ResCCL: Resource-Efficient Scheduling for Collective Communication

Tongrui Liu[1†], Chenyang Hei[1†], Fuliang Li[1*], Chengxi Gao[3], Jiamin Cao[2], Tianshu Wang[2]
Ennan Zhai[2], Xingwei Wang[1*]
[1]*Northeastern University* [2]*Alibaba Cloud*
[3]*Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences*

ABSTRACT

As distributed deep learning training (DLT) systems scale, collective communication has become a significant performance bottleneck. While current approaches optimize bandwidth utilization and task completion time, existing communication libraries (CCLs) backends fail to efficiently manage GPU resources during algorithm execution, limiting the performance of advanced algorithms. This paper proposes *ResCCL*, a novel CCL backend designed for **R**esource-**E**fficient **S**cheduling to address key limitations in current systems. ResCCL enhances execution efficiency by optimizing scheduling at the primitive level (*e.g.,* send and recvReduceCopy), enabling flexible thread block (TB) allocation, and generating lightweight communication kernels to minimize runtime overhead. Our approach tackles the global scheduling problem, reduces idle TB resources, and enhances communication bandwidth. Evaluation results demonstrate that ResCCL achieves up to 2.5× improvement in bandwidth performance compared to both NCCL and MSCCL. It reduces SM resource overhead by 77.8% and increases TB utilization by 41.6% while running the same algorithms. In end-to-end DLT, ResCCL boosts Megatron's throughput by up to 39%.

CCS CONCEPTS

• **Networks** → **Network design and planning algorithms**;

KEYWORDS

Collective Communication, Deep Learning, Scheduling Algorithm

ACM Reference Format:
Tongrui Liu, Chenyang Hei, Fuliang Li, Chengxi Gao, Jiamin Cao, Tianshu Wang, Ennan Zhai, Xingwei Wang. 2025. ResCCL: Resource-Efficient Scheduling for Collective Communication. In *Proceedings of ACM SIGCOMM 2025 Conference (SIGCOMM '25).* ACM, Coimbra, Portugal, 16 pages. https://doi.org/10.1145/3718958.3750514

[†]Tongrui Liu and Chenyang Hei contributed equally to this paper.
[*]Co-corresponding authors.

SIGCOMM '25, September 8-11, 2025, Coimbra, Portugal
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1524-2/25/09
https://doi.org/10.1145/3718958.3750514

1 INTRODUCTION

Collective communication is a specific pattern among workers within a collaborative group, used to synchronize gradients, parameters, and optimizer states across GPUs in distributed deep-learning training (DLT) [4, 23, 30, 31]. However, as distributed training systems continue to scale, collective communication has increasingly become a bottleneck [28, 32, 37]. For example, Domino [36] reports that, even on DGX-H100 systems linked by 400Gb/s InfiniBand, communication overhead still consumes 17%–43% of the total iteration time when training GPT-3-13B end-to-end.

The performance of collective communication is crucial. Existing efforts mainly focus on designing more efficient *collective communication algorithms* to improve the performance. A collective communication algorithm defines the data transfer plan between GPUs. Open-source collective communication libraries (CCLs), such as NCCL [13] and RCCL [1], provide standardized algorithms (*e.g.,* ring and double binary tree [10]). To address various collectives and topologies, advanced collective communication algorithm synthesizers (*e.g.,* SCCL [2], TACCL [33], and TECCL [29]) automatically synthesize near-optimal algorithms by mathematical modeling and solving. However, these efforts primarily focus on algorithm optimization, without taking into account how to execute the algorithm efficiently.

We observe that the efficient execution of collective communication algorithms (implemented by the *collective communication backend*) is crucial for performance. A collective communication backend translates the algorithm to hardware instructions and schedules the GPU thread blocks (TBs) and CPU threads to utilize intra-server (*e.g.,* NVLink) and inter-server (*e.g.,* RDMA) bandwidths. Without optimization at the execution level, even theoretically optimal algorithms may perform poorly. For example, a naive backend lacking execution optimization, when running a standard hierarchical algorithm, may cause the transmission plan for high-bandwidth intra-machine links to frequently wait for inter-machine transmissions. This leads to substantial bandwidth underutilization, resulting in poor algorithm performance.

Generally, a collective communication backend should satisfy two key requirements. First, it should be able to execute any collective communication algorithm efficiently. Second, it should maintain low overhead so that computation is not affected, meaning it should use minimal GPU resources (*i.e.,* streaming multi-processors, SMs). SMs are essential on a GPU, as they are used for both computation and communication. When communication occupies more SMs, fewer are available for computation, which can negatively affect end-to-end performance [3, 11, 12, 22]. In practice, achieving high performance and low overhead requires a careful balance based on specific needs. For example, if a model supports overlapping computation and communication, and if computation is critical, we should focus on reducing communication overhead while enhancing computation performance. Conversely, if communication is more important, our priority should shift to maximizing communication performance.

Current CCL backends, such as NCCL [13], RCCL [1], and MSCCL [7] (which is based on NCCL), do not meet the above two requirements. The root cause is that they employ static resource allocation and scheduling mechanisms, leading to inefficient utilization of bandwidth and SM resources for various collective algorithms. We have identified several bottlenecks in them, which we summarize into the following two key points:

(1) High resource overhead. Current backends use static TB allocation, assigning one TB per GPU peer link for data transport. However, not all links are active throughout the entire algorithm execution, and often only a small fraction of links are engaged in data transfer. This results in many TBs remaining idle for long periods, leading to significant resource wastage. Additionally, existing backends can only achieve low-level optimizations (*e.g.,* pipelining) by consuming extra resources to open additional transmission channels.

(2) Performance degradation. The backend fails to address the global optimal scheduling problem for complete data transfers, executing the algorithm lazily (requiring multiple iterations of communication algorithms during a single data synchronization to complete full data transfer). This results in significant "bubbles" in the algorithm's execution pipeline during the data transfer process. Simultaneously, due to the lack of proper coordination among TBs in these backends, unnecessary dependencies and contention are introduced between TBs, which significantly hampers execution performance. Moreover, current CCLs embed an interpreter at runtime to dynamically parse communication algorithms, including data routing for each step and fixed TB allocation. However, this continuous loading and memory reads during execution significantly reduce algorithm efficiency.

Our approach: ResCCL. In this paper, we propose ResCCL, a CCL backend designed for **R**esource-**E**fficient **S**cheduling. ResCCL ensures that existing optimized collective algorithms achieve their theoretical peak bandwidth through backend-level execution optimizations. More specifically, ResCCL proposes the following three components: **(1) Primitive-level execution scheduling optimization:** To address the performance degradation caused by the lack of global optimal TB resource scheduling and poor collaboration between TBs, ResCCL provides a global dependency analysis for complete data transfer. It also designs fine-grained execution scheduling optimizations at the primitive level (*e.g.,* send, recvReduceCopy *etc.*) to minimize execution pipeline bubbles and approach global optimal scheduling. **(2) Flexible TB resource allocation mechanism:** To address the issue of excessive TB resource consumption, we propose a more flexible TB resource allocation mechanism that overcomes the rigid, communication-connection-based resource distribution in existing communication libraries. By analyzing the transmission patterns between communication pairs, we integrate idle resources while ensuring high link utilization

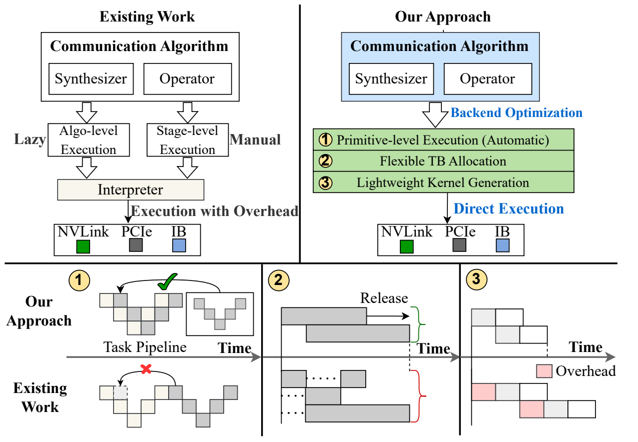


Figure 1: Comparison between our approach (ResCCL) and the state-of-the-art efforts.

and efficient communication bandwidth. **(3) Lightweight kernel generation:** ResCCL directly generates executable communication kernels, eliminating the overhead of runtime interpretation. This lightweight kernel generation ensures both the correctness and efficiency of the synthesized kernels while minimizing system overhead.

Figure 1 illustrates the design comparison between ResCCL and existing CCL, highlighting how ResCCL effectively addresses and overcomes their current limitations.

This paper makes the following contributions.

- We identified and analyzed key performance bottlenecks in existing CCL backends, including inefficiencies in scheduling, resource allocation, and runtime execution.
- We developed ResCCL, a resource-efficient scheduling backend for the collective communication library that optimizes algorithm execution and significantly improves communication bandwidth and thread resource utilization.
- Our evaluation demonstrates that ResCCL outperforms existing solutions, achieving up to 2.5× higher bandwidth and a 39% improvement in end-to-end training throughput compared to NCCL and MSCCL. It also reduces TB consumption by up to 77.8% and increases TB utilization by 41.6%.

This work does not raise any ethical issues.

2 BACKGROUND AND MOTIVATION

2.1 Collective Communication Libraries

Collective communication is a critical component of distributed deep learning [20, 25, 44]. To complete collective communication, each worker must cooperate in executing a specific communication pattern, often referred to as a communication operator, such as AllGather, ReduceScatter, or AllReduce. The efficiency of these operators relies on high-performance collective algorithms that aim to minimize data transfer.

Current collective communication methods can be categorized into two types: standard algorithms and custom algorithms. Typical

Table 1: Global link utilization during the execution of existing expert and synthesized algorithms. MS represents MSCCLang, TA/TE correspond to TACCL and TECCL, and AR/AG refer to the AllReduce and AllGather operators.

Topo Scale	MS-AG	MS-AR	TA-AG	TA-AR	TE-AG
1 Server (8 GPUs)	76.7%	71.0%	51.6%	45.7%	52.7%
2 Servers (16 GPUs)	67.5%	61.8%	34.3%	31.8%	33.2%
4 Servers (32 GPUs)	66.8%	46.1%	44.6%	41.9%	38.1%

open-source communication libraries [21, 26, 41], such as NCCL and RCCL, provide fixed implementations of vendor-specific optimized standard algorithms, along with a dedicated backend execution. Custom algorithms is further divided into two categories: automatically generated algorithms proposed by synthesizers and topology-specific algorithms based on expert knowledge [17]. Algorithm developers can use synthesizers or MSCCLang [8, 15], in conjunction with MSCCL, to flexibly implement and execute algorithms. However, due to MSCCL's reliance on the NCCL backend, custom algorithms suffer from significant performance and resource overhead issues (as detailed in §2.2).

