AdapCC: Making Collective Communication in Distributed Machine Learning Adaptive

Xiaoyang Zhao, Zhe Zhang, Chuan Wu
Department of Computer Science, The University of Hong Kong, Email: {xyzhao, zzhang2, cwu}@cs.hku.hk

***Abstract*—As deep learning (DL) models continue to grow in size, there is a pressing need for distributed model learning using a large number of devices (e.g., GPUs) and servers. Collective communication among devices/servers (for gradient synchronization, intermediate data exchange, etc.) introduces significant overheads, rendering major performance bottlenecks in distributed learning. A number of communication libraries, such as NCCL, Gloo and MPI, have been developed to optimize collective communication. Predefined communication strategies (e.g., ring or tree) are largely adopted, which may not be efficient or adaptive enough for inter-machine communication, especially in cloud-based scenarios where instance configurations and network performance can vary substantially. We propose AdapCC, a novel communication library that dynamically adapts to resource heterogeneity and network variability for optimized communication and training performance. AdapCC generates communication strategies based on run-time profiling, mitigates resource waste in waiting for computation stragglers, and executes efficient data transfers among DL workers. Experimental results under various settings demonstrate 2× communication speed-up and 31% training throughput improvement with AdapCC, as compared to NCCL and other representative communication backends.**

***Index Terms*—distributed training, collective communication**

I. Introduction

With the growth of DL model sizes [1], parallel training of a deep neural network (DNN) using multiple GPUs on multiple servers has become the norm. Collective communication among the devices is required in distributed training for various purposes, including gradient synchronization with AllReduce [2], token dispatching with AlltoAll (as in the MOE framework) [3] and parameter sharding with AllGather in tensor parallelism [4]. Cross-server communication incurs significant overhead in distributed training, occupying up to 50% - 90% of the total training time [5] and preventing linear scaling of training throughput with computation capacity.

A number of libraries have been developed for accelerating collective communication, e.g., NCCL [2], Gloo [6] and MPI [7], which use rings or trees as communication graphs. Several hierarchical methods are also exploited [8]–[12]. These libraries are mostly optimized for a homogeneous environment (same type of devices, stable and homogeneous interconnects). They may not perform efficiently in a less homogeneous setting, e.g., in a cluster where machines of different configurations can be used [13]. In those cases, more adaptive communication strategies are needed, to best cater to the heterogeneity and dynamics, minimize idle time of resources, and maximize distributed training speed.

This work was supported in part by grants from Hong Kong RGC under the contracts HKU 17208920,17204423 and C7004-22G(CRF).

There have been efforts on generating better communication graphs among devices. Blink [14] detects GPU placements in servers and constructs spanning trees for intra-server communication. TACCL [15] formulates an integer problem to decide the routing of each data chunk before rendering the corresponding NCCL kernel, based on communication sketches provided by users. PLink [16] searches for a clustering of servers for a two-level hierarchical reduction strategy. SCCL [17] solves an integer program to generate a pareto-optimal chunk fusion solution, achieving latency-bandwidth tradeoffs of a collective. These designs are all built on top of existing communication libraries, e.g., NCCL or Gloo, which adopt static communication graphs, and may not adapt well to workload interference and link variation in a shared cluster, over the long training period of large models [18].

We enhance the adaptivity of the communication system from three aspects. *First*, communication graphs that best utilize heterogeneous links or devices should be identified for each collective operation, enabling efficient training using diverse resources. *Second*, communication strategies should be adaptive, by efficiently reconstructing communication graphs, deciding chunk size and the need for partial aggregation based on runtime system performance. *Third*, the communication system should work closely with the DL framework (e.g., PyTorch [19]) and be aware of the training status (e.g., computation progress at the workers), to allow efficient communication schedules accordingly.

We design AdapCC (**Adap**tive **C**ollective **C**ommunication Library), a new communication system that provides efficient collective primitives with optimized adaptive communication strategies to accelerate distributed training. AdapCC is open-sourced at *https://github.com/joeyyoung/adapcc*. Our contributions are summarized as follows:

▷ AdapCC integrates a detection module (for inferring GPU placements) and a profiling module (for profiling link properties). It coordinates workers to enable efficient profiling on the fly during training, capturing dynamic network changes.

▷ AdapCC automatically generates optimized communication strategies for different collective primitives based on the profiling results. It solves a network optimization problem to identify communication graphs, chunk size, and aggregation control. AdapCC reconstructs the communication graph accordingly during training, without any need of checkpointing, terminating and restarting the training job.

▷ AdapCC identifies stragglers in each training iteration and decides whether to wait or proceed with communication

among ready workers based on an online algorithm. In the latter case, GPUs of non-ready workers can be used as relays for communication expedition. We also enhance AdapCC with efficient fault recovery capabilities during training.

▷ AdapCC generates CUDA code on each worker for multi-stream parallelism of each collective and manages GPU buffers efficiently. Extensive experiments show up to 2× communication speed-up and 31% training throughput improvement when training various representative DNN models in comprehensive settings, as compared to NCCL.

II. Background and Motivation

A. Resource Heterogeneity

The rapid development of hardware has led to heterogeneity of server configuration and GPU interconnection in a cluster. Link throughput differs across different generations of the same type of links: PCIe throughput has increased from 250 MB/s to 2 GB/s, and NVLink4.0 achieves 900GB/s on H100, almost ten times faster than NVLink1.0 on M40 [20]. NCCL typically assumes a strongly homogeneous scenario, where empirical bandwidth values are assigned to each type of link when constructing the communication graph. It fails to capture the complexities of the actual connections.

Non-regular GPU interconnect topologies often lead to inefficient utilization of inter-GPU links. For instance, when GPUs without direct NVLinks are allocated to a training job, e.g., due to unavoidable resource fragmentation in an IaaS cloud [21], NCCL is unable to form an NVLink ring and falls back to a less efficient PCIe ring instead. Blink [14] constructs topology-aware spanning trees to resolve the problem. However, it considers intra-server communication only and neglects the potential heterogeneity of network links and GPU devices when training with multiple servers. In practice, servers could be equipped with NICs that have varying bandwidths, ranging from 1 Gbps to 200 Gbps, and use different network stacks, such as RDMA or TCP. The adoption of heterogeneous GPUs among servers is also gaining increasing attention, due to the short release cycle of new GPU architectures [13] [22].

Issue 1: Constructing communication graphs that best utilize heterogeneous links/devices for collective operations can enable efficient training over diverse resources, which has not been adequately addressed in existing DL communication systems. AdapCC generates optimized communication strategies that are adaptive to various allocated resources and link properties. Meanwhile, heterogeneity is not mandatory to gain benefits from AdapCC.

B. Volatile Network Status

Network performance varies in a shared cluster. Modern data centers typically adopt a hierarchical network architecture where server-to-server bandwidth is non-uniform due to cross-traffic [16]. In Fig. 1, we measure bandwidth and network latency between two instances (reserved with 16vCPU and 15Gbps throughput) on a public cloud during a 6-hour period. We observe that the performance varies substantially, degrading from the peak as much as 34% and 17%, in terms of bandwidth and latency, respectively. In scenarios where

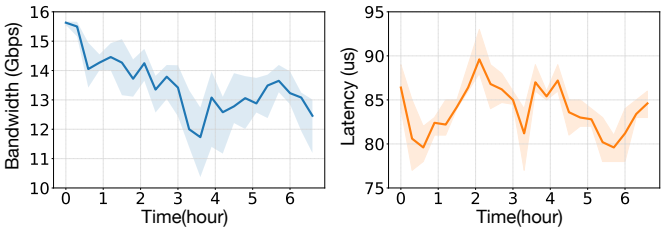


Figure 1: 6-hour profiling using iperf3 and netperf.

