group, where tasks such as gradient synchronization require strict ordering and consistency guarantees. State-of-the-art collective communication libraries (CCL), such as NVIDIA's NCCL, abstract hardware complexities by providing optimized implementations of collective operators. Common operators include AllReduce, AllGather, Broadcast and so on.

Modern communication libraries decouple design into two layers: (1) Algorithm Layer: Defines fine-grained data transmission schedules to implement operators. It determines how to partition data, route transfers, and overlap computation/communication. For instance, a ring-based ALLREDUCE algorithm minimizes bandwidth contention by organizing GPUs into a logical ring for sequential data passing. (2) Backend Layer: Executes algorithms on physical hardware, handling low-level operations like GPU Direct RDMA or network protocol management.

Traditional approaches to algorithm design face scalability-efficiency trade-offs: *(1) Vendor-optimized algorithms* (e.g., NCCL's default schemes) leverage hardware-specific tuning but struggle to adapt to nonstandard or heterogeneous network topologies. *(2) Expert-crafted algorithms* for bespoke topologies achieve near-optimal bandwidth but require labor-intensive manual design, making them impractical for dynamic environments. To address this, recent work introduces automatic algorithm synthesizers [3], [11], [15]. These frameworks systematically explore the algorithmic space using topology-aware cost models and constraint solvers, generating near-optimal schedules with minimal human effort. For example, synthesizers might evaluate thousands of potential data segmentation strategies and communication paths to maximize bandwidth utilization while adhering to hardware constraints.

B. CCL Configurations

Synthesizer configurations. Modern collective communication synthesizers automate the generation of topology-aware algorithms by formalizing hardware constraints and communication logic into *communication sketches* [3]. These sketches serve as structured inputs to guide the synthesis process, capturing two critical dimensions: (1) Hardware Configuration: Physical characteristics of the network, such as intra-server link latency (α) and bandwidth ($1/\beta$), define the cost model for evaluating algorithm efficiency. These parameters directly influence the synthesizer's objective function, which aims to minimize end-to-end communication overhead. (2) Logical Routing Constraints: The $\langle$*GPU Peers*$\rangle$ specification enumerates inter-GPU connections across servers, dictating permissible data paths. For instance, denser peer configurations may increase network card utilization but risk congestion, while sparse configurations prioritize link stability. Additionally, (Rank_i, NIC_i) bindings enforce GPU-to-network-interface affinity, ensuring prioritized data transfers through designated hardware paths. Collectively, these parameters form

TABLE I
SYNTHESIZER AND BACKEND CONFIGURATION PARAMETERS

Category	Parameter	Description	Example Value/Range
Synthesizer	α / Remote α	Intra/Inter-server link latency	3 μs / 26 μs
	β / Remote β	Unit data transfer time on the link	7 μs / 32 μs
	$\langle$GPU Peers$\rangle$	Intra/Inter-server GPU connections	{0-1, 0-3, 2-3, 2-5}
	(Rank_i, NIC_i)	GPU-NIC binding policy	$\text{Rank}_0 \rightarrow \text{NIC}_0$
Backend	Protocol (p)[†]	Memory transfer granularity	[Simple, LL, LL128]
	NChannels (N_c)	Parallel data streams	4
	NThreads (N_t)	Threads per Thread Block	128
	Buffer Size (B)	Per GPU Peer buffer allocation	8 MB

[†]Simple: Bulk transfers; LL: 8/16-byte atomics; LL128: 128-byte atomics.

the *synthesizer configuration* (Table I), enabling topology-aware algorithm exploration.

Backend configurations. While synthesizers generate high-level algorithms, their practical performance is determined by low-level execution parameters, termed *backend configurations* (Table I). These parameters systematically regulate the mapping of algorithms to GPU hardware resources [2]: Protocol(p): Dictates memory transfer granularity and thread block (TB) scheduling. For example: (1) Simple Protocol maximizes bandwidth via large, contiguous memory copies. (2) Low-Latency (LL) protocol minimizes synchronization overhead for small transfers. (3) LL128 balances latency and bandwidth using 128-byte atomic operations.

Parallelism Controls. (1) NChannels (N_c) partitions data into parallel streams, each managed by a dedicated TB. This exploits multi-path network capacity but requires careful channel-to-connection mapping to avoid contention. (2) NThreads (N_t) configures thread count per TB, optimizing compute occupancy for auxiliary operations (e.g., data reduction within buffers). (3) BufferSize (B) allocates memory for intermediate operations (e.g., partial gradient aggregation). Larger buffers amortize kernel launch overhead but increase memory pressure.

Synthesizer-Backend Co-Design. The interplay between synthesizer and backend configurations ensures both algorithmic optimality and hardware efficiency. For instance, a synthesizer might generate a multi-path ALLREDUCE algorithm based on $\alpha - \beta$ parameters [7], while the backend configures $N_c = 4$ to parallelize transfers across four NVLink channels. Such co-design avoids suboptimal outcomes where synthesized algorithms ignore hardware execution nuances (e.g., thread block scheduling limits) or backend settings misalign with topology constraints.

C. Motivation

Insight 1: Default sketch configuration hinders algorithm efficiency, which requires labor-intensive manual intervention. Modern communication algorithm synthesizers (e.g., TACCL) rely heavily on manual parameter tuning to optimize input sketches, and this process is not only time-consuming but also demands high levels of expertise, as the effectiveness of the configurations critically affects synthesis performance and search space complexity. To quantify this dependency, we evaluate the end-to-end link utilization of collective communication algorithms synthesized using default all-to-all connectivity sketches. As shown in Table II, under a V100 cluster environment (detailed in §IV-A), these unoptimized default sketches yield alarmingly low bandwidth utilization—as low as 0.47% in extreme cases. This inefficiency arises because the synthesizer is unable to autonomously adapt communication patterns to hardware-specific constraints, such as topology-dependent connectivity and link load balancing. The manual intervention required to optimize these configurations involves dynamic analysis of network topologies to ensure efficient link usage and congestion avoidance. These tasks demand significant expert experience to properly assess factors such as the topology's unique routing characteristics, communication path overlap, and how to balance the workload across various links to minimize bottlenecks. As a result, fine-tuning these configurations becomes a labor-intensive process that is not only time-consuming but also prone to human error, thus hindering the overall scalability and efficiency of the synthesis process. Therefore, developing an automatic synthesis method for topology-aware sketch configurations is crucial.