Conceptual clarification. Communication operators implemented in communication libraries consist of two components: communication algorithms and backends. The communication algorithm represents the data transfer plan between GPUs, while the backend is responsible for scheduling TBs to transport communication data under the guidance of the algorithm. During backend execution, each TB in the GPU executes a communication kernel to complete the data transfer. Each kernel runs many communication primitives, with the unit of data transmitted in a single operation called a *chunk*. The size of a chunk is typically a small fraction (usually about 1%) of the total data transferred during synchronization. Thus, the backend divides the data to be synchronized into multiple *micro-batches* for sequential transfer. The size of each micro-batch is the total size of all chunks scheduled by a single execution of the communication algorithm. Current communication algorithms does not take the execution of micro-batch transfers into account, leaving this responsibility to the backend. There are two execution granularities for micro-batches in existing backends:

(1) Algorithm-level execution: At the algorithm level, the execution granularity adopts a lazy execution scheduling approach, where only chunk transfers within a single micro-batch are scheduled, without considering execution across multiple micro-batches. This results in low parallelism. Synthesizers execute at this granularity, which is the primary reason they fail to achieve optimal performance.

(2) Stage-level execution: The current backend only provides a manual stage division method for this granularity. It allocates separate TBs and buffer resources for each stage, with parallel execution between stages, completing the entire micro-batch transfer after all stages are executed. Although this execution approach improves parallelism, the heavy scheduling and dependency analysis, combined with additional resource demands, make it both time-consuming and resource-intensive. Expert-customized methods like MSCCLang adopt this approach, which is the root cause of their resource bottlenecks.

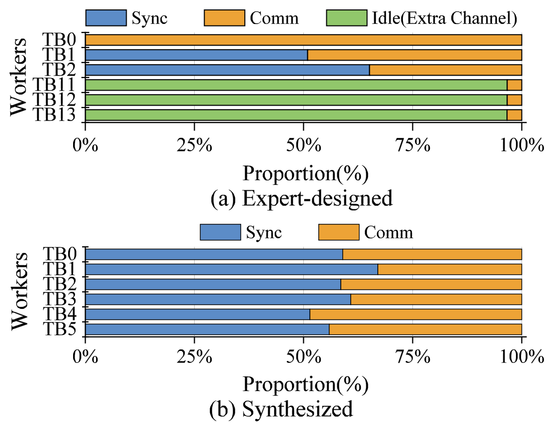


Figure 2: Time cost breakdown of primitives in custom and synthesized single-node AllReduce algorithm on existing CCL runtime.

2.2 Motivation

We conducted multiple experiments to analyze collective communication libraries supporting expert-designed and synthesized algorithms (refer to §5.1 for more information about experiment setup), such as MSCCL [7], and identified several bottlenecks:

Inefficient execution granularity of micro-batches leads to severe bottlenecks: low link utilization and significant resource waste. When the backend executes collective algorithms synthesized by the synthesizer, it lacks a global (cross-micro-batch) scheduling strategy and instead schedules data transfers sequentially within the algorithm. This approach leads to significant pipeline bubble overhead, which is understood as idle link time during algorithm execution. The accumulation of bubbles results in low link utilization during the communication task, severely limiting effective communication bandwidth. As shown in Table 1, on a 16-GPU topology spanning two servers, each equipped with 8 A100 GPUs and interconnected via 200 Gbps RoCE, the AllReduce algorithm synthesized by TACCL utilizes only 31.8% of the active link bandwidth for communication.

The backend also faces challenges when executing expert algorithms, particularly with complex cross-micro-batch dependency analysis. It cannot automatically schedule the execution pipeline, and instead requires manual specification of additional communication channels (which consumes more TB resources). The extra TBs are constrained by the execution order's dependency constraints, leading to periods of inactivity and high resource contention. As shown in Figure 2(a), the TBs running on additional channels remain idle 98.2% of the time, resulting in severe resource wastage.

Inflexible TB allocation. Existing backends rely on connection-based TB resource management, where each connection is assigned a dedicated TB. However, due to varying execution times across different links, this rigid allocation method results in TB synchronization blocking, causing significant and unnecessary resource constraints, particularly in imbalanced topologies. As shown in Figure 2(b), the time spent by TBs under synchronization blocking can

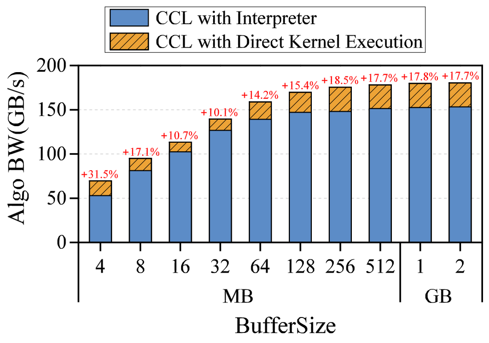


Figure 3: Performance Comparison: Runtime Interpreter vs. Direct Kernel Execution.

reach as high as 67.1%, severely impacting both resource utilization and algorithm efficiency.

Runtime performance degradation. Libraries supporting expert-designed and synthesized algorithms rely on runtime interpreters to continually load and process input algorithms, causing performance degradation. Figure 3 reveals that this process results in an average performance loss of 17.1%.

2.3 Summary

Based on the experiments and analysis above, we conclude that an efficient CCL backend requires optimizations in execution granularity, flexible TB allocation strategies, and lightweight, low-overhead communication kernel generation. In the following sections, we introduce ResCCL as a solution to these challenges. Specifically, §3 analyzes the optimal objectives for execution plan scheduling and presents the key insights we used to address this issue. §4 systematically discusses the design details of ResCCL as a backend. First, ResCCL offers a language (§4.2), *ResCCLang*, for algorithm developers to write flexible and high-performance algorithms. Second, ResCCL introduces an innovative execution granularity (§4.3) to minimize pipeline bubbles during algorithm execution and improve global link utilization. Next, ResCCL presents an efficient TB allocation strategy (§4.4) to significantly reduce idle TB usage and improve resource utilization. Finally, ResCCL transforms the optimized pipeline into lightweight communication kernels to minimize backend runtime overhead (§4.5).

3 PROBLEM AND GOAL

Any communication algorithm can be abstracted as a set of transmission tasks, T, under a specific cluster topology. A transmission task $t(e, d) \in T$ is defined as a chunk transfer between GPU peers, where e represents the communication link used by the task, and d denotes the dependencies between tasks.

We define the dependencies between tasks as two types: **(1) Data dependency:** When two transmission tasks access the same buffer offset at the same time, we define them as having a data dependency. This dependency expresses the sequence in which transmission tasks must be executed according to the algorithm logic to ensure the correctness of the results. **(2) Communication dependency:** This type of dependency refers to link conflicts between tasks that overlap in their time slots. If two tasks use the same link at overlapping times, they are considered to have a communication dependency. Our key insight is that using multiple threads to perform transmission tasks on the same link introduces unnecessary resource contention between tasks. Therefore, introducing communication dependencies into the execution plan is essential to optimize link bandwidth utilization by constraining the execution order.

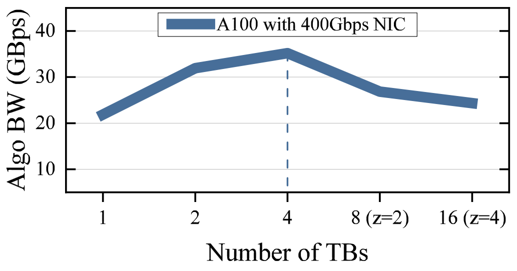


Figure 4: Impact of TB parallelism on communication bandwidth.

We conducted experiments to observe the link contention between TBs caused by communication dependencies, as shown in Figure 4, which investigates the impact of communication dependencies by performing P2P transfers over a single NIC to emulate an ALLGATHER involving only two GPUs, while varying the number of TBs. When the TB count is ≤ 4, the algorithm's bandwidth steadily increases; beyond four TBs, additional TBs actually reduce bandwidth. We attribute this to the balance between aggregate thread-level communication capability and link bandwidth. With four TBs, the combined throughput of all threads just matches the link's capacity. However, when the number increases to 8, the total thread-level communication capacity becomes twice that of the link bandwidth—*i.e.,* each transmission task contends with another task running in parallel for the same link resource.

This experiment demonstrates that when the transmission tasks performed by the TB reach the peak link bandwidth, adding more parallel TBs results in a degradation of communication performance. This phenomenon validates the authenticity of the communication dependencies we have defined. Therefore, optimizing the scheduling of the pipeline should aim to avoid communication dependencies to maximize bandwidth utilization.

The execution time for transmission tasks with communication dependencies is approximated using the following formula:

$$T_{conflict} = n \cdot z \cdot (\alpha + c \cdot \beta) + \mathcal{L}(z) \cdot \gamma \tag{1}$$

Here, n represents the number of micro-batches, z represents the factor by which thread-level transmission capability exceeds the bandwidth of a single link, α denotes the startup overhead of a transmission task, c refers to the data size of a chunk, β is the inverse of the link bandwidth, $\mathcal{L}$ is the penalty term for the performance loss caused by additional TB contention, and γ is a constant factor representing the link contention overhead.

As discussed above, transmission tasks are subject to two types of dependencies: data dependency and communication dependency.

Violating data dependency compromises the correctness of the algorithm, while violating communication dependency leads to link contention, reducing link utilization. We model the optimal scheduling problem for transmission tasks as follows: Given a directed acyclic graph $G_A = (V_T, E)$ that captures algorithm A's data dependencies—where each vertex $v_t \in V_T$ represents a transmission task, each edge $v_{t_i} \to v_{t_j} \in E$ denotes that v_{t_i} consumes the data produced by v_{t_j}. The execution time of each transmission task, accounting for both link latency and bandwidth, is explicitly defined. Task execution must satisfy two rules:

(1) When invoking task v_t for a given micro-batch, all tasks that v_t depends on must have already finished their invocation for that same micro-batch.

(2) When tasks with communication dependencies are scheduled, their execution time is prolonged as described by Equation 1, due to the additional overhead introduced by link contention.

The algorithm's total completion time is defined as the moment when all tasks have completed their invocations across every micro-batch after scheduling. Hence, the goal is to devise a transmission task schedule that minimizes this completion time.

The optimal objective problem described above is equivalent to a complex graph coloring problem [24], which has been proven to be NP-hard and cannot be solved optimally in polynomial time. Furthermore, no known universal solving strategy exists for the original scheduling problem presented in this paper. Therefore, we have summarized constraints in our detailed investigation to reduce the problem's complexity. Our key insight is that, between micro-batches, only communication dependencies exist between transmission tasks, with no data dependencies. As a result, the same combination of transmission tasks can exist across different micro-batches, and in most cases, the optimal solution is constructed by a cycle of transmission task combinations occurring in sequence over time. Based on this insight, we propose a novel execution granularity cross micro-batches: *task-level execution.* The solution under this execution granularity satisfies the following constraint:

> The same TB iteratively executes the same transmission task in sequence until the task is completed across all micro-batches.

Although the problem remains NP-hard after introducing this constraint, we have derived a guiding solution strategy (detailed in §4.3) that can achieve the optimal solution in most cases.

Furthermore, we believe that introducing the above constraints to the original scheduling problem offers the following advantages:

(1) Scalability. The constraint simplifies the problem, requiring only one scheduling step for each transmission task. Without this constraint, all tasks for each micro-batch would be scheduled separately.[1] Specifically, G_A would need to be expanded by a factor of the number of micro-batches, and additional considerations for the source micro-batch would increase the problem's complexity to the power of the number of micro-batches. Therefore, as the number of micro-batches increases, the original problem is difficult to scale. However, after introducing the constraint, the solution space becomes linearly scalable.