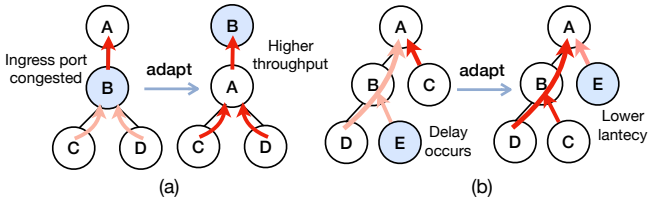


Figure 2: Adaptively construct communication graph.

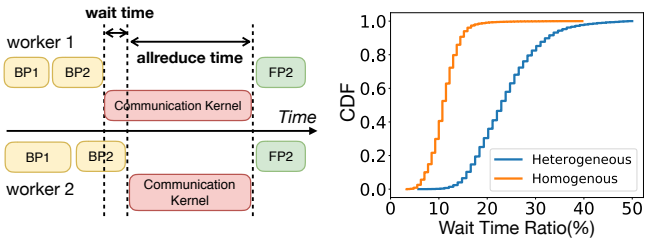
co-located workloads within the same server contend for bandwidth [20], the achieved network performance could be more unpredictable.

When determining the communication strategies (i.e., communication graph, chunk size), Blink [14] uses empirical values for inter-device link bandwidth (similar to NCCL) and assumes it stable during training. TACCL [15] requires users to provide a topology sketch including detailed information about actual link performance between GPUs. However, the sketch is only exploited in the initialization stage and the measured values are assumed stable afterwards. Under practical volatile link status, these communication systems are not able to automatically perceive the network dynamics and the training performance may not be ideal (evaluated in Sec. VI-D). Incorporating a new strategy into existing communication systems by periodically re-running them necessitates the intervention of developers (e.g., providing new sketches as in TACCL), which is not user-friendly. Moreover, this process requires restarting the training process, involving tasks such as checkpointing gradients, rebuilding the process group, and restoring the model, resulting in a considerable overhead as evaluated in Sec. VI-E.

Issue 2: communication strategies should be adaptive to the changing network status during training, by efficiently reconstructing the communication graphs and choosing the chunk sizes without relaunching the model. AdapCC employs profiling to track network status on the fly and reconstructs the communication graph accordingly. In the example depicted in Fig. 2(a), workers communicate with a Reduce primitive; when the ingress bandwidth of server B decreases (e.g., due to cross traffic), AdapCC can adjust the communication topology accordingly for better throughput. In Fig. 2(b), when data transmission from server E is delayed (e.g., due to interference caused by co-located workloads), the topology can be dynamically adjusted to enable partial gradients aggregation of other servers first.

C. Computation Stragglers

A collective primitive is usually triggered when all workers have completed the computation of the respective tensors [12]. However, completion of tensor computation at the workers



(a) timeline example (b) measured time

Figure 3: (a) BP_i and FP_i denote the backward and forward pass of computation operator i, respectively. (b) Cumulative distribution function (CDF) of the wait time ratio collected in GPT-2 training.

may not happen strictly at the same time, which causes waiting time in the communication kernel as shown in Fig. 3(a).

To quantify this, we train GPT-2 model in both heterogeneous and homogeneous settings, using a local batch size of 16. In heterogeneous case, we use two servers each with four V100 GPUs and two servers each with four A100 GPUs; in homogeneous case, we employ four servers each with four A100 GPUs. Each server is configured with 100Gbps RDMA network. In Fig. 3(b), we measure the time taken by the fastest worker to wait for the slowest worker to be ready for an AllReduce operation in each iteration. In heterogeneous case, the wait time accounts for more than 23% of the actual communication time (denoted as wait time ratio) in 50% of the observed iterations; even in homogeneous case, the wait time ratio is larger than 10% in 50% of the observed iterations. This ratio could further increase with a larger batch size, a greater number of training GPUs, or a higher degree of heterogeneity in server configurations. We further observe that in a production hybrid cluster, where online services are run on the CPUs and offline training is run on the GPUs, the straggler problem is more likely to occur due to workload interference.

Issue 3: The communication system should enable flexible collective communication among selected subsets of workers for reduced waiting time and improved training throughput. With NCCL, only ranks within the pre-defined *nccl communicator* context can participate in a collective [23], and changing the workers involved requires terminating the job and relaunching the process group. AdapCC aims to enable participation of an arbitrary set of workers in collective communication and dynamically determine the workers to trigger communication, adaptive to real-time training status. In addition, it exploits GPUs of non-ready workers as relays for data transfer to maximize communication speed.

This adaptivity further enhances the fault tolerance capability in distributed training. In the event of worker failures, NCCL experiences hang-ups and blocks the training process. To recover training, it becomes necessary to checkpoint the model and relaunch the training job. In contrast, AdapCC allows for continued communication and efficient recovery of training among the remaining workers.

III. Design Overview

A DNN job runs across multiple servers or cloud instances. Without loss of generality, we use *instances* in our presentation. Each instance contains multiple GPUs, and each worker

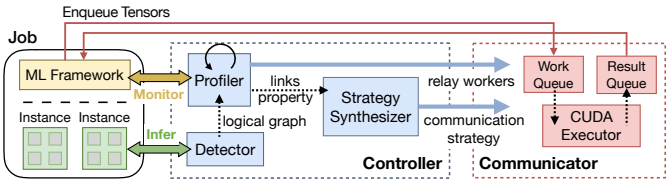


Figure 4: AdapCC System Overview

(aka rank) runs on one GPU. AdapCC runs on each worker and has two modules, as shown in Fig. 4. *Controller* detects inter-worker connectivity, profiles link properties, monitors computation delay of workers, and produces communication strategies. *Communicator* executes respective communications after retrieving tensors from the ML framework.

Detector infers inter-GPU connectivity on each instance by coordinating GPUs on an instance to send bandwidth or latency probes (**Sec. IV-A**). The information collected by local rank0 on each instance about GPU interconnections and NIC affinity is then negotiated among instances to construct a logical topology that connects all GPUs in the job.

Profiler probes properties (e.g., latency, bandwidth) of inter-GPU connections periodically given the detected logical graph (**Sec. IV-B**). Profiled results are gathered by (world) rank0 and provided as input to the synthesizer. A coordinator on rank0 collects the tensor ready times computed by workers to select non-ready workers as relays and schedule partial communication in each iteration (**Sec. IV-C**).

Synthesizer on rank0 incorporates an optimizer (**Sec. IV-D**) to generate communication strategies for a respective primitive that include: (i) parallel communication graphs, with the transmission chunk size; (ii) whether partial tensor aggregation should be performed on each GPU, balancing between bandwidth contention and synchronization delay. The strategies will be dispatched to other workers.

Queues store consecutive communication requests. In each iteration, tensors are pushed into the *Work Queue* by the ML framework and executed in order. Communicated tensors are fetched from the *Result Queue* for continued computation.

Executor parses communication strategies from *Controller*, together with the information of relay workers to generate respective CUDA codes. It splits a tensor into multiple partitions and executes parallel communication with concurrent GPU streams to maximize the throughput (**Sec. V**).

IV. Control Strategy

We now present the *Controller* design.

A. Inferring Logical Topology

Detectors on workers are triggered during the initialization stage of a training job or when a new worker joins the job (e.g., elastic scaling scenario). They first cooperatively decide the topology connecting all GPUs within an instance, including: (1) The CPU affinity of NICs in the instance, which is determined by binding the host process of the local rank0 GPU to different NUMA nodes and doing a socket loopback to each NIC. The smallest latency measured in each case tells which NUMA node is nearest to the NIC card.

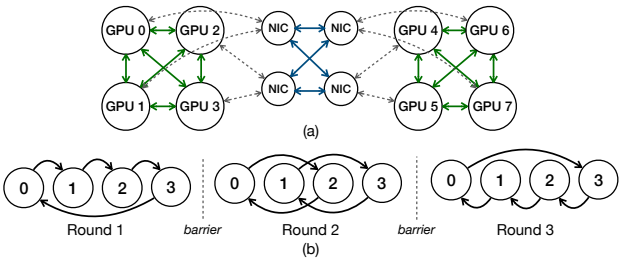


Figure 5: (a) Logical topology: green line denotes NVLink, blue line is network transmission, and the dotted line indicates PCIe transfer. (b) Example of the second stage profiling, where four instances perform three probing rounds in order.