TABLE II
LINK UTILIZATION EFFICIENCY OF SYNTHESIZED ALGORITHMS UNDER DEFAULT COMMUNICATION SKETCHES

Clusters	AllGather	AllReduce
V100	5.77%	0.47%
A100	24.78%	2.79%

Insight 2: Predefined inputs are suboptimal for critical operators. Synthesizers often assume that default configurations provided by communication libraries are pre-optimized. Our controlled experiments reveal this assumption to be flawed. We evaluate ALLGATHER and ALLREDUCE operators under isolated network conditions using a 4 GB payload on a testbed comprising two nodes, each equipped with eight

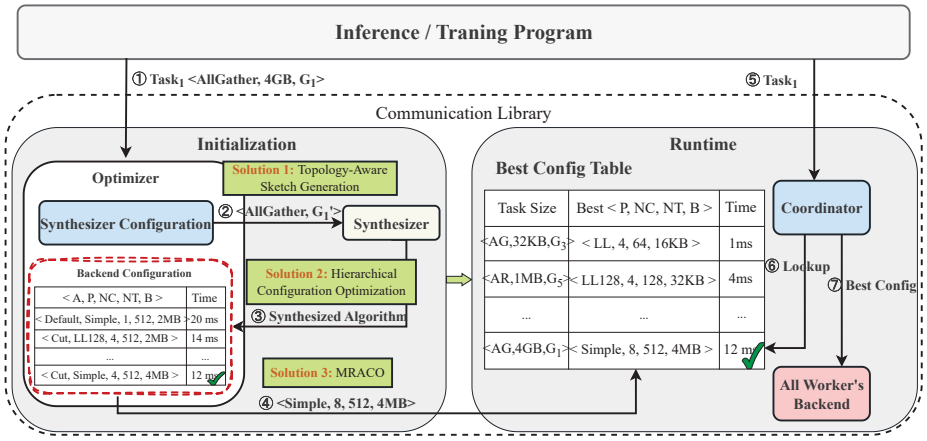
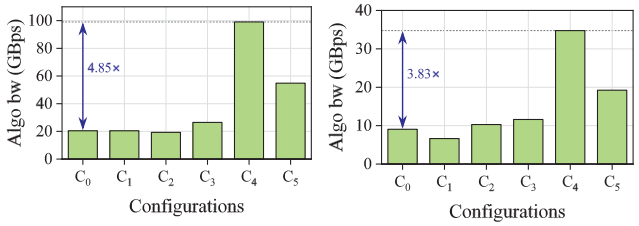


Fig. 1. The online tuning workflow of *TuCCL*.

TABLE III
MSCCL CONFIGURATION SETTINGS FOR ALLGATHER/ALLREDUCE OPERATORS IN DUAL-NODE 16-GPU TOPOLOGY

Config	Sketch	Protocol	NChannels	NThreads	BufferSize
C_0*(default)	Full-Mesh	Simple	1	512	4MB
C_1	Full-Mesh	Simple	1	512	8MB
C_2	Full-Mesh	Simple	4	512	4MB
C_3	Relay	Simple	1	512	8MB
C_4	Relay	Simple	4	512	4MB
C_5	Relay	Simple	4	512	8MB



(a) ALLGATHER (b) ALLREDUCE

Fig. 2. Bandwidth comparison using various MSCCL configurations on dual 8-GPU nodes.

A100 GPUs interconnected via NVSwitch and RoCE-enabled NICs. Table III compares six configuration variants (C0: default; C1–C5: manually tuned), each derived from different combinations of parameter settings, focusing on three critical parameters: `Sketch`, `NChannels`, and `BufferSize`. Figure 2 demonstrates that optimized configurations achieve 4.85x higher peak throughput for ALLGATHER and 3.83x for ALLREDUCE compared to default settings. These findings demonstrate that off-the-shelf parameter choices are not merely sub-optimal but can severely hamper communication efficiency. More importantly, they position configuration as a first-order performance determinant. Accordingly, we are motivated to design an automated, high-fidelity configuration–optimization framework that systematically explores the parameter space and consistently uncovers near-optimal settings.

Insight 3: Joint optimization of multidimensional configurations is critical for maximizing performance. Existing configuration tuning methods for communication libraries focus solely on optimizing backend primitive parameters. As demonstrated in Table II, synthesizer configurations (communication sketches) also critically influence algorithm performance, particularly link utilization during runtime. Suboptimal configurations can reduce utilization to as low as 0.47% under extreme cases. Furthermore, current approaches neglect the necessity of joint multidimensional optimization. Our experimental validation reveals that isolated single-dimensional tuning, as shown in Figure 2 (C1–C3: backend-only optimizations), yields marginal performance gains of up to 4.9%, insufficient to fully exploit algorithmic potential. In contrast, simultaneous optimization of both synthesizer and backend configurations (C4) produces multiplicative effects, achieving up to 4.85× greater improvements. These results confirm that multidimensional joint optimization is essential for maximizing algorithmic performance.

These results expose three critical limitations: **(1) Practical Barriers to Manual Tuning:** While effective in controlled settings, manual optimization is impractical for real-world deployment due to (i) intricate parameter interdependencies (e.g., `BufferSize` constraining `NChannels` efficiency) and (ii) the absence of systematic principles for sketch adaptation. **(2) Configuration Sensitivity:** Synthesis outcomes are highly sensitive to sketch and parameter choices, with default settings proving grossly suboptimal. **(3) Insufficiency of Single-Dimensional Optimization:** Isolated tuning of backend or synthesizer configurations alone yields limited gains, failing to unlock the full performance potential achievable through joint optimization. These findings underscore the necessity for automated, coordinated configuration optimization across system components in practical synthesizers—a critical gap addressed by our approach, which will be introduced in the next section.

III. DESIGN

A. Architecture and workflow of TuCCL

Figure 1 shows the architecture of *TuCCL* with its three key components:

- **Topology-Aware Sketch Generation** characterizes interactions between communication sketches and synthesizers, and generates high-performance communication primitives.
- **Hierarchical Configuration Optimization** models nonlinear relationships between low-level parameters and bandwidth utilization to guide systematic tuning.
- **Multi-phase Resource-Aware Configuration Optimization (MRACO)** adjusts multidimensional configuration and narrows the search space.

The main workflow of *TuCCL* comprises two phases: initialization and runtime execution. The initialization phase occurs prior to model training or inference, while the runtime phase operates dynamically during these processes.