(2) Execution overhead. The scheduling is executed by the GPU's TB. Without constraints, TB must execute each transmission task individually. With the constraints, however, TB only needs to sequentially execute cycles of task combinations, reducing the execution complexity to $\frac{1}{n}$ (where n is the number of micro-batches).

(3) Solution quality. In our scheduling formulation, the total execution time of an algorithm equals the completion time of its last task. Formally,

$$T_{\text{algo}} = \max_{e_i \in E}\{T_i\} \tag{2}$$

where T_{algo} denotes the algorithm's overall runtime and T_i is the execution time of edge e_i (hereafter referred to as the *link execution time*). Consequently, the total runtime is determined by the bottleneck link. Building on this observation, we next detail how different execution strategies impose distinct constraints on the scheduling problem and analyze the resulting solution quality for each strategy.

Algorithm-level execution. Algorithm-level execution repeats the same task sequence for every micro-batch, yielding a fixed schedule. The link's execution time is given by:

$$T_A = n \sum_{j=1}^{m} \left(\alpha + c\,\beta + B_j\right) \tag{3}$$

where n is the number of micro-batches, m is the number of tasks in each micro-batch, and B_j denotes the bubble time incurred by task j due to data dependency stalls. Because the algorithm is executed identically across all iterations—and both expert-designed and synthesizer-generated algorithms are structured to avoid link contention—the total link execution time is modeled as the per-micro-batch cost (in the absence of contention) multiplied by the number of micro-batches.

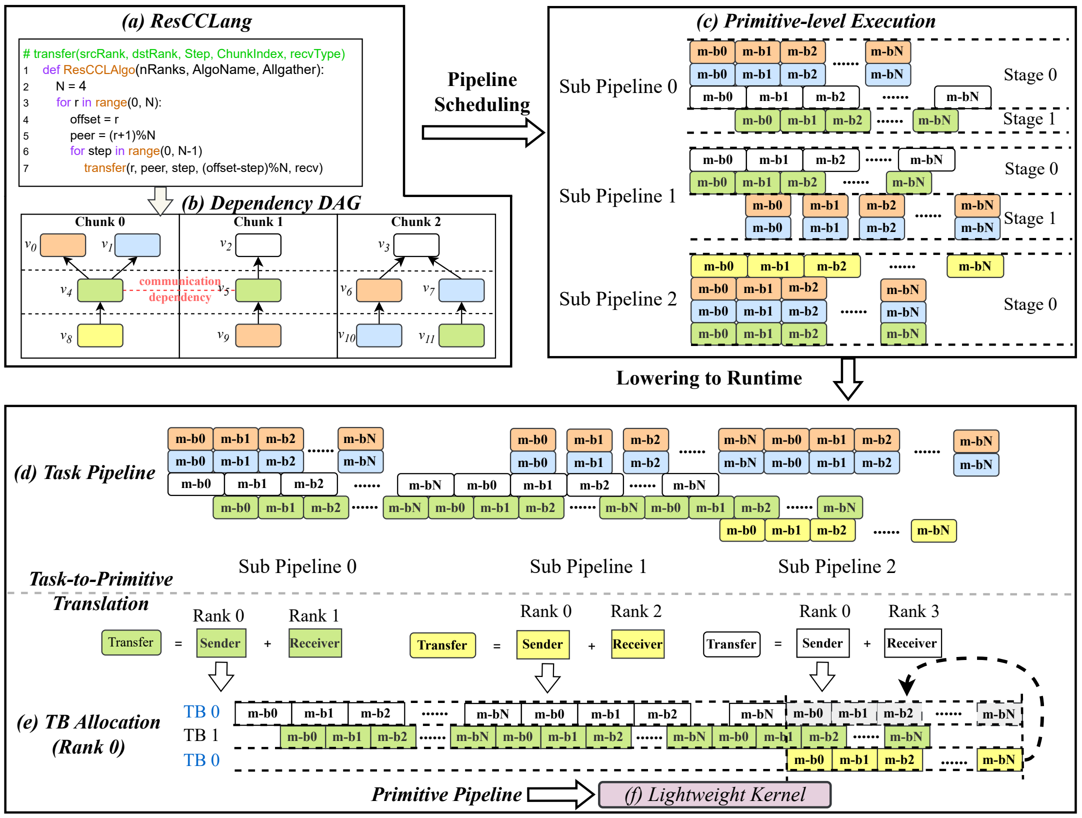
Stage-level execution. Stage-level execution requires partitioning the algorithm into multiple stages, with algorithm-level execution applied within each stage. The stages are only required to satisfy data dependencies between them. Under this constraint, different stage partitions lead to different schedules, but once the partitioning is fixed, the schedule is fully determined. For a partition into K stages, the link execution time is given by the following formula:

$$T_S = \max_{1 \le k \le K} \left\{ n \sum_{j=1}^{m_k} \left[z_k(\alpha + c\beta) + \gamma \mathcal{L}(z_k) + B_j \right] \right\} \tag{4}$$

where m_k represents the number of tasks per micro-batch in stage k, and z_k denotes the instance of z corresponding to the case where the algorithm is divided into k stages. Since stage-level execution splits the workload of a link into multiple stages executed in parallel to reduce bubbles, it results in fewer bubbles but may introduce communication contention.

Task-level execution (Ours). Under task-level execution, once a task is scheduled, it must execute its invocations across all micro-batches. This constraint confers two notable advantages: (1) Because every task processes all of its invocations, tasks that share data dependencies are pipelined: while one invocation stalls on a dependency, another invocation from a different micro-batch can proceed, thereby masking bubbles caused by data stalls. (2) Tasks

[1] Task constraints allow a single scheduling to be extended across all micro-batches globally.



```
# transfer(srcRank, dstRank, Step, ChunkIndex, recvType)
def ResCCLAlgo(nRanks, AlgoName, Allgather):
    N = 4
    for r in range(0, N):
        offset = r
        peer = (r+1)%N
        for step in range(0, N-1)
            transfer(r, peer, step, (offset-step)%N, recv)
```

Figure 5: Workflow of ResCCL.

that would otherwise contend for the same communication link is assigned to separate pipelines, eliminating link-level conflicts.

Solutions produced under this model are therefore not predetermined; they honor all communication dependencies, and the extent to which bubbles are eliminated depends on how the pipelines are constructed. The resulting link execution time is

$$T_P \;=\; t_{\text{Load}} \;+\; n\,m\,(\alpha + c\beta) \;+\; n\sum_{j=1}^{m'} B_j, \quad m' \le m \tag{5}$$

where t_{Load} is the one-time pipeline fill cost, independent of the number of micro-batches; n is the micro-batch count; m' is the number of residual bubbles after scheduling. Because both the size and count of bubbles diminish under task-level execution, the final bubble term is no greater—and typically much smaller—than that of algorithm-level execution.

Comparison. By contrasting the solution costs under the three execution strategies, we obtain

$$\lim_{n\to\infty}\left(T_A : T_S : T_P\right) = \sum_{j=1}^{m} B_j : \sum_{j=1}^{m_k}\left[\gamma\,\mathcal{L}(z_k) + B_j\right] : \sum_{j=1}^{m'} B_j \tag{6}$$

Hence, when the number of micro-batches n is sufficiently large and the pipeline construction substantially reduces bubbles, the task-level schedule yields a strictly better solution than the algorithm- or stage-level strategies.

4 DESIGN

4.1 Backend Optimization Workflow

Figure 5 illustrates the execution optimization workflow of ResCCL for a given collective communication algorithm. First, ResCCL performs a global dependency analysis on the input algorithm, which is described by our DSL, *ResCCLang* (introduced in §4.2 and illustrated in Figure 5(a)), generating a dependency DAG (Figure 5(b)), where each node represents a transmission task, and the directed edges between nodes indicate data dependencies. Since the algorithm does not have cyclic dependencies (which would lead to deadlocks), it forms a DAG. Different chunks are stored in isolated addresses, so there are no data dependencies between transmission tasks for different chunks, while nodes of the same color have communication dependencies. Guided by the dependency DAG, ResCCL performs primitive-level scheduling across micro-batches, as detailed in §4.3. Sub-pipelines for executing tasks are hierarchically assembled (Figure 5(c)), ensuring both data and communication dependencies are respected during the assembly process. Once all transmission task nodes in the dependency DAG are scheduled into the execution pipeline, ResCCL lowers the transmission task pipeline to the runtime and combines all sub-pipelines into a complete execution pipeline (Figure 5(d)). This primitive-level execution pipeline scheduling optimization maximizes bandwidth utilization for collective communication algorithms, allowing their performance to approach the theoretical peak. Subsequently, at runtime, ResCCL translates[2] the transmission tasks into communication primitives (such as `send`, `recv`, `recvReduceCopy`) and proceeds with thread block (TB) allocation optimization (as detailed in §4.4). ResCCL abandons the traditional, rigid connection-based allocation in favor of a more flexible, state-based TB allocation mechanism (Figure 5(e)). ResCCL combines multiple non-overlapping communication primitives based on their temporal sequence and assigns them to the same TB for execution. This flexible allocation mechanism significantly reduces the usage of SM resources. Finally, §4.5 presents ResCCL's lightweight kernel paradigm, which generates streamlined executable kernels for the optimized primitive pipeline, thereby minimizing runtime overhead.

4.2 ResCCLang

Design choices. ResCCL takes algorithm logic as input and automatically optimize the runtime execution and generalize it across diverse deployment scenarios. The input is defined as the data movement between buffers at each step. In this process, algorithm designers and synthesizers only provide the algorithmic logic, without the need to invest additional effort in runtime tuning or in adapting the algorithm to different hardware or execution environments. To achieve this goal, introducing a new DSL for ResCCL is essential, rather than extending existing languages. Specifically, (1) existing collective communication backends lack automated optimization and transmission scheduling capabilities, resulting in complex DSLs (*e.g.,* MSCCLang) that require algorithm designers to manually apply various redundant interfaces for performance tuning—such as explicitly assigning transmissions to separate channels to improve parallelism. In contrast, ResCCL allow algorithm designers to specify only the core algorithm logic, without needing to manage TB allocation, channels, or other low-level details. All optimization tasks are handled internally by ResCCL, enabling a much simpler and more streamlined language interface. (2) Furthermore, we observed that algorithms is executed either in-place or out-of-place during training. Existing languages require algorithm designers to account for both scenarios and write a redundant DSL for each case.

To address these issues, ResCCL introduces *ResCCLang*, a DSL that provides algorithm designers with a flexible and efficient tool for algorithm development. ResCCLang defines a unified abstraction for algorithm logic, enabling both human experts and automated synthesis tools to express a wide range of collective algorithms in a concise and analyzable form. The abstraction is composed of the following key elements: **(1) DataBuffer.** ResCCLang models the input and output memory regions for each rank as a unified `DataBuffer`. Each buffer is partitioned into transmission units called *chunks*, which are indexed by a `ChunkId`. The number of chunks per rank is equal to the total number of ranks, ensuring that each ⟨`Rank`, `ChunkId`⟩ pair maps uniquely to a specific chunk in the system's global memory space. **(2) Step.** Temporal order is represented using a discrete `Step` index. All algorithmic actions are strictly ordered by their step values—actions assigned to smaller steps must occur before those assigned to larger ones—enforcing a total ordering over the sequence of operations. **(3) Transfer.** ResCCLang uses the function `Transfer` to describe communication behaviors. A transfer is defined as: *Transfer(srcRank, dstRank, step, chunkId, opType).* This tuple uniquely identifies a transmission task, specifying the source and destination ranks, the logical execution step, the target data chunk, and the operation type (*e.g.,* send, recv). To further discuss its functionality, we have provided an example diagram of ResCCLang and a description of the DSL syntax in Appendix B.