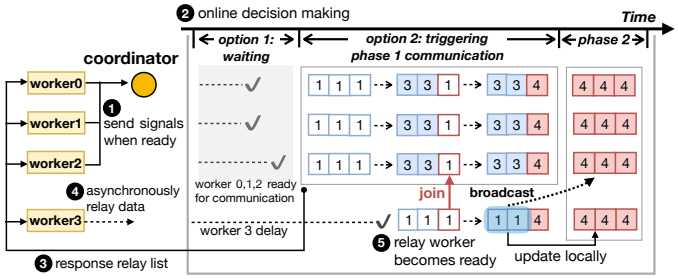


Figure 6: The coordinator assigns non-ready worker 3 as a relay. The example shows a collective communication involving tensors of value 1, resulting in an aggregated tensor of value 4 on each worker.

(2) Whether two GPUs are under the same PCIe switch. For each pair of GPUs on the same instance, we allow one GPU to send 20MB of data to the CPU via 8 parallel transmissions, while the other GPU sends 20MB to the CPU simultaneously with bandwidth measured. Low bandwidth indicates contention on the shared PCIe bus, implying locality. (3) The PCIe locality of NICs. For each GPU pair (detected in (2)), one GPU copies data to the CPU while the CPU conducts a socket loopback to the affinity NIC (detected in (1)). The lowest bandwidth achieved by the GPU copy indicates the locality with NIC due to the competition of PCIe.

Once these tasks are completed on each instance, we consider the instance-to-instance connectivity as a fully connected graph, focusing on the instance-to-instance performance [24].

B. Profiling Link Properties

Profilers on workers conduct profiling based on constructed logical topology (Fig. 5(a)). We use α-β cost model [15] to quantify the property (latency and bandwidth) for each link connecting a pair of local GPUs and each network connection between two NICs. We do not profile PCIe links as the data movement could be overlapped with network transmissions. Here, α is the link latency, and β is the inverse of the link bandwidth. Specifically, we target three types of interconnects: (1) NVLink to connect local GPUs; (2) inter-server RDMA connections; and (3) inter-server TCP connections.

The profiling is triggered at a frequency (e.g., every 500 iterations) that can be customized with our API. To ensure efficient and interference-free profiling on the fly, all instances perform intra-instance GPU-to-GPU profiling, and then move on to inter-instance profiling for NIC-to-NIC links.

Between each GPU pair on an instance, we send a piece of data (with size s) n times and measure the transmission time $n(\alpha + \beta * s)$. Then a group of data with size $n * s$ is sent at once, and the transfer time is $\alpha + \beta * n * s$. Several measurements are performed under different values of n and s, and then α and β for this link can be derived.

For inter-instance profiling, we use a multiple-round scheme to prevent interference among different profiling flows, as in Fig. 5(b). With N instances, there are $N-1$ profiling rounds, each with a synchronization barrier. In round i, the profiler on an instance n sends probes (similar to the GPU-to-GPU profiling flows) to the instance $(n+i)\%N$ simultaneously. This consensus ensures the best flow parallelization, with only one transmission in any ingress or egress port at a time.

Model training is blocked during the profiling to eliminate interference. Profiled results are gathered in rank0 GPU and fed to the synthesizer, which solves an optimization problem to determine the communication strategies (Sec. IV-D). If the resulting communication graph is unchanged, training proceeds to the next iteration directly; otherwise, it waits for the completion of graph reconstruction before proceeding. Throughout the procedure, we do not need to checkpoint the model and the training can resume immediately (the overhead of graph reconstruction is evaluated in Fig. 19(c)).

C. Adaptive Relay Control

AdapCC achieves flexible communication among an arbitrary set of workers, adapting to variations in the computation completion times across workers in each iteration.

In Fig. 6, a coordinator on the rank0 worker receives communication requests from each worker after completing their computation. It periodically (with a 5ms time cycle) makes decisions between two options: (1) waiting for all workers to be ready (i.e., complete the precedent computation) for collective communication, or (2) triggering a partial communication for ready workers (referred to as *phase 1*). The partial communication may not lead to the same model accuracy as involving all workers in each collective. To maintain the same model accuracy, after *phase 1*, the tensor data of workers that become ready later is broadcast to others in *phase 2* so that each worker holds the complete tensors for aggregation.

To improve communication performance in option (2), during *phase 1*, the coordinator assigns non-ready workers as relays and shares the relay list with all workers. GPUs managed by relay workers asynchronously participate in communication as intermediaries, relaying tensor data from other workers using dedicated backend threads (without aggregating the worker itself's tensor). This allows utilizing additional links for communication among ready workers, with better aggregated bandwidth. If relay workers become ready during *phase 1*, i.e., when computed tensor data fills the GPU memory buffer (buffer management is detailed in Sec. V-A), data chunks with the same offset in the buffer join the ongoing aggregation process. Then in *phase 2*, only partial data chunks from relay workers that have not been aggregated need to be broadcast for aggregation. These chunks are locally combined

with aggregated results from *phase 1* (obtained from the relay GPU's result queue) to obtain the final tensor.

We address three challenges in our design. *Firstly*, whether the coordinator should choose option (1) (wait) or (2) (proceed with immediate communication) in each time cycle. This decision is crucial as waiting, like in existing communication libraries, leads to time waste. However, allowing active workers to communicate first brings overhead in *phase 2* for model update consistency. AdapCC employs an online ski-rental algorithm to guide decision-making. *Secondly*, when relay workers remain not ready, such as due to faults, training recovery is needed. *Lastly*, behaviors of each GPU should be defined for arbitrary relay control on a communication graph.

1–Waiting or Proceeding with Communication. In a ski-rental problem [25], a person decides whether to rent skis for an additional day or buy a pair of skis, with each option having a different cost. Our objective is to minimize overall communication time of each collective, where each time cycle represents a decision-making day. Waiting for all workers to be ready corresponds to renting, while proceeding with immediate communication corresponds to buying. Waiting incurs a cost of 1 in each time cycle, and there is a constant time cost for completing collective communication when all workers become ready. The time cost can be estimated by dividing the total communicated data volume S by the aggregate bandwidth B among workers [20]. For example, in AllReduce, S is equal to $2(N-1)$ times the tensor size of each worker, where N is the number of participating workers; in AlltoAll, S is equal to N times the tensor size; and in Broadcast, S is equal to the tensor size. B is obtained by accumulating the profiled link bandwidth in the communication graph. The buying cost is the estimated time spent on *phase 1* and *phase 2* of option (2), and this cost can vary over time due to changes in the number of ready workers, which in turn affects the data volume S.

We employ the break-even algorithm, known as the best deterministic method of the ski rental problem with a competitive ratio of 2. Specifically, the coordinator waits at most until the accumulated waiting cost exceeds the buying cost in a time cycle. If all workers are ready during this waiting period, they engage in communication together. Otherwise, *phase 1* communication is initiated among the active workers, followed by *phase 2* communication. By monitoring the training status and enabling relay GPUs, AdapCC enables adaptability and outperforms naive waiting policies in existing libraries.

2–Fault Tolerance. NCCL may cause training to hang when a faulty worker occurs, and the entire job is required to restart. In our design, after a period of time T_{fault} has passed since the completion of *phase 1* communication, the coordinator identifies the remaining non-ready workers as faulty workers. The threshold T_{fault} is set as five times the duration since the fastest worker became ready (PyTorch Elastic uses a keep-alive signal with a duration of 15 seconds to identify any fault). These faulty workers are excluded from the training group. Each remaining worker proceeds with the model update of the current iteration using the communicated tensor. Afterwards, the coordinator notifies the data loader of remaining workers for a redistribution of the training data, to ensure that the global batch size remains consistent throughout the whole training process. With this fault tolerance, AdapCC enables continued training efficiently without interruption.