Initialization Phase. During initialization, *TuCCL* probes the cluster to extract the current topology and automatically construct the lookup table. Specifically, *TuCCL* precomputes optimal configurations for all potential communication tasks. Consider a case involving a communication task termed $Task_1$: The optimizer extracts critical parameters for $Task_1$, including the operator type (ALLGATHER), input data size (4 GB), and communication topology (G_1). These parameters define the task's requirements for subsequent optimization. The communication topology undergoes heuristic pruning to eliminate redundant connections, resulting in a simplified topology (G_1'). This refined topology, paired with the operator type, is fed into the algorithm synthesizer to generate candidate communication algorithms. Bayesian optimization [16] iteratively evaluates these algorithms to identify the optimal configuration for $Task_1$—in this case, a parameter set specifying the protocol (Simple), nchannels (8), nthreads (512 units), and buffer size (4 MB). All optimized configurations are stored in a lookup table indexed by operator type, data size, and topology, completing the initialization phase.

Runtime Phase. When the training/inference pipeline executes $Task_1$: The backend receives the task descriptor (ALLGATHER, 4 GB, G_1) and queries the precomputed lookup table for the corresponding optimized configuration. The retrieved configuration is broadcast to all workers to ensure consistency across distributed nodes. The collective communication kernel executes using the synchronized parameters, enabling efficient data exchange. Before each collective operation, we select the configuration group with the shortest recent completion time. After the operation, we record the measured latency and update the lookup table. Topology may change during training because of packet loss or node failures. To handle these occasional topology changes, TuCCL reinitializes the lookup table before each aggregation and whenever training restarts.

B. Topology-Aware Sketch Generation

TuCCL 's initialization phase begins by acquiring a *task triplet* (operator type, input data size, and topology graph) from the training/inference workload. While operator types and data sizes are provided by the application layer, the topology graph must be dynamically constructed due to the application's inherent topology-agnosticism.

For Intra-Node Topology Discovery, within a node, GPUs interconnect via NVLink or PCIe, while NICs and GPUs form PCIe-based tree topologies. *TuCCL* reconstructs these topologies by parsing base address register (BAR) mappings exposed by the OS in '/sys/bus/pci/devices'. To resolve ambiguity in NVLink-CPU-PCIe hierarchies, we deploy CUDA-based peer-to-peer (P2P) microbenchmarks that empirically measure latency and bandwidth between all GPU pairs. These measurements disambiguate physical interconnects (e.g., distinguishing direct NVLink connections from indirect PCIe paths) and populate a weighted intra-node graph.

For Inter-Node Topology Construction, *TuCCL* abstracts physical complexities by modeling interconnects as NIC-to-NIC links. To avoid exhaustive pairwise measurements (prohibitively expensive in large clusters), we apply heuristic pruning: (1) Hierarchy-aware sampling: Prioritize NIC pairs within the same rack or subnet based on IP/MAC address patterns. (2) Traffic pattern filtering: Discard links with historically low utilization (logged from prior workloads). Remaining links are profiled using ICMP echo requests (latency) and NVIDIA Perftest-based bandwidth probes, constructing a sparse inter-node topology graph.

To generate guided sketch, the synthesized topology graph, annotated with link latencies and bandwidths, drives the generation of *guidance sketches*. These sketches encode constraints for the synthesizer (e.g., "prioritize NVLink paths for gradients $>$10 MB") and initialization parameters (e.g., chunk sizes aligned to PCIe packet granularity). By injecting this topology-aware guidance, the synthesizer avoids brute-force exploration. Here, some critical choices are as follows: (1) Empirical disambiguation: Combines static BAR parsing with dynamic P2P benchmarks to resolve hardware ambiguities. (2) Constraint embedding: Guidance sketches transform raw topology data into synthesizer-actionable constraints, bridging abstraction layers.

TuCCL also proposes to prune sketch. As demonstrated in Section II-C, synthesizer input configurations—particularly network connectivity patterns—significantly impact the performance of synthesized algorithms. A key insight emerges: reducing the number of inter-node connections not only shrinks the synthesizer's search space but surprisingly improves algorithm quality. We attribute this phenomenon to synthesizers' inability to model NIC time-division multiplexing, instead assuming uniform bandwidth allocation across all connections. This assumption deviates from practical NIC contention patterns, leading to suboptimal bandwidth utilization as connection density increases.

To mitigate this mismatch, we propose a three-stage heuris-

tic pruning method:

Stage 1: GPU-NIC Binding. For each node, map GPUs to their physically co-located NICs via PCIe switch hierarchies (retrieved from '/sys/bus/pci/devices'). Each GPU is assigned to the NIC sharing its PCIe root complex.

Stage 2: Bandwidth-Aware Grouping. Cluster NICs across nodes into *communication groups*, where all NICs in a group exhibit matched bandwidth capacities. Groups are formed by greedily pairing NICs with minimal intra-group bandwidth variance, ensuring no two NICs from the same node reside in one group.

Stage 3: Topology Transformation. Replace NIC-centric connections with GPU-centric equivalents. Within each group, create all-to-all GPU connections among nodes using the bound GPU-NIC pairs (e.g., if NIC A on Node 1 is grouped with NIC B on Node 2, connect GPU A_1 to GPU B_2). This abstracts NIC links into GPU-to-GPU edges for synthesizer compatibility.

Our design adopts a fundamentally different strategy from NVIDIA's. *TuCCL* performs topology-agnostic, logical-layer pruning: virtual communication subgraphs are removed according to congestion scores, and physical wiring is left untouched. By contrast, GPU–NIC binding approaches such as NVIDIA RAIL [17] hard-wire device mappings at the hardware layer and are unable to react adaptively at runtime.

C. Hierarchical Configuration Optimization

Modern GPU clusters exhibit intricate tradeoffs between communication protocols, hardware parallelism, and data granularity [15]. These trade-offs become especially important in distributed multi-GPU training environments where communication bandwidth, computational efficiency, and resource allocation need to be balanced [18]. Our framework addresses three interwoven challenges: **(1) Protocol-Bandwidth Dilemma**: The LL128 protocol, which uses 128-byte atomic operations, reduces header overhead but demands an excessive number of communication channels. This trade-off is particularly critical in the context of large-scale distributed systems, where optimizing bandwidth usage becomes crucial (Line 3, Algorithm 1: The n_s sampling considers these protocol constraints). **(2) Buffer Size Paradox**: Smaller chunks reduce network congestion but amplify protocol overhead, as more frequent transmission events are needed. This is a common issue in systems where the overhead of small packets diminishes the expected gains from reducing congestion. Our framework resolves this by dynamically adjusting the resource allocation (r_i) as shown in Line 7. **(3) Thread-Channel Deadlock**: When the number of threads N_t is insufficient relative to the number of channels N_c (specifically, when $N_t < 32N_c$), the warp schedulers experience stalls. This situation violates the SM occupancy rules, leading to inefficient hardware utilization. We address this deadlock issue by normalizing the thread-to-channel ratio, as indicated in Line 8 through the function $\hat{f}_i(x)$.