[2]The *task* abstraction is employed for global analysis, whereas a *primitive* denotes the unit actually executed at runtime, we will alternate between the two terms in the subsequent sections.

Algorithm 1 Hierarchical Priority-based Dynamic Scheduling (HPDS)

```
1:  Input: $G = (V, E)$
2:  Output: $Pr$
3:  $Q \leftarrow$ PriorityQueue()
4:  $Pr \leftarrow$ PipelineResult()
5:  while $G \neq \emptyset$ do
6:      $Pc \leftarrow$ CurrentPipeline()
7:      $F \leftarrow$ ChunkFlag()
8:      while $\neg$ all $F$ = False do
9:          $C \leftarrow$ Q.GetHighestWithFlag($F$)
10:         $NodeList \leftarrow \emptyset$
11:         for each $Node \in G[C]$ & without data dependency do
12:             if $Node$ satisfies all comm dependencies then
13:                 $NodeList$.append($Node$)
14:             end if
15:         end for
16:         if $NodeList = \emptyset$ then
17:             $F$.setFalse($C$)
18:         else
19:             $Pc$.insert($NodeList$)
20:             $C$.priority $\leftarrow$ $C$.priority - 1
21:             Q.sort()
22:             $G$.remove($NodeList$)
23:         end if
24:     end while
25:     $Pr$.append($Pc$)
26: end while
```

4.3 Primitive-Level Execution Scheduling

As discussed in §3, the scheduling problem remains NP-hard even under primitive(task)-level execution, making direct optimization intractable. To address this issue, we first analyzed existing expert-designed and synthesized algorithms and obtained the following insight: For the same `ChunkId`, primitives with later `Step` values depend on those with earlier `Step` values, introducing data dependencies. However, primitives across different `ChunkIds` are independent and free from such constraints. Consequently, algorithms tend to distribute primitives with different `ChunkIds` across separate links within the same `Step` to avoid communication conflicts. This observation suggests that after scheduling the primitives of a given `ChunkId`, it is often more beneficial to switch to scheduling primitives of other `ChunkIds` rather than continuing with the current one.

Building on this insight, we propose the **hierarchical priority-based dynamic scheduling (HPDS)** strategy, defined by the iterative process outlined in Algorithm 1, which presents the pseudocode of our proposed hierarchical priority-based dynamic scheduling (HPDS) algorithm. Given a dependency DAG G as input, the algorithm outputs a global task pipeline P_r, which is constructed by concatenating multiple sub-pipelines P_c. Each sub-pipeline consists of tasks that simultaneously satisfy both data and communication dependencies. Since this constraint may not hold globally, multiple sub-pipelines must be constructed to cover the entire DAG (lines 5–26).

Sub-pipeline Construction. The construction of each sub-pipeline P_c proceeds as follows: (i) Candidate selection. From the priority queue Q, select the Chunk-DAG $G[C]$ with the highest priority whose flag F_C is set to `true` (line 9). (ii) Task extraction. Traverse all tasks in $G[C]$ and collect those satisfying both data and communication dependencies into a list `NodeList` (lines 11–15). (iii) Scheduling decision. If `NodeList` is empty, it indicates that no eligible task remains in $G[C]$ for this sub-pipeline; the flag F_C is updated to `false` (lines 16–17). Otherwise, the tasks in `NodeList` are appended to the current sub-pipeline P_c, the DAG G is updated accordingly, and the priority queue Q is adjusted to reflect the changed priority of $G[C]$ (lines 18–24). (iv) The inner loop terminates once all F_C are `false`, signifying no further tasks can be added to the current P_c (Line 8).

Global pipeline assembly. Once a sub-pipeline P_c is completed, it is appended to the global pipeline P_r (Line 25). The process is repeated until all tasks in G have been scheduled, at which point the final pipeline P_r is returned (lines 5–26).

The algorithm works by dynamically assigning tasks to *sub-pipelines* $P_{c1}, P_{c2}, \ldots, P_{ck}$ based on their priorities and dependency relationships, such that each sub-pipeline P_{ci} is constructed iteratively. Each sub-pipeline represents a modular unit of execution, contributing to global coordination. The sub-pipelines are combined progressively, forming a complete execution pipeline.

The priority assignment mechanism plays a central role in the HPDS strategy. Let $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ represent the set of priorities, with each p_i corresponding to a task $t_i \in T$. Tasks with lower execution frequency (*i.e.,* underutilized chunks) are assigned higher priority, ensuring *dynamic load balancing*. This priority mapping is dynamically updated based on the progress of each sub-pipeline. The scheduling proceeds by selecting tasks with no data dependencies from the dependency DAG, ensuring that tasks are scheduled following communication dependency constraint: For each task t_i, it must not be scheduled to share the same communication link as any task t_j from the current sub-pipeline, *i.e.,* $\forall t_i, t_j \in P_k$, if $\text{comm}(t_i, t_j) \neq \emptyset \Rightarrow l_i \neq l_j$, where $\text{comm}(t_i, t_j)$ represents the communication dependency between tasks.

This process continues iteratively until all tasks have been successfully scheduled, satisfying the task dependencies and communication constraints.

Minimizing pipeline bubbles. The effectiveness of a scheduling algorithm is measured by the extent to which it eliminates pipeline bubbles; below, we analyze how efficiently HPDS reduces such bubbles. Let $b(t_i, t_j)$ denote the *pipeline bubbles*, which are defined as an idle time interval caused by an imbalance in task execution times across different links. Task execution times vary significantly based on link latency and bandwidth. Specifically, let λ_{intra} and λ_{inter} represent the intra-machine and inter-machine latency, respectively. Our experiments show that $\lambda_{inter} \geq 2.5 \times \lambda_{intra}$, even under identical bandwidth conditions. Therefore, if an intra-machine task t_j is scheduled in the same sub-pipeline as its dependent inter-machine task t_i, the latency mismatch between them delays t_j, resulting in a bubble $b(t_i, t_j)$.

To minimize these bubbles, HPDS assigns tasks with lower execution frequencies to higher priority positions, ensuring that inter-machine and intra-machine tasks dependent on them are not scheduled in the same sub-pipeline. This approach reduces the bubble $b(t_i, t_j)$ and improves overall pipeline efficiency.

Task-to-primitive translation. The transmission tasks T are mapped to a set of primitives $R = \{r_1, r_2, \ldots, r_m\}$, where each task t_i is represented by a pair of send ($r_{\text{send}}(t_i)$) and receive ($r_{\text{recv}}(t_i)$) primitives. This mapping is one-to-one, with each task t_i corresponding to a single transfer of a data chunk. After all tasks are mapped to primitives, a global set of primitives R is generated, which will be used in subsequent steps of the scheduling process.

4.4 Flexible TB Allocation

In traditional methods, the *connection-based allocation* approach is used, where each transmission link (connection) between GPUs is independently assigned a TB. This results in the number of TBs being equal to the number of connections, *i.e.,* for each connection $c_i \in C$, a corresponding $TB(c_i)$ is assigned, where C is the set of all GPU connections. This leads to a potentially inefficient allocation because many of these connections share the same network interface card (NIC), causing congestion due to their communication dependencies.

The connection-based allocation would result in only one TB being active at a time for each link, thus leading to inefficiency. To address this inefficiency, we introduce the **state-based allocation** strategy, which works as follows:

Timeline analysis. First, the entire pipeline $\mathcal{P}$ is analyzed over time, and the active times of each connection are identified. Let $\text{active}_l(t)$ represent the time interval during which link l is active at time t. The aim is to identify and merge connections that will never be active simultaneously.

Merging connections. Connections l_i and l_j are merged if they are not active simultaneously, *i.e.,* if $\text{active}_{l_i}(t) \cap \text{active}_{l_j}(t) = \emptyset$. The merged connections are then assigned a single $TB(l_i, l_j)$, reducing the number of TBs without affecting overall execution time. This method relies on serial activity of connections: all inter-connections that do not overlap in active time are merged.

Thus, the total number of TBs is reduced from $|C|$ (the number of connections) to a smaller number $|TB|$, where:

$$|TB| = \left| \bigcup_{(l_i, l_j) \in \text{Merged Connections}} TB(l_i, l_j) \right| \quad (7)$$

Network contention. Collective communication operates at a higher abstraction level, making it orthogonal to network-layer issues such as ECMP hash collisions. Thus, explicitly considering network-level path selection and underlying congestion is beyond the current scope of ResCCL. Future enhancements may leverage programmable data planes, employing telemetry to obtain in-network state information through platforms such as Crux [3] or vFabric [38]. Nevertheless, ResCCL's current capabilities effectively alleviate intra-job network congestion. ResCCL allocates thread blocks with full awareness of flow dependencies, thereby limiting the number of simultaneous connections on each link and eliminating link-level conflicts in the resulting schedule. Because such conflicts manifest as transmission slowdowns—fundamentally a form of link congestion—ResCCL's state-based allocation inherently mitigates congestion. Consequently, the system experiences significantly less performance degradation under network contention.

4.5 Lightweight Kernel Generation

To improve runtime system efficiency, ResCCL adopts directly generated kernels as its execution engine, significantly reducing the overhead associated with control logic inside the kernel. *As the first collective communication backend to implement primitive-level execution*, ResCCL runtime needs to execute invocations of each primitive across all micro-batches in one pass, whereas existing runtimes only support primitive invocation within a single micro-batch. Therefore, a dedicated runtime system is required to support this execution model.

To address this, we establish a *general paradigm for kernel generation* that applies to any collective communication algorithm. This paradigm is defined across three dimensions: **(1) Rank dimension:** Specifies the complete set of primitives that each GPU (rank) must execute during runtime. **(2) TB dimension:** Further refines the rank dimension by specifying the primitives assigned to each thread block. **(3) Pipeline dimension:** Provides a finer partitioning of the TB dimension by grouping primitives according to pipeline indices. Each specific pipeline dimension defines the primitive to be executed by the current TB and cycles through all corresponding micro-batch invocations during execution. This paradigm provides an effective model for lowering primitive pipelines into executable kernels. ResCCL materializes the resulting pipeline into concrete, lightweight kernels, enabling hardware-level execution of the optimized schedule. Compared with conventional online interpreters, these kernels eliminate substantial control overhead and deliver significantly higher runtime efficiency.

5 EVALUATION

In this section, we conduct an extensive experimental evaluation of ResCCL. We benchmark the two most widely used collective communication operators: AllReduce and AllGather, and compare their performance and resource savings across various network topologies and algorithm configurations. We also quantify the overhead introduced by ResCCL's proposed techniques and report the real-world performance gains observed in end-to-end training scenarios.