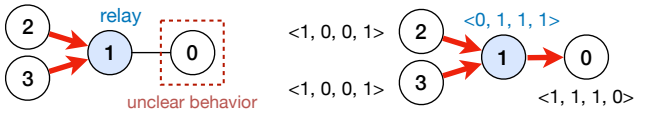


Figure 7: Illustration of the GPU behavior abstraction.

3–GPU Behavior Abstraction. Current communication libraries [2] [15] [17] [26] [27] lacks the flexibility to control GPUs as relays. For example, when constructing the communication channels in NCCL, each rank only interacts with the memory buffers of its previous and next ranks. The channel information is initialized for the NCCL communicator and cannot be adjusted without relaunching. A Reduce kernel is thus pre-defined on each rank: chunks are received from the previous rank, reduced with data in the current rank, and then sent to the next rank. However, the introduction of a relay GPU breaks the *receive-reduce-send* dependency. For example of a 4-GPU reduce graph in Fig. 7(a), where GPU1 acts as a relay and does not contribute tensors in the aggregation. Since GPU0 only interacts with GPU1, it cannot decide whether to wait for receiving tensors from GPU2 and GPU3. Also, if GPU2 is not ready, GPU1 does not need to launch the aggregation kernel but can directly relay traffic from GPU3 to GPU0.

To abstract the complex behavior of each GPU on a given communication graph with arbitrary ready workers, we share the complete communication graph structure generated by the synthesizer with each rank and define a tuple with four Boolean values, *<isActive, hasRecv, hasKernel, hasSend>*:

- *isActive*: it indicates whether the current GPU rank is active or not. A GPU rank is active if the worker is ready for communication (i.e., not a relay). Each rank holds the active status of all ranks provided by the coordinator.
- *hasRecv*: it decides whether a GPU should wait for receiving data from its direct predecessors. Each rank recursively checks whether its predecessors have data to send, and the flag is set as long as an active rank is found.
- *hasKernel*: it implies whether an aggregation kernel should be launched to aggregate received data and local data. For collectives such as AlltoAll and Broadcast, this flag is not set. For Reduce, AllReduce or Reduce-Scatter collectives, we always set the flag unless one of the following conditions is met: (1) *hasRecv* is not set. The current rank only needs to send its local data to its successor. (2) *isActive* is false and there is only one active precedent. Hence, the rank acts as a relay and aggregation is not needed since it only receives data from one rank. (3) The synthesizer explicitly indicates that aggregation will not be done on the rank, and data received from different precedent ranks will be directly sent to subsequent ranks without synchronization (Sec. IV-D).
- *hasSend*: it implies whether the rank should launch sending

events to its successor. The flag will not be set when both *isActive* and *hasRecv* are false. For ranks without a subsequent rank (e.g., root in a tree), *hasSend* is false.

Once the coordinator has decided the relay workers, the behavior tuple of each GPU is decided. Fig. 7(b) gives the behavior tuples of the GPUs when GPU1 plays the relay role. Based on the behavior tuple, the *communicator* then generates CUDA code, which determines actions such as waiting for data from predecessors, launching the aggregation kernel, and sending data to successors. It ensures that on a constructed communication graph, arbitrary GPU relay control can be applied without reconstructing the graph.

D. AdapCC Synthesizer

We next describe how the synthesizer produces communication strategies based on the logical graph G (Fig. 5) with profiled link information. We denote GPU node set as G_{gpu} and NIC node set as G_{nic}. The set of edges is denoted as E.

We formulate strategies for Reduce, Broadcast and AlltoAll, which represent many-to-one, one-to-many and many-to-many primitives, respectively. These optimizations can be readily used for other primitives: for AllReduce, we generate strategies for Reduce and execute Broadcast reversely; for AllGather, we use a combination of one Broadcast on each GPU.

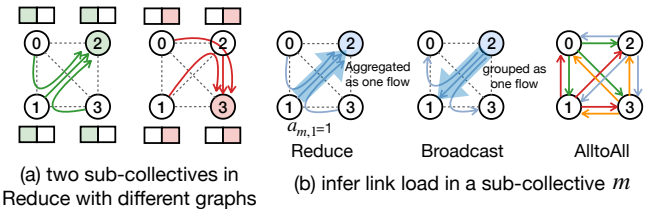
In a collective communication, each GPU $g \in G_{gpu}$ has a tensor of total size S to communicate, which is divided into M partitions for parallel sub-collectives (M is a tunable system parameter), as shown in Fig. 8(a). Each sub-collective $m \in M$ communicates one tensor partition of size S_m ($\sum_{m\in M} S_m = S$) and has a different communication graph.

For Reduce and Broadcast, there is a root GPU g in each sub-collective m, indicated by a binary variable $r_m^g = 1$ ($r_m^g = 0$ for other non-root GPUs), which aggregates gradients from or broadcasts parameters/tokens to other GPUs. Consider the tensor data sent out from a GPU g in sub-collective m as a flow set F_m^g. In Reduce, F_m^g contains one flow from GPU g to the root; in Broadcast, only the root's flow set is non-empty, including multiple flows sending aggregated parameters to non-root GPUs; in AlltoAll, each GPU sends multiple flows to all other GPUs. The source and destination GPUs of a flow f are denoted by Src_f and Dst_f, respectively. F_m is the set of all flows in sub-collective m, i.e., $F_m = \{F_m^g\}_{g\in G_{gpu}}$.

Routing Decision. The synthesizer generates routing paths for each flow, indicated by binary variables $x_{i,j}^f$ with $x_{i,j}^f = 1$ if flow $f \in F_m$ traverses edge $(i,j) \in E$ and $x_{i,j}^f = 0$, otherwise. The following constraints applied to each node $i \in G$ ensure that a flow starts from the source, ends at the destination, and complies with flow conservation at intermediate nodes:

$$\forall f \in F_m: \sum_{j:(i,j)\in E} x_{i,j}^f - \sum_{i:(j,i)\in E} x_{j,i}^f = \begin{cases} 1, & i = Src_f \\ -1, & i = Dst_f \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

Chunked Transmission. For each sub-collective m, tensor data is further divided into chunks with size C_m (a decision variable) for pipelined transmission. The transmission time of each chunk in flow f on the edge (i,j) is computed as $t_{i,j}^f = \alpha_{i,j} + \tilde{\beta}_{i,j} C_m$, where $\alpha_{i,j}$ is the latency profiled on link (i,j) and $1/\tilde{\beta}_{i,j}$ is the available bandwidth competed by concurrent flows (to be decided later in 'Bandwidth Sharing').



(a) two sub-collectives in Reduce with different graphs

(b) infer link load in a sub-collective m

Figure 8: In each sub-collective (with a dedicated graph), GPUs send gradients, parameters or tokens to corresponding destinations.

We use a binary variable $a_{m,g}$ to decide aggregation on node $g \in G_{gpu}$ ($a_{m,g} = 0$ if $g \in G_{nic}$): when $a_{m,g} = 1$, an aggregation kernel is launched on node g to synchronize and aggregate local data chunks with received chunks from precedent nodes; for primitives not requiring aggregation, $a_{m,g} = 0$. Taking the Reduce graph in Fig. 8(b) as an example, GPU1 needs to wait for the same chunks sent from GPU0 and GPU3 before launching an aggregation kernel if the respective $a_{m,g}$ is 1; the aggregated gradients, whose size is only one-third of the data volume when directly forwarding gradients without aggregation, are then transmitted to GPU2.

We use h_j^f to indicate the time when a chunk of flow f is ready at node $j \in G$ to be sent to the successor. When aggregation is not done on node j for sub-collective m that flow f belongs to ($a_{m,j} = 0$), we compute the time according to the ready time of the chunk at j's predecessor i and the transfer time $t_{i,j}^f$ on link (i,j); when $a_{m,j} = 1$, a chunk at node j is sent to the successor only when the same chunks from all other flows, which traverse node j have arrived:

$$\forall m \in M, f \in F_m, j \in G, (i,j) \in E, x_{i,j}^f = 1:$$
$$h_j^f = \begin{cases} h_i^f + t_{i,j}^f, & \text{if } a_{m,j} = 0 \\ \max_{\hat{f}, i: x_{i,j}^{\hat{f}} = 1} \{h_i^{\hat{f}} + t_{i,j}^{\hat{f}}\}, & \text{if } a_{m,j} = 1 \end{cases} \quad (2)$$

The *max* indicates waiting for the slowest flow to arrive.