TABLE IV
PRIMITIVE CONFIGURATION PARAMETERS OF MSCCL

Parameter	Search Space
NChannels (N_c)	$2^i,\ i \in \{1, 2, 3, 4\}$
NThreads (N_t)	$n = 32 \times \mathbb{Z},\ \mathbb{Z} \in \{1, 2, 3, \ldots, 20\}$
Buffer size (B)	$n = 128 \times 2^{\mathbb{N}},\ \mathbb{N} \in \{1, 2, 3, \ldots, 15\}$

Algorithm 1 Multi-Stage Resource-Aware Configuration Optimization (MRACO)

Require: Configuration space $\mathcal{C}$, max resource $R_{\max} \in \mathbb{N}^+$, ratio $\eta > 1$
Ensure: Optimal configuration $x^* \in \mathcal{C}$
1: $s_{\max} \leftarrow \lfloor \log_\eta R_{\max} \rfloor$
2: **for** $s = 0$ **to** $s_{\max}$ **do**
3: $\quad n_s \leftarrow \lceil \eta^{s_{\max}-s} \rceil$
4: $\quad R_s \leftarrow \lfloor R_{\max} \cdot \eta^{-s} \rfloor$
5: $\quad \mathcal{S}_s \leftarrow \textsc{Sample}(n_s, \mathcal{C})$
6: $\quad$ **for** $i = 0$ **to** s **do**
7: $\quad\quad r_i \leftarrow R_s \cdot \eta^i$
8: $\quad\quad \hat{f}_i(x) \leftarrow \frac{1}{r_i}\mathbb{E}\left[\textsc{ProfilePerf}(x, r_i)\right]$ for all $x \in \mathcal{S}_s$
9: $\quad\quad \mathcal{S}_s \leftarrow \arg\max_{\mathcal{S}} \left\{ \sum_{x \in \mathcal{S}} \hat{f}_i(x) \mid |\mathcal{S}| = \lceil n_s \eta^{-i} \rceil \right\}$
10: $\quad\quad x^* \leftarrow \arg\max_{x \in \{x^*\} \cup \mathcal{S}_s} \mathbb{E}\left[\textsc{ProfilePerf}(x, R_{\max})\right]$
11: $\quad$ **end for**
12: **end for**
13: **return** x^*

We structure the search space $\mathcal{C}$ in a hierarchical manner, which reflects the inherent complexities and constraints of the system:

$$\mathcal{C} = \underbrace{\{LL, LL128, Simple\}}_{p} \times \underbrace{2^{i^+}}_{N_c} \times \underbrace{32\mathbb{Z}^+}_{N_t} \times \underbrace{2^{\mathbb{N}^+}}_{B} \quad (1)$$

In Eq. 1, the search space $\mathcal{C}$ is defined by four key dimensions: 1) Communication policy p with three discrete strategies; 2) Channel count N_c constrained to powers of two [1, 2, 4, 8] to match hardware multiplexing capabilities; 3) Thread count N_t quantized in 32-integer multiples (32-640) to align with warp-level execution units; 4) Buffer size B restricted to powers of two (256-8,388,608 bytes) to optimize memory alignment and access patterns. This formulation ensures configuration validity while preserving essential hardware-specific constraints: N_t granularity prevents partial warp execution, and B's exponential scaling avoids misaligned memory transactions. The detailed configuration search space is summarized in Table IV.

D. Multi-Stage Resource-Aware Configuration Optimization

To efficiently identify the optimal configuration from the vast parameter space, this paper proposes a *multi-stage resource-aware configuration optimization* (MARCO) algorithm, as shown in Algorithm 1. This algorithm syn-

ergizes *Tree-structured Parzen Estimator's*[2] (TPE's) adaptive exploration-exploitation balance with Hyperband's resource efficiency. TPE-based modeling addresses the high-dimensionality challenge by constructing likelihood distributions $g(x)$ and $l(x)$ from historical trials (Eq. 2), prioritizing configurations where the *expected improvement* (Eq. 3)—the product of exploration potential ($g(x)/l(x)$) and performance gain ($\mu_g - \mu_l$)—is maximized. Unlike conventional Bayesian methods requiring Gaussian process priors, TPE's non-parametric approach adapts to discontinuous parameter spaces (e.g., protocol switches between LL/LL128), while Hyperband pruning (Eq. 4) eliminates low-yield candidates early, redirecting resources to promising regions. The convergence criteria (Eq. 5) ensure termination only when both performance gain and parameter distribution stabilize, preventing premature stops. The MACRO algorithm employs a three-stage architecture that systematically transitions from broad configuration space exploration to targeted optimization, as detailed below:

a) Stage 1: TPE-Based Exploration (Lines 2-5): To systematically balance exploration and exploitation, we employ a TPE framework that probabilistically models high-performance configurations. This is achieved through Gaussian mixture models to estimate the likelihood of different configurations based on historical data:

$$l(x) = \sum_{i:\eta_i<\eta^*} \exp\left(-\frac{\|x - x_i\|^2}{2\sigma_p^2}\right) \tag{2}$$

Here, x_i represents the configurations, and η_i is the performance metric associated with configuration x_i. σ_p is a variance term that adapts depending on the protocol. For example, LL128, which experiences volatile bandwidth, will have a wider variance than simpler protocols. The acquisition function used to select the next configuration is given by:

$$x_{n+1} = \arg\max_{x\in\mathcal{C}} \underbrace{\frac{g(x)}{l(x)}}_{\text{Exploration}} \cdot \underbrace{(\mu_g - \mu_l)}_{\text{Exploitation}} \tag{3}$$

In Eq. 3, $g(x)$ represents the likelihood function for high-performance configurations, and $l(x)$ is the likelihood for low-performance configurations. The terms μ_g and μ_l are the expected values of the performance metrics for the high and low-performing configurations, respectively.

b) Stage 2: Hyperband Pruning (Lines 6-9): To reduce the search space and avoid overfitting, early elimination of poorly performing configurations is achieved using Hyperband. This approach uses a dynamic resource allocation strategy:

$$\begin{aligned} \mathcal{S}_s^{(i)} &= \left\lceil \frac{n_s}{\eta^i} \right\rceil \text{ top configs via} \\ \hat{f}_i(x) &= \frac{1}{r_i}\mathbb{E}[\textsc{ProfilePerf}(x, r_i)] \end{aligned} \tag{4}$$