5.1 Experiment Setup

Implementation. ResCCL encompasses both offline scheduling of communication algorithms and their runtime execution, and is therefore realized in a three-layer architecture comprising roughly 6K+ lines of code. **(1) Offline compiler (top layer).** Modularly embedded in Python, the compiler accepts an algorithm logic, performs primitive-level scheduling, and generates lightweight communication kernels. **(2) Control plane (middle layer).** Implemented in C++, this layer serves as the control plane of the communication library, supplying CPU-side coordination and management for the runtime system. **(3) Runtime system (bottom layer).** Developed in CUDA, the runtime extends NCCL primitives, allowing flexible selection of communication links and providing backend support for executing the lightweight kernels.

Table 2: Summary of experimental setup: cluster, network, CCL, and training parameters.

Cluster Config	**GPU**	**Intra-Fabric**	**Inter-Fabric**	**Scale**	**Topo**
	A100	NVSwitch	RoCE	32 GPUs	Clos
CCL Config	**Instance**	**Algorithm**	**ChunkSize**	**Protocol**	**nWarps**
	Default/4	Ring/Custom	1MB	Simple	16
Training Config	**Model**	**Size**	**BS**	**TP**	**DP**
	GPT-3	6.7–44B	32	8	2–4
	T5	220M–3B	16	1	16

Testbed. We set up an A100 server cluster consisting of four nodes, each equipped with 8 NVIDIA A100 80GB GPUs, 32 GPUs in total. Each GPU provides 300 GBps of communication bandwidth and is connected via 6 NVSwitches operating at 600 GBps. In addition, each server is equipped with four NICs, each offering 200 Gbps of network bandwidth. The cluster employs a two-tier Clos network topology. Each server connects to a Top-of-Rack (ToR) switch via four links (one per NIC), with every two GPUs on the host sharing the same NIC. Communication between servers located in different racks (*i.e.,* not attached to the same ToR) is forwarded through second-tier aggregation switches.

Comparison baselines. We compared ResCCL against three baselines: NCCL [13], MSCCL [7], and MSCCLang/TACCL/TECCL [15, 29, 33] with MSCCL. NCCL (v2.25.1) represents the most widely used vendor-standardized library, while MSCCL (v1.0.2) is a unified collective communication library from Microsoft that supports expert-designed and synthesized algorithms. For expert-designed algorithms, we adopt the hierarchical algorithm, whose detailed design is provided in Appendix A. For synthesized algorithms, we used the TACCL and TECCL synthesizer to generate communication schedules tailored to our experimental setup and then executed TACCL/TECCL with MSCCL and ResCCL as backends, respectively, for comparison.

Configurations. Table 2 details the principal experimental settings. To guarantee a fair and consistent comparison, all communication backends—ResCCL, MSCCL, and NCCL—run under identical conditions: the same cluster hardware, network topology, backend parameters, and model configurations. Any parameters not explicitly listed in the table also follow identical, standard default values.

Collective-communication libraries generally expose three transport protocols—`Simple`, `LL`, and `LL128`—each offering a different latency-bandwidth trade-off. `Simple` delivers the highest sustained bandwidth, `LL` minimizes latency, and `LL128` preserves `LL`'s low latency while partially recovering bandwidth. Because our evaluation focuses on training throughput, we configure all three backends

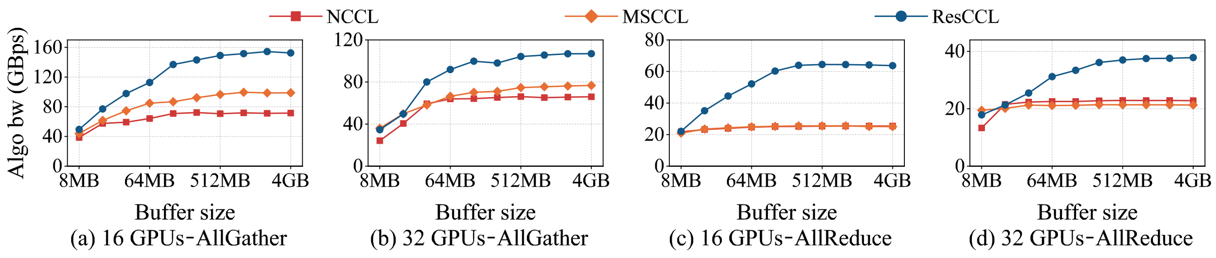


Figure 6: Communication performance of Expert-Designed AllReduce and AllGather across buffer sizes.

under test to use the Simple protocol. To evaluate the effectiveness of ResCCL in optimizing expert-designed algorithm execution, we developed several high-performance algorithms tailored to our experimental topology with detailed designs available in the Appendix A.

5.2 Communication Benchmark

Performance on expert-designed algorithms. Figure 6(a)-(b) presents AllGather algorithm bandwidth[3] achieved by ResCCL under two cluster scales and two algorithms optimized for specific topologies, compared against the same algorithms executed with MSCCL and the baseline NCCL. The transfer-chunk size is fixed at 1MB for all three backends. In the 16-GPU (two servers) setting, ResCCL delivers the largest speed-up, outperforming NCCL by 28.1%–2.2× and MSCCL by 12.4%–1.6× as the buffer size increases from 8MB to 4GB. In larger scales (Figure 6(b)), ResCCL consistently yields substantial algorithm bandwidth gains across all configurations. For buffer sizes above 32MB, it improves over NCCL by at least 38.2%, reaching up to 1.6× at 4GB, and achieves up to a 1.4× speed-up relative to MSCCL.

The AllReduce operator is implemented by combining AllGather with its reverse operation, conceptually similar to ReduceScatter and exhibiting an analogous traffic pattern. Using the same server configurations as in the AllGather experiments, Figure 6(c)-(d) shows that ResCCL improves algorithm bandwidth by 6.7%–2.5× over NCCL and by 10.7%–2.5× over MSCCL. Only in the four-server, 32-GPU setting, ResCCL is slightly slower (at most 8.3%) than MSCCL when the buffer size is below 16MB. This is because small messages yield fewer micro-batches, reducing scheduling opportunities. However, ResCCL amortizes its fixed overhead over larger micro-batches, enabling scheduling efficiency and performance gains to scale with workload size and delivering effective communication speed-ups across a wide range of data sizes in real-world training workloads.

Performance on synthesized algorithms. We supplied TACCL and TECCL with inputs tailored to our experimental topology and used each synthesizer to generate AllGather and AllReduce algorithms for these topologies. Notably, the open-source release of TECCL does not natively support AllReduce, we extended TECCL using the general assembly technique introduced in the expert-algorithm evaluation and synthesized a TECCL-AllReduce variant. We then executed the synthesized algorithms on both MSCCL and

[3]Algorithm bandwidth and latency are equivalent metrics, as bandwidth is derived from total data divided by communication latency, so reporting one fully captures the other.

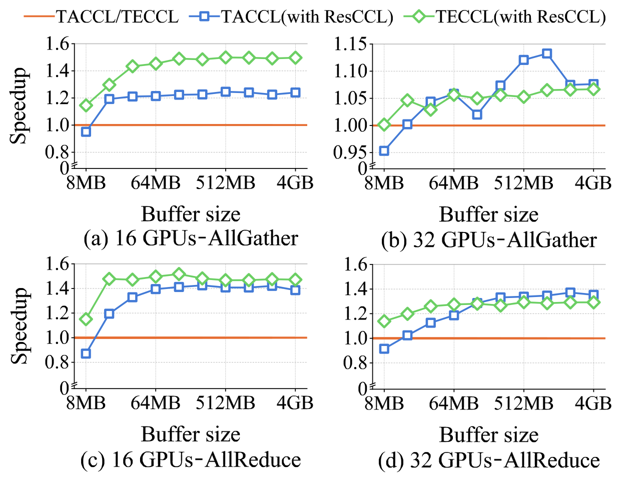


Figure 7: Communication performance of Synthesized AllReduce and AllGather across buffer sizes.

ResCCL backends and compared their algorithm bandwidth performance. Figure 7 presents the resulting speedups, with the y-axis denoting the relative acceleration and the orange horizontal line indicating the normalized baseline performance of MSCCL running the synthesized algorithms. The blue (square marker) and green (diamond marker) curves indicate the speedup achieved by ResCCL over MSCCL when executing the synthesized algorithms.

ResCCL consistently accelerates TECCL-synthesized algorithms over the entire buffer-size range, achieving speedups from 4.6% up to 1.5×. For TACCL, in the 2-server (16 GPUs) configuration, ResCCL consistently outperformed MSCCL for all larger buffers, achieving speedups of up to 1.4×, with only a slight performance drop (up to 8.5%) when the buffer size is below 8 MB, due to the same pipeline-fill and limited scheduling opportunities effects observed in the expert-algorithm experiments. In larger cluster scales, the trend is similar: once the buffer size exceeds 16 MB, ResCCL consistently yields substantial algorithm bandwidth improvements, with speedups ranging from 12.6% to 1.4×.

Communication benchmarks across additional topologies. To further validate the versatility and scalability of ResCCL, we conducted supplementary bandwidth benchmarks on two additional cluster configurations: (i) two servers with four A100 GPUs each, and (ii) four servers with four A100 GPUs each. As illustrated in

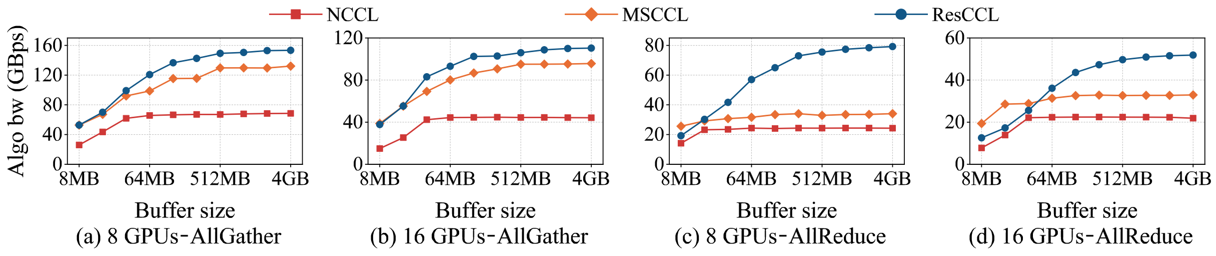


Figure 8: Communication performance of expert-designed AllReduce and AllGather under additional topologies.

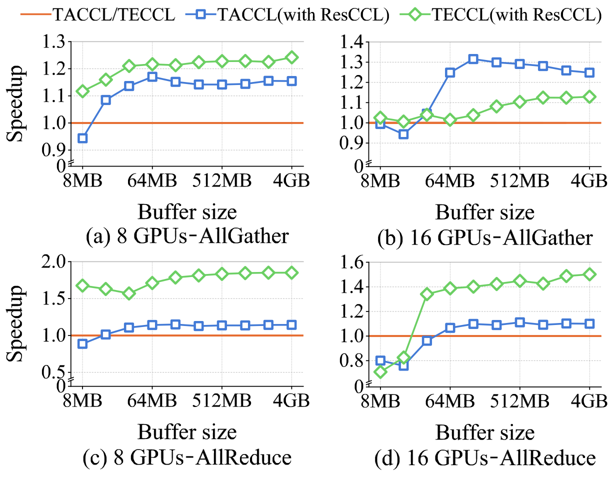


Figure 9: Communication performance of synthesized AllReduce and AllGather under additional topologies.