Bandwidth Sharing. The profiled bandwidth $1/\tilde{\beta}_{i,j}$ of each link (i,j) is equally shared by traversing flows. $N_{i,j}^m$ denotes the traffic load contributed by sub-collective m (Fig. 8(b)).

▷ For Reduce, when aggregation is done at node i (i.e., $a_{m,i} = 1$), only one flow sent out to the subsequent node j contributes to the link load:

$$\forall m \in M, (i,j) \in E: N_{i,j}^m = \mathbb{I}\big(\exists f \in F_m: x_{i,j}^f = 1\big)$$

Otherwise, traversing flows are sent to node j individually and the load on edge (i,j) equals the sum of flows generated by node i and flows routed from precedent edges:

$$\forall m \in M, (i,j) \in E:$$
$$N_{i,j}^m = \mathbb{I}\big(\exists f \in F_m: x_{i,j}^f = 1\big)\big(|F_m^i| + \sum_{(k,i)\in E} N_{k,i}^m\big)$$

$|F_m^i|$ indicates the number of flows sent out from node i.

▷ For Broadcast, each flow sent out from the root carries a replica of the same data, and flows traversing the same link are grouped as one flow to contribute to the link load:

$$\forall m \in M, (i,j) \in E: N_{i,j}^m = \mathbb{I}\big(\exists f \in F_m: x_{i,j}^f = 1\big)$$

▷ For AlltoAll, flows carry different parameters, and the link load sums all traversing flows:

$$\forall m \in M, (i,j) \in E : N_{i,j}^m = \sum_{f \in F_m} x_{i,j}^f$$

Considering bandwidth sharing among all sub-collectives, the available bandwidth for each flow on link $(i,j) \in E$ is:

$$\frac{1}{\tilde{\beta}_{i,j}} = \frac{1}{\beta_{i,j} \sum_M N_{i,j}^m} \tag{3}$$

Objective. We aim to derive the communication strategies (routing paths of sub-collectives, chunk size and aggregation control) that minimize the completion time of the collective communication, i.e., when all flows have been transferred:

$$\text{minimize} \max_{m \in M, f \in F_m} T_f \tag{4}$$

where T_f denotes the finish time of flow f. With the pipelined chunk transmission design, transferring $\lceil S_m/C_m \rceil$ chunks of a flow f of sub-collective m takes the following time:

$$\forall m \in M, f \in F_m : T_f = h_{Dst_f}^f + \lceil S_m/C_m \rceil T_{bottle}^f \tag{5}$$

where T_{bottle}^f denotes the transmission time on the bottleneck link along the flow routing path, i.e.,

$$\forall m \in M, f \in F_m : T_{bottle}^f = \max_{(i,j) \in E : x_{ij}^f = 1} \{h_j^f - h_i^f\} \tag{6}$$

The optimization problem, with objective in (4) subject to constraints in (1)-(3)(5)(6), is NP-hard with mixed integers. The synthesizer uses the state-of-the-art solver Gurobi [28] to derive a solution, with the solving time under different scales measured in Fig. 19(c). The strategies are output in an XML format and parsed by the *Communicator*.

V. Data Communication

We next present the *Communicator* design.

A. Parallel Transmissions

A distributed training system typically utilizes multiple processes, where each process manages a GPU device. To manage parallel sub-collectives in the same GPU process, we introduce the concept of *transmission context* (each for one concurrent sub-collective), as illustrated in Fig. 9.

Each *transmission context* has a unique ID shared among all processes, used for sub-collective identification. For each *context*, a persistent thread is launched to continuously poll data from the *Work Queue*. Especially for AllReduce, a *context* executes a Reduce thread and a Broadcast thread simultaneously to pipeline the two stages. Unlike NCCL which uses a single CUDA stream, we use one stream per thread to parallelize the transmissions.

Three types of memory buffers are registered for each *context*: (1) *local buffer*, which stores data of the current GPU to communicate; (2) *receive buffer*, which receives data sent from precedent GPU nodes; (3) *result buffer*, which contains communicated data to be pushed into the Result Queue. To enable threads in one process to access *receive buffer* in another process for data transmissions, AdapCC utilizes CUDA inter-process communication (IPC) technology.

AdapCC sets up each *transmission context* in a distributed manner as shown in Fig. 10. We first allocate the memory

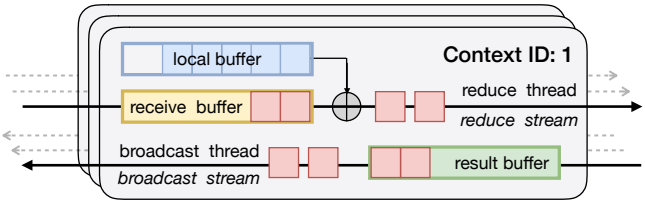


Figure 9: Illustration of *transmission contexts* managed by one GPU process in an AllReduce collective.

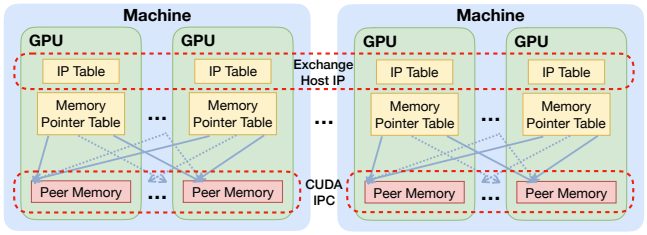


Figure 10: Set-up phase of one *transmission context*.

buffers in GPUs for receiving data and expose their memory pointers through CUDA IPC handles. Once all *contexts* on different processes with the same *context ID* have obtained their own IPC handles, an AllGather communication is performed to enable each thread in the *context* to get all the IPC handles on the same server with the same ID and store them in a memory pointer table. In this way, memory allocated on other GPUs on the same machine can be directly accessed for peer-to-peer transmission. Since CUDA IPC only works within the same server, *contexts* on different servers also exchange host IPs to form an IP table for across-server transmissions.

The overhead of allocating GPU memory and performing IPC is non-negligible. Nonetheless, AdapCC executes the set-up procedure before training starts. The memory registered in the set-up phase is reused by later communication requests, and reclaimed after training is completed, making it possible to perform CUDA IPC once at the beginning.

B. Pipelined chunk transmissions

In each *transmission context*, the behavior of a GPU follows the determined behavior tuple as discussed in Sec. IV-C. The executor executes pipelined chunk transmissions.

Chunk pipeline on a server. Let A (with behavior tuple *<isActive=1, hasRecv=0, hasKernel=0, hasSend=1>*) and B (with behavior tuple *<isActive=1, hasRecv=1, hasKernel=1, hasSend=1>*) be a pair of sending and receiving GPUs on the same server, A first calls *cudaMemcpyPeerAsync()* to send one chunk, which is asynchronous with host; an event is then recorded into the same stream of data transfer. A continuously sends remaining chunks by repeating these steps.

On the receiver side, B calls *cudaStreamWaitEvent()* to wait for events recorded by A in the same order as they were recorded. Each event indicates that the chunk has been completely received by the *receive buffer* of B. If no other precedent nodes, B launches an aggregation kernel and the aggregated chunk is then sent to the successors. By pipelining chunks, we maximally mitigate the kernel launch overhead as it could be overlapped with NVLink transmission time.

Hidden memory movements. When GPU A and B are on different servers, the data transfer uses IB Verbs (for RDMA) or sockets (for TCP). The data sent from GPU A are first moved to the CPU memory and then to the NIC buffer through PCIe if GPU-Direct is not supported [20]. Reverse steps are carried out at GPU B. With data chunking, the memory movement could be hidden: the device-host memory copies of later chunks can be overlapped with network transfers of earlier chunks in a pipelining fashion.