[2]TPE is a Bayesian optimization method that models conditional densities for good vs. poor configurations and selects new trials favoring regions with higher probability of good outcomes

In Eq. 4, r_i represents the resource allocation at stage i, and $\hat{f}_i(x)$ is the normalized performance function. By allocating resources according to the formula $r_i = R_s\eta^i$, we prevent overfitting to transient performance metrics and maintain generalization across different configurations. Following the pruning phase, convergence detection mechanisms monitor optimization trajectories to determine termination criteria through quantifiable statistical thresholds.

c) Stage 3: Convergence Detection (Line 10): Optimization is terminated when either of the following conditions is met:

$$\exists t : \frac{\eta_t - \eta_{t-\Delta}}{\eta_{t-\Delta}} < 0.01 \ \wedge \ D_{KL}(p_t \| p_{t-\Delta}) < 0.1 \tag{5}$$

Here, D_{KL} represents the Kullback-Leibler divergence between the parameter distributions at two consecutive time steps, t and $t-\Delta$. This condition ensures that the optimization process has converged, meaning there are minimal changes in both the performance metric and the configuration distribution.

The MRACO framework provides the following theoretical guarantees: **(1) Efficiency**: The total number of trials will not exceed $(s_{\max}+1)R_{\max}$ (Line 2's bracket scheduling), ensuring the optimization process remains efficient. **(2) Feasibility**: The budget constraints are respected for all stages $s \leq s_{\max}$, with $R_s\eta^s \leq R_{\max}$ (Line 3's budget allocation). This ensures that the resource allocation at each stage does not exceed the available resources. **(3) Normalization**: The performance function $\hat{f}_i(x)$ is normalized according to r_i^{-1} to adjust for varying resource levels across different stages. **(4) Optimality**: There exists an optimal configuration x^* in the set $\mathcal{S}_{s^*}$ at the final stage s^*, as guaranteed by Line 10's elite retention.

IV. Evaluation

A. Experiment Setup

In this section, we conduct extensive experiments to evaluate the performance of *TuCCL*, and the comparison results highlight the following observations:

- TuCCL improves the bandwidth by 1.75x–11.49x over state-of-the-art systems.
- TuCCL's MRACO module can reduce optimization time by up to 90.2% while preserving solution quality.
- For end-to-end training, *TuCCL* is 1.22x-2.52x faster across diverse model scales.

Implmentation. The *TuCCL* framework is architected to operate across both preparation and execution phases of training/inference tasks, divided into two coordinated components: an initialization optimizer and a runtime configurator. **(1) Initialization Phase:** The optimizer core is implemented in Python, leveraging Shell scripting to automate hardware topology discovery (e.g., GPU/NIC interconnects) and environment profiling. To empirically evaluate the performance of synthesized communication primitives, we developed CUDA-based microbenchmarks that replicate real-world collective patterns (e.g., AllReduce, AllGather) under configurable parameters. These benchmarks iteratively test candidate sketches across topology permutations, generating

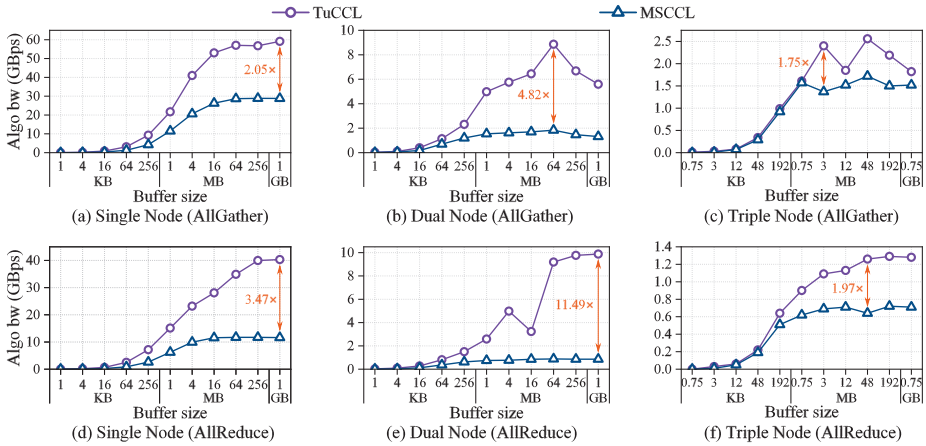


Fig. 3. Bandwidth improvements with *TuCCL* configuration tuning in V100 GPU clusters: single, dual, and triple-node topologies.

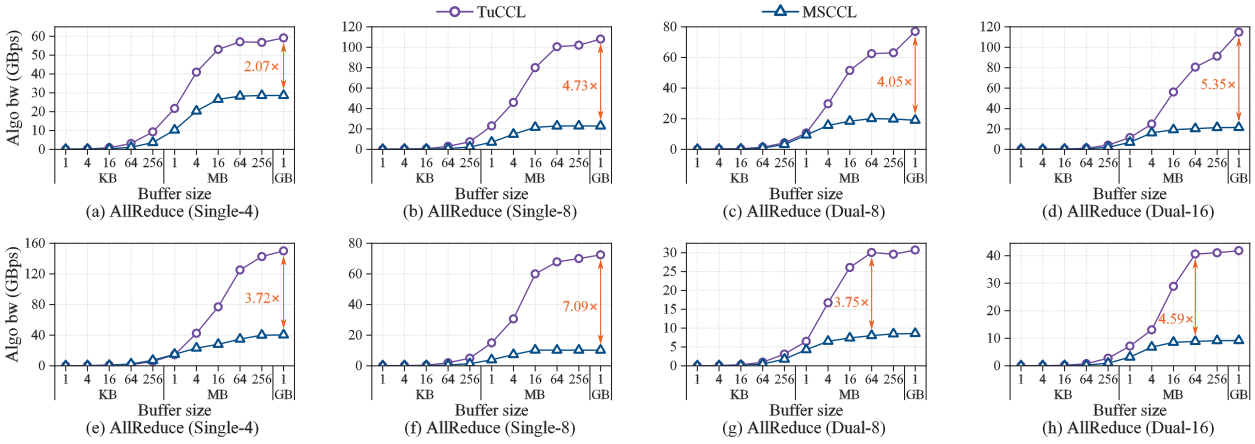


Fig. 4. Bandwidth improvements with *TuCCL* configuration tuning in A100 GPU clusters.