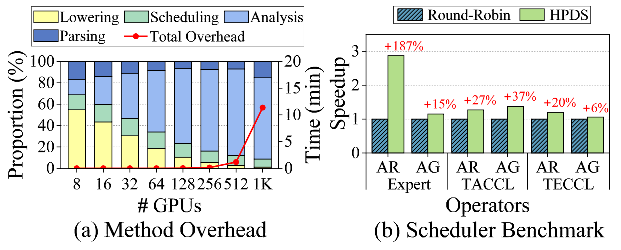


Figure 10: Runtime overhead & scheduling comparison.

Figures 8 and 9, ResCCL maintains clear performance superiority across both expert-designed and synthesized algorithms. For AllGather, relative to NCCL, ResCCL improves bandwidth by 1.6×–2.3× on expert algorithms. Compared with MSCCL, it delivers 6.8%–23.1% speed-ups for expert algorithms and 9.8%–31.1% for synthesized algorithms. For AllReduce, ResCCL achieves up to a 3.7× performance improvement over NCCL and up to a 2.4× speedup over MSCCL when using expert-designed algorithms. Under synthesized algorithms, ResCCL outperforms MSCCL by up to 50.1%.

Comparison for Custom Algorithm on V100 GPUs. We further evaluated the generality of ResCCL on a heterogeneous GPU cluster with V100 GPUs interconnected via 100G RoCE, using high-performance collective operations. In HM-AllGather (Figure 11 left), ResCCL achieved 1.6×-2.7× better performance for smaller inputs and 6.1%-18.2% for larger inputs compared to MSCCL, with a 2.1×-3.7× improvement over NCCL, especially benefiting from larger buffers. For HM-ReduceScatter (Figure 11 middle) experiment, ResCCL demonstrated up to 30.4% improvement over MSCCL for smaller inputs, with 4.9%-8.5% gains for larger buffers. Compared to NCCL, ResCCL achieved a performance increase of 1.9×–4.2×. Lastly, in the HM-AllReduce (Figure 11 right), ResCCL's pipelined execution resulted in 10.3%-68.2% better performance than MSCCL, and 2.3×-3.9× better than NCCL.

5.3 Workflow Breakdown

Phase-by-phase scalability metrics. In our micro-benchmark study of ResCCL, we isolated the major execution phases of the workflow and measured their scalability, specifically, the time each phase consumes as we scale the collective algorithm across increasingly large clusters. Figure 10(a) depicts these results as a per-phase breakdown. Because the phases in ResCCL execute serially—the output of one feeding directly into the next—we recorded the start and end times of each stage to expose their individual contributions, while the red curve in the figure shows the overall end-to-end latency.

The pipeline unfolds as follows. *Parsing* translates the DSL into an abstract-syntax tree and then extracts the primitive pipeline required by the communication backend. *Analysis* converts the AST into a dependency DAG. *Scheduling* applies the HPDS algorithm to that DAG, producing an optimally ordered task sequence. Finally, *Lowering* turns the task pipeline into the primitive pipeline that the runtime executes. Even at the largest scale we tested—1,024 GPUs[4] —ResCCL completes the entire DSL processing pipeline in roughly 11 minutes. Because this workflow is executed once, offline, before training begins, the cost is negligible over a multi-hour or multi-day training run.

HPDS vs. RR. To further validate the effectiveness of the HPDS scheduler, we implemented a baseline round-robin (RR) policy. RR is a classic scheduling approach that simply cycles through tasks in a fixed order. In our implementation, we traverse each chunk's DAG in ascending chunk-ID order, visit the chunks in a circular queue, and schedule them in that same immutable sequence. We compared HPDS and RR on an 8-GPU, two-server topology—the other configurations exhibit similar trends—and measured their impact on both expert-designed and synthesizer-generated algorithms.

[4]The GPU scale was emulated on the host since the workflow overhead is incurred entirely offline.

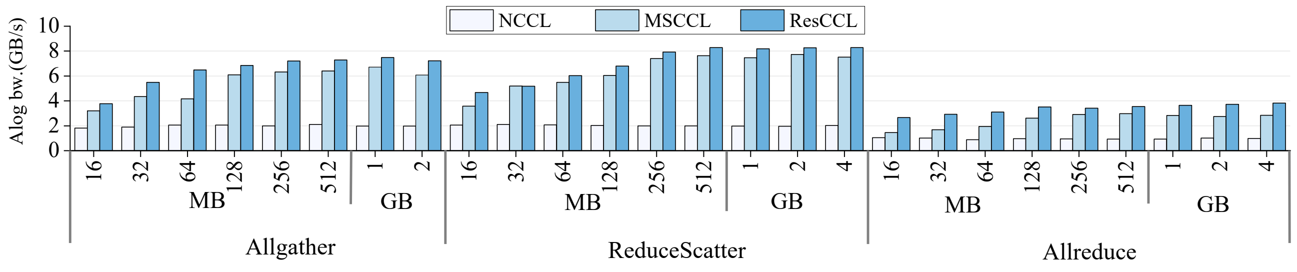


Figure 11: Performance comparison of ResCCL, NCCL, and MSCCL across multiple GPU servers for different communication operators.

Table 3: Comparison of TB resource utilization between ResCCL and MSCCL across different algorithms.

Backend	TB Utilization	Expert AllReduce				Expert AllGather				Synthesized AllReduce				Synthesized AllGather			
		Topo1	*Topo2*	*Topo3*	*Topo4*	*Topo1*	*Topo2*	*Topo3*	*Topo4*	*Topo1*	*Topo2*	*Topo3*	*Topo4*	*Topo1*	*Topo2*	*Topo3*	*Topo4*
MSCCL	# TB	14	30	14	30	14	30	14	30	10	18	9	8	10	18	5	5
	Comm Time	71.6%	66.2%	63.7%	63.7%	95.2%	82.3%	67.9%	66.1%	33.1%	23.9%	50.9%	34.9%	96.3%	97.8%	47.1%	60.4%
	Avg Idle	28.4%	33.8%	36.3%	36.3%	4.8%	17.7%	32.1%	33.9%	66.9%	76.1%	49.2%	65.1%	3.7%	2.2%	52.9%	39.6%
	Max Idle	72.38%	99.5%	99.0%	99.0%	19.2%	99.1%	99.5%	99.0%	90.2%	99.9%	99.9%	99.9%	36.6%	87.3%	90.2%	98.1%
ResCCL	# TB	**8**	16	8	16	**8**	16	8	16	**6**	6	6	6	**4**	4	4	4
	Comm Time	**95.7%**	95.8%	85.7%	85.0%	**98.3%**	98.2%	93.1%	92.3%	**67.6%**	58.4%	66.5%	61.1%	**99.4%**	99.2%	88.7%	83.8%
	Avg Idle	**4.3%**	4.2%	14.3%	15.0%	**1.7%**	1.8%	6.9%	7.7%	**32.4%**	41.6%	33.5%	38.9%	**0.6%**	0.8%	11.3%	16.2%
	Max Idle	**19.6%**	20.2%	22.5%	21.4%	**5.6%**	7.4%	21.4%	20.8%	**56.4%**	69.8%	60.8%	62.8%	**4.9%**	8.9%	28.3%	30.7%

As illustrated in Figure 10(b), HPDS consistently outperforms the Round-Robin baseline, delivering speedups of up to 187%.

5.4 SM Resource Utilization

Efficient SM utilization is crucial to distributed DL training, where contention between computation and communication for these shared resources often becomes the dominant bottleneck. We quantify this efficiency by examining three metrics: (i) the total number of thread blocks (TBs) allocated to communication tasks, (ii) the average TB idle ratio, and (iii) the maximum TB idle ratio. For each TB, the idle ratio is the fraction of its lifetime spent busy-waiting—either synchronizing with other TBs or waiting for a peer to become ready—while still occupying SM resources.

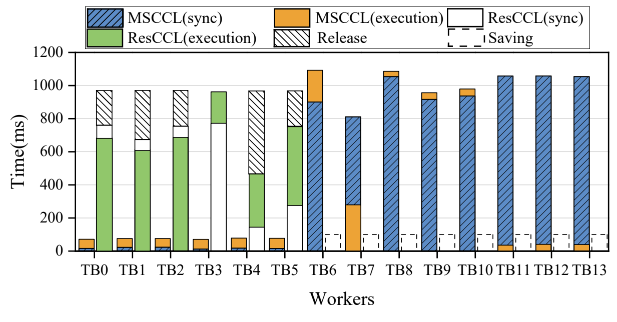
Table 3 summarizes these metrics for ResCCL and MSCCL across four topologies and their corresponding algorithms. *Topo1* and *Topo2* use 2 servers with 4 and 8 GPUs per server, respectively, while *Topo3* and *Topo4* use 4 servers with 4 and 8 GPUs per server. A large gap between the average and maximum idle ratios signals poor load balance. MSCCL suffers from pronounced imbalance: some TBs remain almost completely idle yet continue to consume SM capacity, with idle ratios reaching 99.9%. This waste stems from the extra communication channels MSCCL opens to increase parallelism, as discussed in §2.2.

In contrast, ResCCL consistently delivers higher thread utilization, lowers the total number of occupied threads, and achieves more balanced TB usage than MSCCL across both expert-designed and synthesizer-generated algorithms. The advantages are most pronounced when running the expert AllGather algorithm, where ResCCL sustains an effective TB utilization of more than 92.3%, improving average TB utilization by up to 26.2% over MSCCL, and reducing total TB occupancy by up to 75%. Moreover, ResCCL's maximum TB idle ratio never exceeds 21.4%, underscoring its ability to maintain well-balanced thread utilization throughout execution. When executing synthesized algorithms such as those generated by TACCL, we observe a decline in thread utilization. This is chiefly because TACCL's solver abstracts away certain real-world details, yielding synthesized algorithms that distribute link load unevenly. TECCL shows similar, if not worse, inefficiencies. Despite the execution imbalance and frequent idle phases introduced by these synthesized algorithms, ResCCL still achieves efficient resource optimization. Compared to MSCCL, ResCCL reduces thread consumption by up to 77.8% and average idle time by 41.6%.

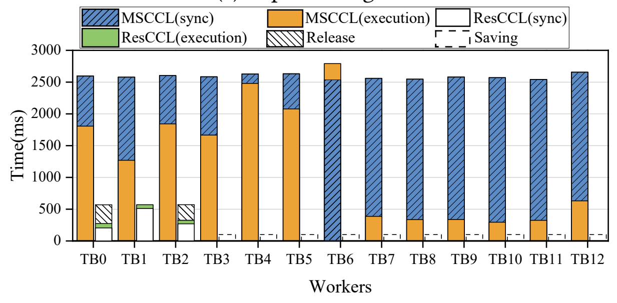
Resource utilization evaluation on V100 GPUs. Figure 12(a) and Figure 12(b) show that ResCCL reduced thread resource consumption by up to 75% compared to MSCCL when scheduling the same algorithm. Additionally, ResCCL minimized thread occupation time to as little as 3.8% of that in MSCCL, with the added flexibility of early release. Additionally, ResCCL achieves 43.4%-66.9% higher average resource utilization, indicating a more efficient and intensive execution pipeline.

5.5 End-to-End Training

We evaluate ResCCL on the distributed training of large language models (LLMs) of various scales, including both GPT-3 and T5, and compare it against the native Megatron-LM [9] implementation using the latest version of NCCL as the communication backend, as well as a Megatron variant integrated with MSCCL, all under



(a) Expert-Designed



(b) Synthesized

Figure 12: Time cost breakdown comparison of primitives for ResCCL and MSCCL executing the same expert and synthesized algorithms.