Multi-stage parallelism. We enable pipelining across different stages in a multi-stage collective. For AllReduce, reduce and broadcast are pipelined, i.e., chunks aggregated on the root are immediately broadcast to other GPUs, which have *result buffers* to receive the aggregated data. For AlltoAll, a GPU simultaneously receives data from and sends data to others.

VI. Evaluation

A. Implementation.

We implement AdapCC with 5,000+ LoC in C++ and 2,000+ LoC in Python. The C++ backend relies on CUDA runtime and is wrapped into a Python module. To use AdapCC, users only need to add a few lines of code in the training script. First, import our Python module with *import adapcc* and call *adapcc.init()* to initialize the controller. This includes detecting topology, profiling links and generating strategies for a specific collective. To set up the communicator, call *adapcc.setup()* before the training starts. We expose various primitives as APIs (*allreduce()*, etc.) and also provide a communication hook for PyTorch DDP. For runtime profiling, we allow users to specify the profiling period by calling the *adapcc.profile()*. Profiling is executed in the C++ backend and does not terminate the host process nor involve any model checkpointing.

B. Methodology

Testbed Set-up. Our testbed includes six servers: four are equipped each with four A100 GPUs with NVLinks, two AMD EPYC-7H12 CPUs, PCIe 4.0, 256GB RAM, one Mellanox 100Gbps NIC; two are equipped each with four V100 GPUs with NVLinks, two Intel 6230 CPUs, PCIe 3.0, 256GB RAM, one Mellanox 50Gbps NIC. All servers run Ubuntu 20.04 LTS and are installed with Nvidia Driver 520.61.05, CUDA 11.3 and OpenMPI 4.1.1 for launching multi-process programs.

Baselines. We compare AdapCC with three representative communication systems: (1) *NCCL*-v2.14 [2], a state-of-the-art GPU collective library that builds ring/tree graphs based on a backtracking algorithm, which injects flows into the topology to saturate links with empirically labeled throughput. NCCL divides data into small chunks for pipelining. (2) *MSCCL* [29], which is a communication platform built on top of NCCL that executes customized algorithms. TACCL [15] system is not fully open-sourced and currently only provides template communication strategies for specific server configurations (i.e., DGX2 and NDv2). Since it relies on MSCCL as its run-time tool for data transmissions, we use pareto-optimal algorithms [17] officially recommended by MSCCL, which searches through different latency-bandwidth tradeoffs, to run multi-server communication. (3) *Blink* [14], which uses spanning trees for intra-server communication and NCCL primitives for inter-server communication. BLink sets chunk size empirically (i.e., 8MB). Since it is not open-sourced, we implement a prototype of Blink with CUDA and NCCL-v2.14.

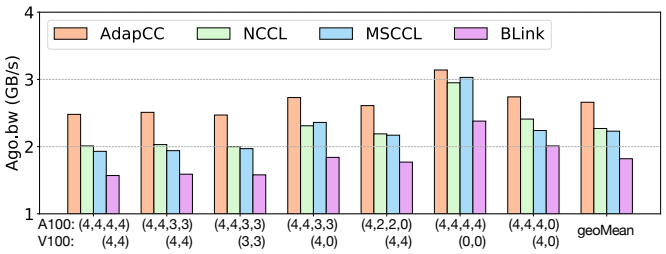


Figure 11: Performance comparison of Reduce.

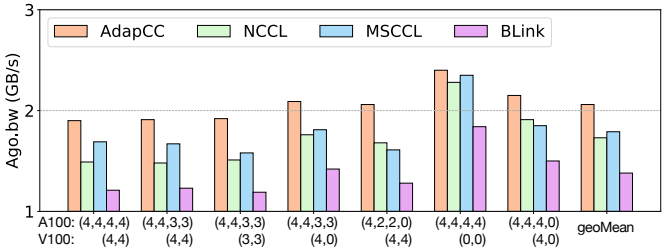


Figure 12: Performance comparison of AllReduce.

C. Benchmarking Performance without Relay

To evaluate the synthesized communication strategies without the relay control, we first benchmark AdapCC on three most frequently used collective primitives, Reduce, AllReduce and AlltoAll, in terms of the *algorithm bandwidth* (abbreviated as *Algo.bw*). The algorithm bandwidth is derived by running each primitive with an input float data of 256 MB (similar performance is observed in various data sizes) and dividing the data size by the time taken to complete the communication. We set the number of sub-collectives in each collective as $M = 4$ when producing the communication strategy. The performance impact of M is discussed and evaluated in Sec. VI-E.

Reduce. In Fig. 11, the x axis indicates different cases of GPUs involved in the communication. For example, 'A100: (4,4,4,4) V100:(4,4)' specifies using 4 GPUs on each of the A100 servers and 4 GPUs on each of the V100 servers in executing the collective. AdapCC outperforms all baselines, and achieves 1.06×-1.23× (1.17× in geometric mean, which is averaged among all cases) speed-up as compared to NCCL, 1.03×-1.29× (1.19× in geometric mean) to MSCCL, and 1.32×-1.58× (1.46× in geometric mean) to Blink.

We observe that NCCL fails to fully utilize heterogeneous links and devices. A binary tree is used for inter-server transmission, which assumes each node homogeneous and causes the one with less network capacity to become the bottleneck. Within each server, only one communication channel is launched to reduce data onto the GPU closest to an NIC, which cannot fully utilize all NVLinks. The communication strategies employed by MSCCL are designed for architectures similar to DGX1, without taking into account the actual properties of the underlying links. In the provided sketches, the chunk size also remains fixed, which does not effectively optimize the tradeoff between chunk pipelining and reduced latency

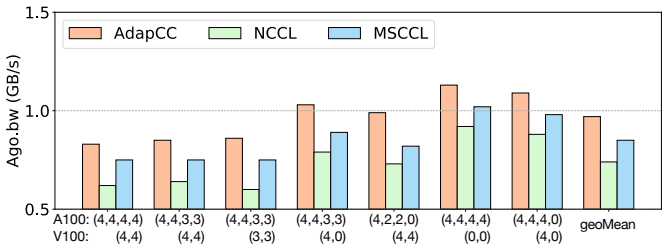


Figure 13: Performance comparison of AlltoAll.

in practice. Blink's performance is the lowest in multi-server communication scenarios as it is primarily optimized for intra-server communication, relying on NCCL operations for inter-server aggregation. The two stages of intra- and inter-server communications are not effectively pipelined.

AdapCC generates communication strategies for different resource configurations by leveraging profiling results. It synthesizes the communication graph to prevent any single node from becoming a bottleneck, identifies optimized chunk size to pipeline the transmission, and utilizes multiple parallel sub-collectives to achieve higher throughput. Even in the homogeneous case with 16 A100 GPUs, AdapCC demonstrates improvement as compared to baselines.

AllReduce. As shown in Fig. 12, AdapCC achieves 1.05×-1.29× (1.19× in geometric mean) speed-up as compared to NCCL, 1.02×-1.21× (1.15× in geometric mean) to MSCCL, and 1.30×-1.61× (1.49× in geometric mean) to Blink, due to better parallelization between reduce and broadcast stages in AllReduce and the awareness of link properties.

AlltoAll. NCCL does not natively support AlltoAll primitive, and we implement it using multiple *ncclSend* and *ncclRecv* operations. We did not compare with Blink, as it does not support AlltoAll in the multi-server case. Fig. 13 shows that AdapCC achieves on average 31% and 14% better algorithm bandwidth as compared to NCCL and MSCCL, respectively.

D. Training Performance

We next present end-to-end training results with adaptive relay control enabled.