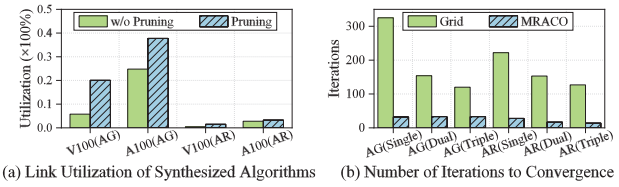


Fig. 5. Sketch-optimized link utilization and MRACO algorithm efficiency.

a performance database to guide subsequent optimizations. **(2) Runtime Phase.** Building on the MSCCL framework, we extend its backend with a C++-based dynamic configuration engine. This engine performs two critical functions: *a) Configuration Lookup:* A hash-mapped registry stores pre-optimized sketch-configuration pairs, indexed by topology signatures and collective operation types. *b) Distributed Broadcast:* Upon identifying the optimal configuration, we utilize MPI's native broadcast operator (MPI_Bcast) to synchronize parameters across all nodes, ensuring sub-millisecond latency and fault tolerance via checksum verification. To minimize runtime overhead, the lookup table is preloaded during task initialization, while configuration switching is atomic and non-blocking.

Testbed. We constructed different GPU server clusters for evaluation. The first cluster comprises NVIDIA V100 GPUs, where each server is equipped with an SXM2 module housing four V100 GPUs interconnected via six second-generation NVLink bridges, delivering an aggregate bidirectional bandwidth of 300 GB/s [19]. These servers are networked using Mellanox CX5/CX6 adapters with 100 Gb/s bandwidth [20], connected to the SXM2 modules through PCIe switches.

To further assess the impact of communication sketch and configuration optimizations in high-performance GPU systems, we extended our experiments to a DGX-A100 cluster. Each server in this cluster integrates eight A100 80GB GPUs, each providing 300 GB/s of bandwidth, interconnected via six NVSwitches operating at 600 GB/s. Additionally, every server features four network interface cards (NICs) with a combined bandwidth of 400 Gb/s. The cluster employs a CLOS fabric with top-of-rack switches; each ToR provides 400 Gbps/s uplink capacity and 200 Gbps/s downlink capacity.

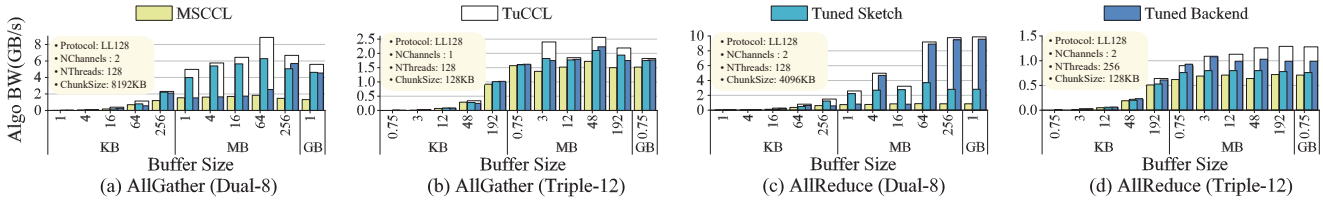


Fig. 6. Bandwidth gains from single-dimensional and joint optimization in V100 clusters.

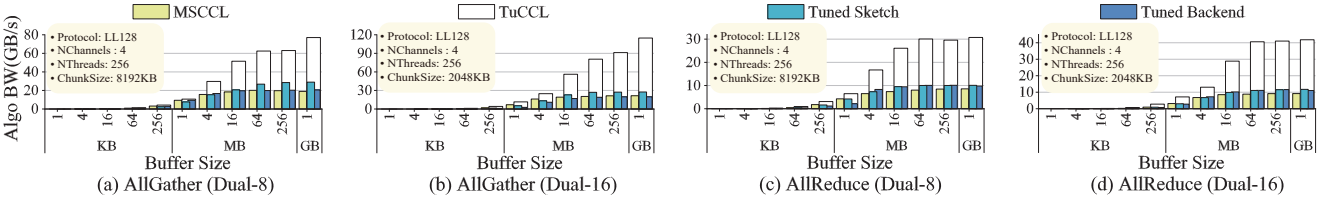


Fig. 7. Bandwidth gains from single-dimensional and joint optimization in A100 Clusters.

Baseline Systems. We evaluate *TuCCL* against MSCCL [14], a state-of-the-art framework supporting both synthesized and expert-designed algorithms with built-in tuning capabilities. For fairness, MSCCL's auto-tuner is fully enabled, and it applies a uniform global configuration across all communication tasks (e.g., ALLREDUCE, ALLGATHER). In contrast, *TuCCL* employs *fine-grained, task-specific configurations* optimized per collective operator, topology, and buffer size.

Furthermore, performance evaluation requires a synthesizer to validate opportunities for communication sketches adjustment. For this purpose, we employ the advanced TACCL framework as a unified synthesizer to ensure equitable evaluation of configuration optimizations across both *TuCCL* and MSCCL systems. Our findings demonstrate that *TuCCL*'s integrated configuration optimization framework achieves maximum performance potential when applied to algorithms generated through the same TACCL workflow.

B. Communication Micro-Benchmarks

We evaluate *TuCCL* 's performance gains across NVIDIA V100 and A100 clusters to eliminate hardware-specific biases and explore its efficacy in high-performance scenarios. Our analysis focuses on two critical collective primitives, ALLGATHER and ALLREDUCE, under varying hardware topologies and data sizes (1 KB–1 GB). ALLREDUCE can be implemented as REDUCESCATTER plus ALLGATHER, while Broadcast is a special case of ALLGATHER. Hence, performance gains for ALLREDUCE and ALLGATHER generalize to other collectives. Following MSCCLang, TACCL, and TACOS, we concentrate on these two primitives and omit redundant micro-benchmarks.

Performance on V100 Clusters. *TuCCL* consistently outperforms the baseline MSCCL (using uniform default configurations) across all communication primitives. As shown in Figure 3, we evaluate *TuCCL* 's performance improvements for two operators, ALLGATHER and ALLREDUCE, across

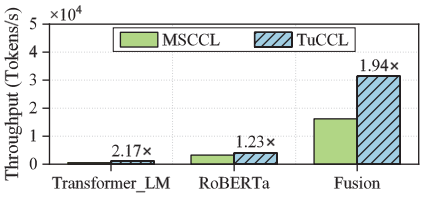


Fig. 8. Training throughput for different models between *TuCCL* and MSCCL.

three network topologies. The results demonstrate speedups of 1.75x (three-server cluster) to 4.82x (dual-node setup) for ALLGATHER, and 1.97x to 11.49x for ALLREDUCE. These gains stem from *TuCCL* 's topology-aware path pruning, which reduces redundant inter-node links by up to 75% (e.g., decreasing NIC connections from 8 to 2 per node in three-server configurations), and its adaptive buffer allocation strategies that minimize PCIe contention.