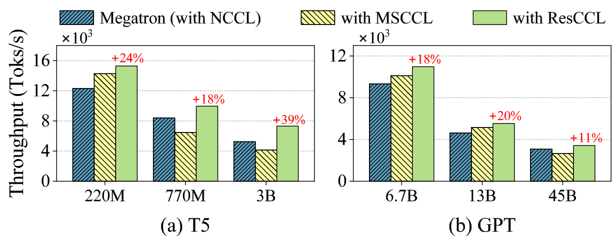


Figure 13: Training throughput for GPT-3 and T5 models of varying sizes using ResCCL as the Megatron communication backend, compared with NCCL and MSCCL.

identical settings. For models with fewer than 13 billion parameters, we deploy them on two servers (16 GPUs) with a batch size of 16. Larger models are deployed on four servers (32 GPUs) with a batch size of 32. In terms of distributed parallelism strategies, data parallelism is applied to the T5 models, while tensor parallelism is used for GPT-3 models.

Integrating ResCCL into Megatron requires only a straightforward relink and rebuild, making the integration process seamless. Figures 13(a) and 13(b) illustrate the throughput improvements achieved by using ResCCL as the collective communication backend compared to NCCL, across a range of model sizes from 220M to 45B parameters. For T5 models (220M–3B), ResCCL accelerates training throughput by 18%–39% over native Megatron and achieves 7.1%–1.8× improvement compared to Megatron with MSCCL. For larger GPT-3 models (6.7B–45B), ResCCL delivers up to 11%–20% performance improvement over native Megatron, and 7.5%–29.3% over the MSCCL-integrated variant.

6 RELATED WORK

Collective communication optimization. Prior work [2, 7, 29, 33, 40] has focused on optimizing collective algorithms, leveraging topology-aware techniques to design efficient algorithms tailored for specific network topologies. Recent research [39, 43] has taken a step further by jointly optimizing the topology and collective communication scheduling. Additionally, several open-source communication libraries offer vendor-specific optimizations, such as Nixl [14], which is designed to optimize point-to-point (P2P) communication within NVIDIA's inference framework, Dynamo. However, they have predominantly concentrated on optimizing the communication algorithms themselves, neglecting the performance bottlenecks caused by suboptimal runtime scheduling and imbalanced resource utilization. In contrast, ResCCL is the first to identify suboptimal performance arising from inefficiencies in execution strategies within existing communication libraries. ResCCL demonstrates that efficient scheduling of transmission tasks can alleviate such bottlenecks and significantly enhance communication performance.

Scheduling optimization in internet and datacenter systems. Prior work [5, 6] has extensively explored coflow scheduling in large-scale data processing frameworks, aiming to minimize overall communication time by considering inter-flow dependencies and compute placement strategies. While these efforts share conceptual similarities with the constrained transmission scheduling problem addressed by ResCCL, a key distinction lies in the complexity of the dependency structures and the heterogeneity of the communication topology in ResCCL. Additionally, a wealth of studies in HTTP/2 [18, 27, 35], RPC [16, 42], and web service [19, 34] domains focus on resource allocation and thread execution optimization. However, these approaches typically address single P2P connection scenarios. On the contrary, ResCCL tackles a more complex, many-to-many collective communication model that requires coordinated scheduling across multiple dependent transmission tasks.

7 CONCLUSION

In this work, we propose ResCCL, a resource-efficient scheduling scheme for the collective communication library, designed to address the inefficiencies in existing CCL solutions. By introducing primitive-level scheduling, dynamic resource allocation, and lightweight kernel generation, ResCCL optimizes both communication bandwidth and thread resource utilization. Our evaluation demonstrates that ResCCL outperforms existing solutions across algorithm bandwidth, TB utilization, and end-to-end training performance.

ACKNOWLEDGMENTS

We thank our shepherd, Mosharaf Chowdhury, and the anonymous reviewers for their insightful comments. This work is supported by the National Natural Science Foundation of China under Grant Nos. 62432003 and U22B2005; the LiaoNing Revitalization Talents Program under Grant No. XLYC2403086.

REFERENCES

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

[2] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2021. Synthesizing Optimal Collective Algorithms. In *Proc. of ACM PPoPP*. 62–75. https://dl.acm.org/doi/pdf/10.1145/3437801.3441620.

[3] Jiamin Cao, Yu Guan, Kun Qian, Jiaqi Gao, Wencong Xiao, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. 2024. Crux: GPU-Efficient Communication Scheduling for Deep Learning Training. In *Proc. of ACM SIGCOMM*. 1–15. https://dl.acm.org/doi/pdf/10.1145/3651890.3672239.

[4] Sanghun Cho, Hyojun Son, and John Kim. 2023. Logical/physical topology-aware collective communication in deep learning training. In *Proc. of IEEE HPCA*. 56–68. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10071117.

[5] Mosharaf Chowdhury and Ion Stoica. 2015. Efficient coflow scheduling without prior knowledge. In *Proc. of ACM SIGCOMM*. 393–406. https://dl.acm.org/doi/pdf/10.1145/2829988.2787480.

[6] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with Varys. In *Proc. of ACM SIGCOMM*. 443–454. https://dl.acm.org/doi/pdf/10.1145/2619239.2626315.

[7] Microsoft Corporation. 2022. Microsoft Collective Communication Libarary. https://github.com/microsoft/msccl.

[8] Microsoft Corporation. 2022. MSCCL Tools. https://github.com/microsoft/msccl-tools.

[9] NVIDIA Corporation. 2019. Megatron-LM. https://github.com/NVIDIA/Megatron-LM.

[10] NVIDIA Corporation. 2019. NCCL Tree Algorithm. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4.

[11] NVIDIA Corporation. 2020. CUDA Refresher: The CUDA Programming Model. https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model.

[12] NVIDIA Corporation. 2022. PTX ISA – Cache Operators. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#cache-operators.

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

[14] NVIDIA Corporation. 2025. NVIDIA Inference Xfer Library. https://github.com/ai-dynamo/nixl.

[15] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. 2023. MSCCLang: Microsoft Collective Communication Language. In *Proc. of ACM ASPLOS*. 502–514. https://dl.acm.org/doi/pdf/10.1145/3575693.3575724.

[16] Jon Crowcroft, Ian Wakeman, Zheng Wang, and Dejan Sirovica. 1992. Is layering, harmful?(remote procedure call). *IEEE Network* 6, 1 (1992), 20–24. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=120719.

[17] Daniele De Sensi, Tommaso Bonato, David Saam, and Torsten Hoefler. 2024. Swing: Short-cutting Rings for Higher Bandwidth Allreduce. In *Proc. of USENIX NSDI*. 1445–1462. https://www.usenix.org/system/files/nsdi24-de-sensi.pdf.

[18] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. 2021. Prognosis: closed-box analysis of network protocol implementations. In *Proc. of ACM SIGCOMM*. 762–774. https://dl.acm.org/doi/pdf/10.1145/3452296.3472938.

[19] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. 2013. Reducing web latency: the virtue of gentle aggression. In *Proc. of ACM SIGCOMM*. 159–170. https://dl.acm.org/doi/pdf/10.1145/2486001.2486014.

[20] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. 2024. Characterization of large language model development in the datacenter. In *Proc. of USENIX NSDI*. 709–729. https://www.usenix.org/system/files/nsdi24-hu.pdf.

[21] Ltd Huawei Technologies Co. 2024. Huawei Collective Communication Library (HCCL). https://www.hiascend.com/hccl.

[22] Changho Hwang, KyoungSoo Park, Ran Shu, Xinyuan Qu, Peng Cheng, and Yongqiang Xiong. 2023. ARK: GPU-driven Code Execution for Distributed Deep Learning. In *Proc. of USENIX NSDI*. 87–101. https://www.usenix.org/system/files/nsdi23-hwang.pdf.

[23] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 2022. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proc. of ACM ASPLOS*. 402–416. https://dl.acm.org/doi/pdf/10.1145/3503222.3507778.

[24] Tommy R Jensen and Bjarne Toft. 2011. *Graph coloring problems.* John Wiley & Sons. Available: Graph coloring problems.

[25] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *Proc. of USENIX NSDI*. 745–760. https://www.usenix.org/system/files/nsdi24-jiang-ziheng.pdf.

[26] Heehoon Kim, Junyeol Ryu, and Jaejin Lee. 2024. TCCL: Discovering Better Communication Paths for PCIe GPU Clusters. In *Proc. of ACM ASPLOS*. 999–1015. https://dl.acm.org/doi/pdf/10.1145/3620666.3651362.

[27] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. 2017. The quic transport protocol: Design and internet-scale deployment. In *Proc. of ACM SIGCOMM*. 183–196. https://dl.acm.org/doi/pdf/10.1145/3098822.3098842.

[28] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network aggregation for multi-tenant learning. In *Proc. of USENIX NSDI*. 741–761. https://www.usenix.org/system/files/nsdi21-lao.pdf.

[29] Xuting Liu, Behnaz Arzani, Siva Kesava Reddy Kakarla, Liangyu Zhao, Vincent Liu, Miguel Castro, Srikanth Kandula, and Luke Marshall. 2024. Rethinking Machine Learning Collective Communication as a Multi-Commodity Flow Problem. In *Proc. of ACM SIGCOMM*. 16–37. https://dl.acm.org/doi/pdf/10.1145/3651890.3672249.

[30] Kshiteej Mahajan, Ching-Hsiang Chu, Srinivas Sridharan, and Aditya Akella. 2023. Better Together: Jointly Optimizing ML Collective Scheduling and Execution Planning using SYNDICATE. In *Proc. of USENIX NSDI*. 809–824. https://www.usenix.org/system/files/nsdi23-mahajan.pdf.

[31] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. 2024. CASSINI: Network-Aware Job Scheduling in Machine Learning Clusters. In *Proc. of USENIX NSDI*. 1403–1420. https://www.usenix.org/system/files/nsdi24-rajasekaran.pdf.

[32] Joshua Romero, Junqi Yin, Nouamane Laanait, Bing Xie, M Todd Young, Sean Treichler, Vitalii Starchenko, Albina Borisevich, Alex Sergeev, and Michael Matheson. 2022. Accelerating collective communication in data parallel training across deep learning frameworks. In *Proc. of USENIX NSDI*. 1027–1040. https://www.usenix.org/system/files/nsdi22-paper-romero.pdf.

[33] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. 2023. TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches. In *Proc. of USENIX NSDI*. 593–612. https://www.usenix.org/system/files/nsdi23-shah.pdf.

[34] Enge Song, Yang Song, Chengyun Lu, Tian Pan, Shaokai Zhang, Jianyuan Lu, Jiangu Zhao, Xining Wang, Xiaomin Wu, Minglan Gao, et al. 2024. Canal Mesh: A Cloud-Scale Sidecar-Free Multi-Tenant Service Mesh Architecture. In *Proc. of ACM SIGCOMM*. 860–875. https://dl.acm.org/doi/pdf/10.1145/3651890.3672221.

[35] Daniel Stenberg. 2014. HTTP2 explained. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 120–128. https://dl.acm.org/doi/pdf/10.1145/2656877.2656896.