Workloads. We train four representative models with PyTorch implementation: (1) VGG16 (528MB), that uses ImageNet as the training dataset [30]. (2) GPT2 (475MB), using the personal chat dataset [31]. (3) ViT, aka Vision Transformer (208MB) [32], on ImageNet dataset. Official training scripts are modified to use AdapCC as the communication backend. (4) MOE (512MB), based on the fastMoE framework [33] and dummy input data for simplicity, with each GPU worker running one expert with two linear layers. We call *adapcc.alltoall()* to replace the default implementation with NCCL P2P in fastMoE. By default, the local batch size per GPU for GPT2 is 16, and 128 for other models.

Stable Environment. We first run model training in a stable scenario, i.e., the network status is not volatile and there are no co-located interference workloads. In Fig. 14, data-parallel training is run on four A100 servers in the homogeneous setting (ref as 'Homo'), and on two A100 servers and two V100 servers in the heterogeneous setting (ref as 'Heter'). The

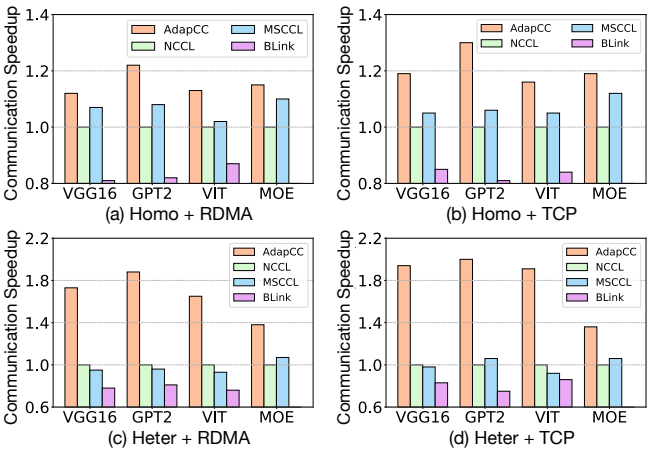


Figure 14: Communication speed-up as compared to NCCL.

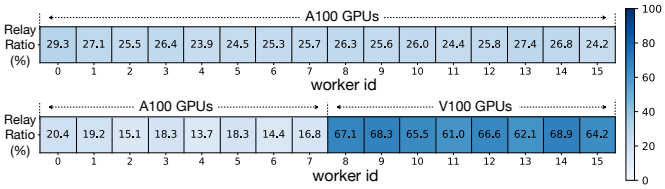


Figure 15: Probabilities of workers being chosen as relays.

two settings are compared under both RDMA and TCP network scenarios. The communication time measured includes the waiting time of faster workers and the actual execution time of AllReduce primitive for VGG16, GPT2 and ViT, and the communication time is the AlltoAll time for MoE model.

AdapCC achieves 1.12×-1.30× speed-up in homogeneous cases, by synthesizing optimized communication strategies, leveraging the waiting time of workers to perform timely partial communication and controlling GPUs as relays to expedite data transmissions. Our design brings more benefits in heterogeneous cases, where AdapCC achieves up to 2× speed-up. In addition, NCCL only launches one channel for inter-server transmission, which fails to saturate the available bandwidth. Especially for TCP, we found that the peak bandwidth achieved by a single channel is around 20Gbps (due to kernel space overhead), which is far less than the 100Gbps capacity. AdapCC uses parallel sub-collectives for better throughput. We further calculate the probability of a worker being chosen as the relay during training iterations, as in Fig. 15. In the heterogeneous case, GPUs with lower computing capacity have a higher probability of being selected; while in the homogeneous case, the distribution is more evenly spread.

In Fig. 16 and Fig. 17, we evaluate the impact of different training batch sizes (which affect computation time variance) on the training throughput, computed by $\frac{\text{global batch size}}{\text{iteration time}}$. AdapCC brings up to 31% and 20% training throughput improvement over NCCL, for GPT2 and VIT, respectively. With a larger batch size, the variance in computation time among workers also increases. In such cases, our adaptive relay control offers more advantages compared to simply waiting for all workers to complete computation.

Volatile Network. To assess the adaptability of AdapCC in adjusting communication strategies based on network status,

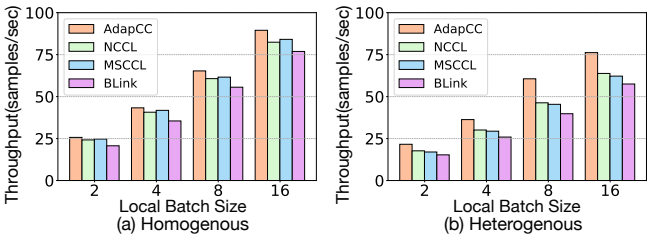


Figure 16: Training throughput of GPT2 (RDMA network).

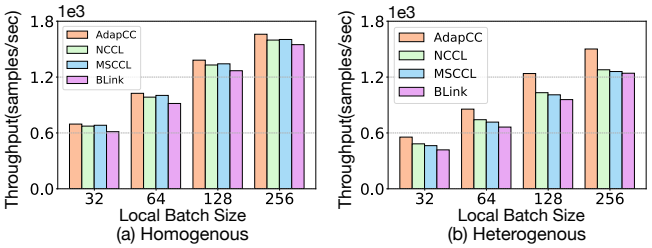


Figure 17: Training throughput of ViT (RDMA network).

we conduct experiments in a cloud scenario, where the network status can vary due to cross-traffic within the data center or bandwidth contention from co-located jobs. We train models on four homogeneous A100 servers with RDMA and use the tc tool on each server to manipulate the available bandwidth over time, following the trace collected from a public cloud (Sec. II-B). To offer more volatile cases, we amplify the bandwidth changes from the traces by a factor of x. If the bandwidth on a link drops or increases, it decreases or increases to $1-x$ or $1+x$ times the values observed in the trace data, respectively. Each model is trained for 10^4 iterations, and we measure the makespan, which represents the time taken to complete these training iterations. Profiling period is set to 500 iterations. In Fig. 18(a), we observe that AdapCC achieves greater makespan reduction compared to NCCL when network performance is more unstable. This improvement is attributed to AdapCC's ability to react to network variance and adjust communication strategies accordingly.

Online Serving Interference. Some shared clusters in a company have hybrid deployments [34], a multi-GPU training job could be co-located with some online CPU serving workloads. Those online tasks will cause performance interference with training workers on GPUs, due to resource contention of CPU cache and memory bandwidth. To verify the effectiveness of AdapCC in this scenario, we use four homogeneous A100 servers with RDMA to train each model. During model training, we randomly choose $0-2$ GPUs on each server every 5 minutes to launch online tasks (which conduct model inferences) on their affinity CPU socket. The CPU utilization of each online task is denoted as CPU interference level, ranging from 0% to 400% for observations. As shown in Fig. 18(b), when the interference level grows, computations of the chosen GPU workers are more likely to be slowed down. Benefited from the adaptive relay control, AdapCC achieves up to 1.49× faster communication as compared to NCCL.

E. Micro-Benchmark

Parallelization Degree. To investigate effect of the hyper-parameter M, which denotes the number of parallel sub-

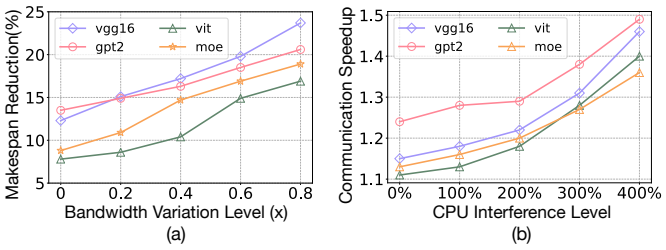


Figure 18: AdapCC's adaptivity in cloud scenario.