Performance on A100 Clusters. Figure 4 illustrates *TuCCL* 's scalability in high-performance environments. The labels Single-4/8, Dual-8/16, etc., denote configurations with 4/8 GPUs per server and 1-2 servers. Across all topologies, *TuCCL* achieves superior bandwidth compared to baselines, with peak improvements of 5.35x for ALLGATHER and 7.09x for ALLREDUCE. Notably, the larger gains on A100 GPUs (versus V100) highlight *TuCCL* 's ability to leverage NVSwitch's enhanced bisection bandwidth (600 GB/s) through parallel transfers and optimized buffer sizing.

Link Utilization. Beyond bandwidth evaluation, we conduct experimental analysis of link utilization to demonstrate the necessity of communication sketch optimization. As shown in Figure 5(a) , *TuCCL* 's sketch pruning adjustments comprehensively improve link utilization. Our metric measures aggregate link usage during the entire algorithm execution, which is constrained by data dependencies (e.g., sequential execution of primitives sharing the same buffer) and optimization biases favoring intra-machine high-bandwidth communication. These

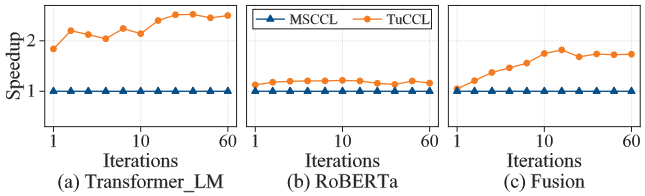


Fig. 9. *TuCCL* training iteration acceleration across models vs. MSCCL.

factors inherently increase network idle time. Nevertheless, *TuCCL* achieves maximum link utilization improvements of 18.45% and 34.46% for the evaluated operators on V100/A100 clusters, even under these constraints.

Convergence Speed and Scalability. Our framework significantly accelerates convergence while maintaining scalability. As shown in Figure 5(b), *TuCCL* 's MRACO algorithm achieves rapid convergence and demonstrates *near-linear* scalability as system size increases. Compared to conventional grid search methods, MRACO reduces optimization time by up to 90.2% while preserving solution quality.

C. Ablation Study on Configuration Synergy

To isolate the impact of sketch synthesis versus runtime configuration tuning, we conduct ablation tests across V100 and A100 clusters (Figure 6–7). Figure 6 and 7 present the configuration groups that achieve the maximum bandwidth. On the V100 cluster, optimizing sketches and configurations independently improves ALLGATHER performance by 3.2x and 3.5x, respectively, while ALLREDUCE gains 2.3x and 2.5x. However, joint optimization amplifies gains to 4x (ALLGATHER) and 5x (ALLREDUCE), demonstrating multiplicative synergies.

On the A100 cluster, isolated adjustments yield marginal improvements (1.27x–1.52x), as next-generation hardware intensifies parameter interdependencies (e.g., NVSwitch bandwidth saturation masks chunk size optimizations). In contrast, joint optimization resolves these couplings—aligning chunk granularity with buffer allocation and protocol selection—to achieve 4.85x speedups. This validates that *TuCCL* 's co-optimization strategy is essential for unlocking performance in modern, high-bandwidth environments where single-dimensional tuning fails.

D. End-to-End Training

To demonstrate *TuCCL*'s effectiveness in accelerating real-world distributed DNN training, we deploy three application-specific models: Transformer-LM-247M [21], RoBERTa-125M [22], and Fusion-270M [23]. Figure 8 compares their training throughput. *TuCCL* outperforms MSCCL implementations with speedups of 2.17x, 1.23x, and 1.94x, respectively. This improvement stems from *TuCCL* 's reduced communication-computation interference, which alleviates resource contention in streaming multiprocessors (SMs) that slows iteration progress in MSCCL. As shown in Figure 9, *TuCCL* also demonstrates stable acceleration across training iterations, achieving average speedups of 1.22x–2.52x.

V. RELATED WORK

Synthesis-Driven Collective Algorithm Generation. Emerging synthesis frameworks automate the discovery of topology-optimal collective algorithms. SCCL [12] formulates bandwidth-latency tradeoffs as quantifier-free SMT constraints, systematically exploring Pareto-optimal solutions. TACCL [3] combines sketch-guided pruning with integer linear programming (ILP) to synthesize algorithms for multi-node clusters while accounting for network hierarchy. However, existing synthesis methods assume static network configurations and overlook two critical aspects addressed in our work: (1) automatic exploration of topology-aware sketch spaces for synthesis inputs, and (2) runtime adaptive optimization of communication primitives based on dynamic system states.

Optimizing CCL for specific systems. To empower developers with flexible algorithm customization, MSCCLang [2] proposes a domain-specific language (DSL) that decouples high-level collective semantics from low-level implementation details. TCCL [24] provides a specialized library to optimize collective communication in PCIe systems. Complementary efforts focus on designing latency-hiding communication primitives through aggressive pipeline parallelism and computation-over-communication strategies [25]–[30]. While these works improve programmability, they require manual expertise to achieve peak performance across diverse network topologies. SwitchML [31] and ATP [32] use programmable switches to implement in-network computing, reducing the additional communication overhead in PS architectures for completing the REDUCE operation and increasing overall training throughput. However, these works still assume that default communication library configurations are expert configurations and do not consider the optimization and adjustment of such configurations. Yet, this setting often fails to achieve the theoretical optimal performance of the methods.

VI. CONCLUSION

In this paper, we present *TuCCL*, a framework that jointly optimizes communication strategies and runtime configurations to overcome the limitations of labor-intensive manual intervention, overreliance on predefined input and suboptimal isolated configuration in existing works. *TuCCL* proposes topology-aware sketches and hierarchical configuration optimization, and coordinates multidimensional configuration via multi-phase resource-aware optimization scheme. Experiments have shown *TuCCL*'s superior performance in improving bandwidth and training speed, and validated its faster configuration search scheme and robustness.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China under Grant Nos. 62432003, U22B2005 and 62402487, as well as the LiaoNing Revitalization Talents Program under Grant No. XLYC2403086. Additional support is provided by the Basic Research Program of Shenzhen under Grant No. JCYJ20220531100804009.

REFERENCES

[1] S. Agarwal, H. Wang, S. Venkataraman, and D. Papailiopoulos, "On the utility of gradient compression in distributed training systems," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 652–672, 2022.