[36] Guanhua Wang, Chengming Zhang, Zheyu Shen, Ang Li, and Olatunji Ruwase. 2024. Domino: Eliminating Communication in LLM Training via Generic Tensor Slicing and Overlapping. *arXiv preprint arXiv:2409.15241* (2024). https://arxiv.org/pdf/2409.15241.

[37] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. 2024. Towards Domain-Specific Network Transport for Distributed DNN Training. In *Proc. of USENIX NSDI*. 1421–1443. https://www.usenix.org/system/files/nsdi24-wang-hao.pdf.

[38] Shuai Wang, Kaihui Gao, Kun Qian, Dan Li, Rui Miao, Bo Li, Yu Zhou, Ennan Zhai, Chen Sun, Jiaqi Gao, et al. 2022. Predictable vFabric on informative data plane. In *Proc. of ACM SIGCOMM*. 615–632. https://dl.acm.org/doi/pdf/10.1145/3544216.3544241.

[39] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. 2023. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *Proc. of USENIX NSDI*. 739–767. https://www.usenix.org/system/files/nsdi23-wang-weiyang.pdf.

[40] William Won, Midhilesh Elavazhagan, Sudarshan Srinivasan, Swati Gupta, and Tushar Krishna. 2024. TACOS: Topology-Aware Collective Algorithm Synthesizer for Distributed Machine Learning. In *Proc. of IEEE MICRO*. 856–870. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10764470.

[41] Yongji Wu, Yechen Xu, Jingrong Chen, Zhaodong Wang, Ying Zhang, Matthew Lentz, and Danyang Zhuo. 2024. MCCS: A Service-based Approach to Collective Communication for Multi-Tenant Cloud. In *Proc. of ACM SIGCOMM*. 679–690. https://dl.acm.org/doi/pdf/10.1145/3651890.3672252.

[42] Bohan Zhao, Wenfei Wu, and Wei Xu. 2023. NetRPC: Enabling In-Network computation in remote procedure calls. In *Proc. of USENIX NSDI*. 199–217. https://www.usenix.org/system/files/nsdi23-zhao-bohan.pdf.

[43] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Jason Fantl, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. 2025. Efficient Direct-Connect Topologies for Collective Communications. In *Proc. of USENIX NSDI*. 705–737. https://www.usenix.org/system/files/nsdi25-zhao-liangyu.pdf.

[44] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, et al. 2024. Resiliency at Scale: Managing Google's TPUv4 Machine Learning Supercomputer. In *Proc. of USENIX NSDI*. 761–774. https://www.usenix.org/system/files/nsdi24-zu.pdf.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A CUSTOM ALGORITHM

Figure 15 illustrates the hierarchical mesh (HM) AllGather and AllReduce algorithms we developed during experimental evaluation to enhance performance. In both algorithms, the buffer is partitioned into n equal segments—one for each GPU—referred to as *chunks*. Because the algorithm is designed for a single iteration and its logic is identical across micro-batches, we simplify the discussion by assuming a single micro-batch and in-place execution. Chunks are then indexed sequentially. The example below follows the configuration shown in the figure 15, in which each machine contains four GPUs.

HM AllGather. The HM AllGather algorithm consists of two stages:

(1) Broadcast 1. Each GPU g_i broadcasts its own Chunk c_i to all other GPUs within the same node and to its ring-aligned peer GPUs across nodes. Intra-node communication uses a full-mesh (direct send) approach, while inter-node communication employs a ring-based broadcast. **(2) Broadcast 2.** Each GPU g_i then rebroadcasts to all local GPUs the chunks it received from remote, ring-aligned peers during the first stage. This stage also uses full-mesh communication within the node.

HM AllReduce. HM AllReduce proceeds in four stages: **(1) Intra-ReduceScatter.** GPU g_i performs a full-mesh ReduceScatter with every other GPU in the same node. Specifically, g_i sends to g_j all chunks whose IDs are $j + 4x$ (for integer x) and, conversely, receives from each peer all chunks whose IDs are $i + 4x$. **(2) Inter-ReduceScatter.** g_i engages in a ring-based ReduceScatter with its ring-aligned peers across nodes, operating only on chunks whose IDs are $i + 4x$. **(3) Inter-AllGather.** The same ring group then performs an AllGather on the identical chunk subset ($i + 4x$). **(4) Intra-AllGather.** Finally, g_i conducts a full-mesh AllGather within the node, sending every chunk with ID $i + 4x$ to all other local GPUs.

B SYNTAX AND EXAMPLE OF RESCCLANG

Figure 14 illustrates the BNF syntax of ResCCLang. Figure 16 presents a ResCCLang implementation of the HM AllReduce algorithm, targeting a 32-GPU configuration across four nodes. Algorithm definition (line 1): The algorithm is defined using `def ResCCLAlgo`, which initializes a ResCCLang instance with necessary global parameters, such as topology size and buffer configuration. Parameter initialization (lines 2-4): Common variables (*e.g.*, ranks and chunk size computations) are declared to enhance code readability and reduce redundancy. Intra-node ReduceScatter phase (lines 5-12): This phase is implemented via a Python-style `for` loop, iterating over each GPU and communication step. For each ⟨GPU, step⟩ pair, the corresponding transmission parameters—including source rank, destination rank, step index, chunk ID, and communication type—are derived based on the algorithmic logic. The `Transfer` primitive is then invoked to register each transmission task. Inter-node ReduceScatter phase (lines 13-19): A ring-based ReduceScatter is performed among GPUs on the same logical track across nodes, operating exclusively on chunks with IDs of the form $i + 4x$, where i is the intra-node GPU index. Inter-node AllGather phase (lines 20-27): An AllGather follows the same ring structure and operates on the same subset of chunks. Intra-Node AllGather phase (lines 28-35): A final intra-node AllGather is executed using full-mesh communication, distributing the gathered chunks to all local peers.

This implementation demonstrates the simplicity and flexibility of ResCCLang, where communication patterns are concisely captured by the `Transfer` abstraction. Step indices explicitly define execution order, ensuring correct synchronization and dependency resolution across communication stages.

def	::=	*funcName* (*paramList*) : *stat*	Definition
func	::=	**ResCCLAlgo**	Function
paramlist	::=	**nRanks** = *digit*	Parameter List
	|	**nChannels** = *digit*	
	|	**nWarps** = *digit*	
	|	**AlgoName** = *string*	
	|	**OpType** = *opType*	
	|	**GPUPerNode** = *digit*	
	|	**NICPerNode** = *digit*	
stat	::=	*assign* | *for* | *transfer*	Statement
assign	::=	*id* = *exp*	Assignment
for	::=	**for** *id* **in range** (*exp+*) : *stat*	For Loop
transfer	::=	**transfer** (*exp**, *commType*)	Transfer Call
id	::=	*letter* (*letter* | *digit* | _)*	Identifier
exp	::=	*digit*	Expression
	|	*id*	
	|	*exp mop exp*	
	|	(*exp*)	
mop	::=	+ | - | * | / | %	Math Operator
opType	::=	**"Allgather"**	Operator Type
	|	**"Allreduce"**	
	|	**"Reducescatter"**	
commType	::=	**"recv"** | **"rrc"**	Communication Type

Figure 14: The BNF syntax of ResCCLang.

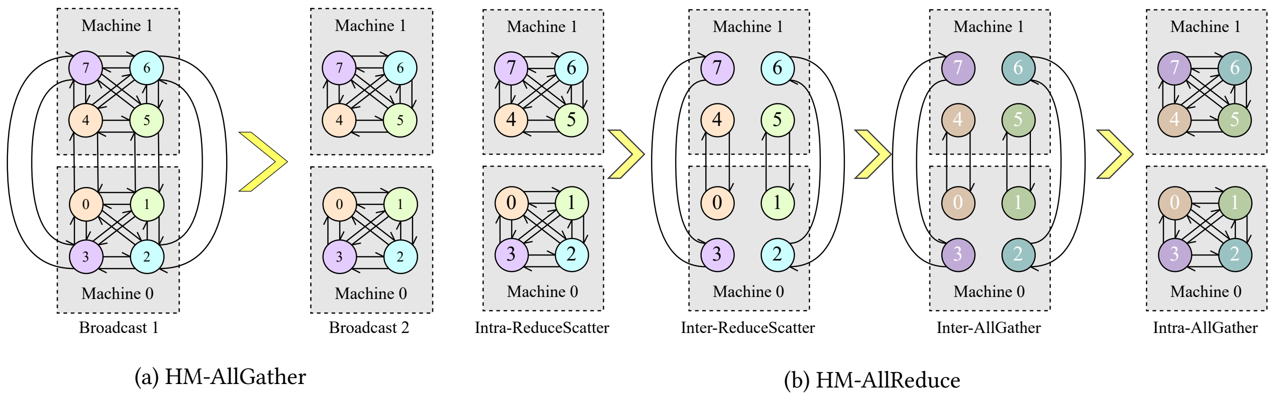


Figure 15: Custom hierarchical mesh algorithm design for a dual-node 8-GPU system.

```
def ResCCLAlgo(nRanks=32, nChannels=4, nWarps=16, AlgoName="HM", OpType="Allreduce", GPUPerNode=8, NICPerNode=8):
    nNodes = 4
    nGpusperNode = 8
    nChunks = nNodes * nGpusperNode
    for n in range(0, nNodes):
        for r in range(0, nGpusperNode):
            for baseStep in range(0, nNodes):
                for offset in range(0, nGpusperNode - 1):
                    srcRank = nGpusperNode * n + r
                    dstRank = (r + offset + 1) % nGpusperNode + nGpusperNode * n
                    step = baseStep * (nGpusperNode - 1) + offset
                    transfer(srcRank, dstRank, step, (dstRank + baseStep * nGpusperNode) % nChunks, rrc)
    for n in range(0, nNodes):
        for r in range(0, nGpusperNode):
            for baseStep in range(0, nNodes - 1):
                srcRank = nGpusperNode * n + r
                dstRank = (srcRank + nGpusperNode) % nChunks
                step = nNodes * (nGpusperNode - 1) + baseStep
                transfer(srcRank, dstRank, step, (srcRank + nChunks - baseStep * nGpusperNode) % nChunks, rrc)
    for n in range(0, nNodes):
        for r in range(0, nGpusperNode):
            for baseStep in range(0, nNodes - 1):
                srcRank = nGpusperNode * n + r
                dstRank = (srcRank + nGpusperNode) % nChunks
                step = nNodes * (nGpusperNode - 1) + nNodes - 1 + baseStep
                chunkId = (srcRank + nChunks - (baseStep + nNodes - 1) * nGpusperNode) % nChunks
                transfer(srcRank, dstRank, step, chunkId, recv)
    for n in range(0, nNodes):
        for r in range(0, nGpusperNode):
            for baseStep in range(0, nNodes):
                for offset in range(0, nGpusperNode - 1):
                    srcRank = nGpusperNode * n + r
                    dstRank = (r + offset + 1) % nGpusperNode + nGpusperNode * n
                    step = nNodes * (nGpusperNode - 1) + 2 * nNodes - 2 + baseStep
                    transfer(srcRank, dstRank, step, (srcRank + baseStep * nGpusperNode) % nChunks, recv)
```

Figure 16: Example program in ResCCLang.