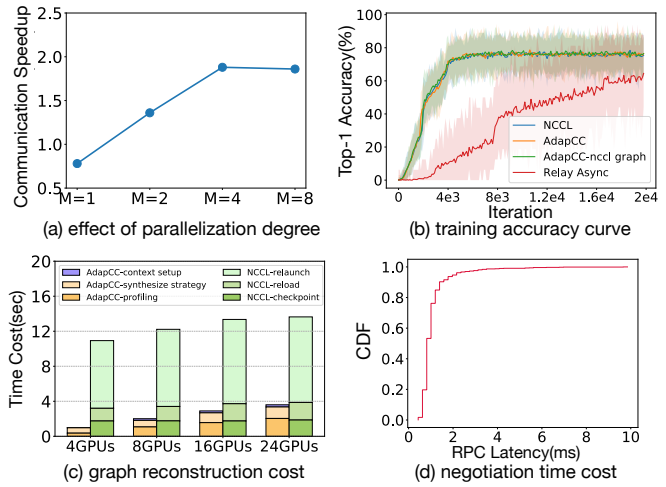


Figure 19: Micro-benchmarks of AdapCC.

collectives in one transmission, we evaluate the communication speed-up over NCCL when training VGG16 under different values of M, with testbed servers. As depicted in Fig. 19(a), increasing the number of parallel transmissions enables better utilization of available bandwidth. We choose $M = 4$ for our testbed as it offers a promising communication speedup while also minimizing GPU resource consumption.

Model Accuracy. We present the top-1 accuracy curve obtained while training VGG16 with testbed servers on a down-scaled Imagenet dataset consisting of $100,000$ images. As in Fig. 19(b), AdapCC achieves consistent accuracy as NCCL but could provide larger training throughput in various cases as validated before. By incorporating the adaptive relay control, AdapCC enables partial communication among ready workers first, and then proceeds with the remaining communication for relay workers. This approach ensures that the aggregated results remain consistent with a normal collective communication process. In contrast, one can simply discard the tensors of relay workers (denoted as 'Relay Async'), which indeed leads to higher throughput but also adversely impacts convergence. Besides, 'AdapCC-nccl graph' implies using the graph dumped from NCCL for communication, and the findings also support the notion that altering the tensor aggregation order, which is caused by a different communication graph [21], has minimal impact on training convergence.

Graph Reconstruction Overhead. We measure the graph reconstruction cost of AdapCC at different scales as in Fig. 19(c). For NCCL, when a user observes a non-optimal communication graph and wants to reconstruct it, the job needs to be terminated, and the whole process group and the model

should be rebuilt, thus introducing extra checkpointing and relaunching overhead. AdapCC does not need to terminate the job; the graph reconstruction undergoes profiling, solving the optimization problem and setting up the transmission context. 74%-91% time is saved as compared to NCCL. Additionally, the time for inferring the topology only occurs during the initial resource allocation phase of the job and remains constant as the job scale increases. In our measurements, this process takes 1.2s and is executed concurrently on each server.

For fault recovery, AdapCC undergoes graph reconstruction and data loader redistribution. This enables non-elastic ML frameworks to continue training without hanging. Compared to elastic frameworks like PyTorch Elastic, which by default takes 15s to detect faults and requires restarting the job [19], AdapCC ensures more efficient continuation of training.

RPC Delay in Relay Control. We measure the RPC latency for exchanging the relay information between workers and the coordinator. Fig. 19(d) shows the CDF of latencies collected on workers over 1,000 training iterations of VGG16 using 6 servers. The negotiation latency for 90% of the data points is smaller than 1.5ms. As compared to the multi-server communication time, this overhead is negligible.

VII. Conclusion

We present AdapCC, an adaptive communication library designed to accelerate distributed training. AdapCC includes a detection and profiling module that efficiently generates communication strategies based on observed performance during training. Additionally, AdapCC incorporates an adaptive relay control mechanism that reduces waiting time for collective communications. We evaluate AdapCC on multiple DNNs with various training settings on our GPU testbed, and demonstrate that it achieves significant improvements in communication speed and training throughput compared to state-of-the-art communication libraries.

References

[1] OpenAI, "Gpt-4 technical report," *arXiv preprint:2303.08774*, 2023.

[2] "Nccl," https://developer.nvidia.com/nccl, 2024.

[3] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," *arXiv preprint arXiv:2006.16668*, 2020.

[4] J. Yuan, X. Li, C. Cheng, J. Liu, R. Guo, S. Cai, C. Yao, F. Yang, X. Yi, C. Wu *et al.*, "Oneflow: Redesign the distributed deep learning framework from scratch," *arXiv preprint arXiv:2110.15032*, 2021.

[5] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.

[6] "Gloo," https://github.com/facebookincubator/gloo, 2024.

[7] "Mpi," https://github.com/open-mpi/ompi, 2024.

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

[9] H. Mikami, H. Suganuma, Y. Tanaka, Y. Kageyama *et al.*, "Massively distributed sgd: Imagenet/resnet-50 training in a flash," *arXiv preprint arXiv:1811.05233*, 2018.

[10] M. Cho, U. Finkler, D. Kung, and H. Hunter, "Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 241–251, 2019.

[11] H. Zhao and J. Canny, "Butterfly mixing: Accelerating incremental-update algorithms on clusters," in *Proceedings of the 2013 SIAM International Conference on Data Mining*. SIAM, 2013, pp. 785–793.

[12] J. Romero, J. Yin, N. Laanait, B. Xie, M. T. Young, S. Treichler, V. Starchenko, A. Borisevich, A. Sergeev, and M. Matheson, "Accelerating collective communication in data parallel training across deep learning frameworks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 1027–1040.

[13] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-aware cluster scheduling policies for deep learning workloads," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, 2020, pp. 481–498.

[14] G. Wang, S. Venkataraman, A. Phanishayee, N. Devanur, J. Thelin, and I. Stoica, "Blink: Fast and generic collectives for distributed ml," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 172–186, 2020.

[15] 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," *arXiv preprint arXiv:2111.04867*, 2021.

[16] L. Luo, P. West, A. Krishnamurthy, L. Ceze, and J. Nelson, "Plink: Discovering and exploiting datacenter network locality for efficient cloud-based distributed training," *MLSys 2020*, 2020.

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

[18] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, "Taming performance variability," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 409–425.

[19] "Pytorch," https://pytorch.org/docs/stable/distributed.elastic.html, 2024.

[20] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, 2020, pp. 463–479.

[21] M. Li, W. Xiao, B. Sun, H. Zhao, H. Yang, S. Ren, Z. Luan, X. Jia, Y. Liu, Y. Li *et al.*, "Easyscale: Accuracy-consistent elastic training for deep learning," *arXiv preprint arXiv:2208.14228*, 2022.

[22] J. H. Park, G. Yun, M. Y. Chang, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y.-r. Choi, "{HetPipe}: Enabling large {DNN} training on (whimpy) heterogeneous {GPU} clusters through integration of pipelined model parallelism and data parallelism," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 307–321.

[23] "Nccl-issue," https://github.com/NVIDIA/nccl/issues/670, 2022.

[24] Z. Zhang, C. Wu, and Z. Li, "Near-optimal topology-adaptive parameter synchronization in distributed dnn training," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[25] B. Wu, W. Bao, D. Yuan, and B. Zhou, "Competitive analysis for multi-commodity ski-rental problem," *ratio*, vol. 10, no. 10, p. 10.

[26] L. Pan, J. Liu, J. Yuan, R. Zhang, P. Li, and Z. Xiao, "Occl: a deadlock-free library for gpu collective communication," *arXiv preprint arXiv:2303.06324*, 2023.

[27] J. Dong, S. Wang, F. Feng, Z. Cao, H. Pan, L. Tang, P. Li, H. Li, Q. Ran, Y. Guo *et al.*, "Accl: Architecting highly scalable distributed training systems with highly efficient collective communication library," *IEEE Micro*, vol. 41, no. 5, pp. 85–92, 2021.

[28] L. Gurobi Optimization, "Gurobi optimizer reference manual," 2021.

[29] "Msccl," https://github.com/microsoft/msccl-tools, 2021.

[30] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[31] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[32] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[33] J. He, J. Qiu, A. Zeng, Z. Yang, J. Zhai, and J. Tang, "Fastmoe: A fast mixture-of-expert training system," *arXiv preprint arXiv:2103.13262*, 2021.

[34] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "Antman: Dynamic scaling on gpu clusters for deep learning." in *OSDI*, 2020, pp. 533–548.