[2] M. Cowan, S. Maleki, M. Musuvathi, O. Saarikivi, and Y. Xiong, "MSCClang: Microsoft Collective Communication Language," in *Proc. ACM ASPLOS*, vol. 2, 2023, pp. 502–514.

[3] A. Shah, V. Chidambaram, M. Cowan, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, O. Saarikivi, and R. Singh, "TACCL: Guiding collective algorithm synthesis using communication sketches," in *Proc. 20th USENIX Symp. Networked Syst. Des. Implement. (NSDI)*, 2023, pp. 593–612.

[4] W. Deng, J. Pan, T. Zhou, D. Kong, A. Flores, and G. Lin, "Deeplight: Deep lightweight feature interactions for accelerating ctr predictions in ad serving," in *Proc. of ACM WSDM*, 2021, pp. 922–930.

[5] NVIDIA Corporation, "NVIDIA Collective Communications Library (NCCL)," 2024, https://github.com/NVIDIA/nccl.

[6] Advanced Micro Devices, Inc, "ROCm Communication Collectives Library (RCCL)," 2024, https://github.com/ROCm/rccl.

[7] A. A. Awan, K. Hamidouche, A. Venkatesh, and D. K. Panda, "Efficient large message broadcast using nccl and cuda-aware mpi for deep learning," in *Proceedings of the 23rd European MPI Users' Group Meeting*, 2016, pp. 15–22.

[8] K. Lakhotia, K. Isham, L. Monroe, M. Besta, T. Hoefler, and F. Petrini, "In-network allreduce with multiple spanning trees on polarfly," in *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, 2023, pp. 165–176.

[9] D. De Sensi, E. C. Molero, S. Di Girolamo, L. Vanbever, and T. Hoefler, "Canary: Congestion-aware in-network allreduce using dynamic trees," *Future Generation Computer Systems*, vol. 152, pp. 70–82, 2024.

[10] D. De Sensi, S. Di Girolamo, S. Ashkboos, S. Li, and T. Hoefler, "Flare: Flexible in-network allreduce," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–16.

[11] X. Liu, B. Arzani, S. K. R. Kakarla, L. Zhao, V. Liu, M. Castro, S. Kandula, and L. Marshall, "Rethinking machine learning collective communication as a multi-commodity flow problem," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 16–37.

[12] Z. Cai, Z. Liu, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, and O. Saarikivi, "Synthesizing Optimal Collective Algorithms," in *Proc. of ACM PPoPP*, 2021, pp. 62–75.

[13] Z. Wang, Y. Zhou, C. Tian, X. Wang, and X. Chen, "Afnfa: An approach to automate nccl configuration exploration," in *Proc. of the ACM APNet*, 2023, pp. 204–205.

[14] Microsoft, "MSCCL," https://github.com/microsoft/msccl, 2022.

[15] Z. Cai, Z. Liu, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, and O. Saarikivi, "Synthesizing optimal collective algorithms," in *Proc. 26th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2021, pp. 62–75.

[16] X. Wang, Y. Jin, S. Schmitt, and M. Olhofer, "Recent Advances in Bayesian Optimization," *ACM Comput. Surv.*, vol. 55, no. 13s, pp. 1–36, 2023.

[17] W. Wang, M. Ghobadi, K. Shakeri, Y. Zhang, and N. Hasani, "Optimized network architectures for training large language models with billions of parameters," 2023.

[18] S. Shi, X. Chu, and B. Li, "Exploiting Simultaneous Communications to Accelerate Data Parallel Distributed Deep Learning," in *Proc. IEEE INFOCOM 2021*. IEEE, 2021, pp. 1–10.

[19] NVIDIA Corporation, "Nvidia NVLink and NVSwitch," 2021, https://www.nvidia.com/en-us/data-center/nvlink.

[20] NVIDIA Corporation, "Nvidia ConnectX-5," 2024, https://docs.nvidia.com/networking/display/connectx5en.

[21] A. Baevski and M. Auli, "Adaptive input representations for neural language modeling," *arXiv preprint arXiv:1809.10853*, 2018.

[22] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[23] A. Fan, M. Lewis, and Y. Dauphin, "Hierarchical neural story generation," *arXiv preprint arXiv:1805.04833*, 2018.

[24] H. Kim, J. Ryu, and J. Lee, "TCCL: Discovering Better Communication Paths for PCIe GPU Clusters," in *Proc. of ACM ASPLOS*, 2024, pp. 999–1015.

[25] S. Cho, H. Son, and J. Kim, "Logical/Physical Topology-Aware Collective Communication in Deep Learning Training," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*. IEEE, 2023, pp. 56–68.

[26] S. Wang, J. Wei, A. Sabne, A. Davis, B. Ilbeyi, B. Hechtman, D. Chen, K. S. Murthy, M. Maggioni, Q. Zhang *et al.*, "Overlap Communication with Dependent Computation via Decomposition in Large Deep Learning Models," in *Proc. of ACM ASPLOS*, 2022, pp. 93–106.

[27] A. Jangda, J. Huang, G. Liu, A. H. N. Sabet, S. Maleki, Y. Miao, M. Musuvathi, T. Mytkowicz, and O. Saarikivi, "Breaking the Computation and Communication Abstraction Barrier in Distributed Machine Learning Workloads," in *Proc. ACM ASPLOS*, 2022, pp. 402–416.

[28] K. Mahajan, C.-H. Chu, S. Sridharan, and A. Akella, "Better Together: Jointly Optimizing ML Collective Scheduling and Execution Planning using SYNDICATE," in *Proc. of USENIX NSDI*, 2023, pp. 809–824.

[29] Z. Jiang, H. Lin, Y. Zhong, Q. Huang, Y. Chen, Z. Zhang, Y. Peng, X. Li, C. Xie, S. Nong *et al.*, "MegaScale: Scaling Large Language Model Training to More Than 10, 000 GPUs," in *Proc. of USENIX NSDI*, 2024, pp. 745–760.

[30] Y. Wu, Y. Xu, J. Chen, Z. Wang, Y. Zhang, M. Lentz, and D. Zhuo, "Mccs: A service-based approach to collective communication for multi-tenant cloud," in *Proc. of ACM SIGCOMM*, 2024, pp. 679–690.

[31] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtárik, "Scaling Distributed Machine Learning with In-Network Aggregation," in *Proc. of USENIX NSDI*, 2021, pp. 785–808.

[32] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift, "ATP: In-network Aggregation for Multi-tenant Learning," in *Proc. of USENIX NSDI*, 2021, pp. 741–761.