MCCS: A Service-based Approach to Collective Communication for Multi-Tenant Cloud

Yongji Wu[1] Yechen Xu[1] Jingrong Chen[1]
Zhaodong Wang[2] Ying Zhang[2] Matthew Lentz[1] Danyang Zhuo[1]
[1]Duke University [2]Meta

Abstract

Performance of collective communication is critical for distributed systems. Using libraries to implement collective communication algorithms is not a good fit for a multi-tenant cloud environment because the tenant is not aware of the underlying physical network configuration or how other tenants use the shared cloud network—this lack of information prevents the library from selecting an optimal algorithm. In this paper, we explore a new approach for collective communication that more tightly integrates the implementation with the cloud network instead of the applications. We introduce MCCS, or Managed Collective Communication as a Service, which exposes traditional collective communication abstractions to applications while providing control and flexibility to the cloud provider for their implementations. Realizing MCCS involves overcoming several key challenges to integrate collective communication as part of the cloud network, including memory management of tenant GPU buffers, synchronizing changes to collective communication strategies, and supporting policies that involve cross-layer traffic optimization. Our evaluations show that MCCS improves tenant collective communication performance by up to 2.4× compared to one of the state-of-the-art collective communication libraries (NCCL), while adding more management features including dynamic algorithm adjustment, quality of service, and network-aware traffic engineering.

CCS Concepts

• **Networks** → *Data center networks*; • **Computing methodologies** → *Machine learning*; • **Computer systems organization** → **Cloud computing**.

Keywords

Collective communication, cloud computing, distributed training

ACM Reference Format:
Yongji Wu, Yechen, Xu, Jingrong Chen, Zhaodong Wang, Ying Zhang, Matthew Lentz, Danyang Zhuo. 2024. MCCS: A Service-based Approach to Collective Communication for Multi-Tenant Cloud. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3651890.3672252

ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0614-1/24/08
https://doi.org/10.1145/3651890.3672252

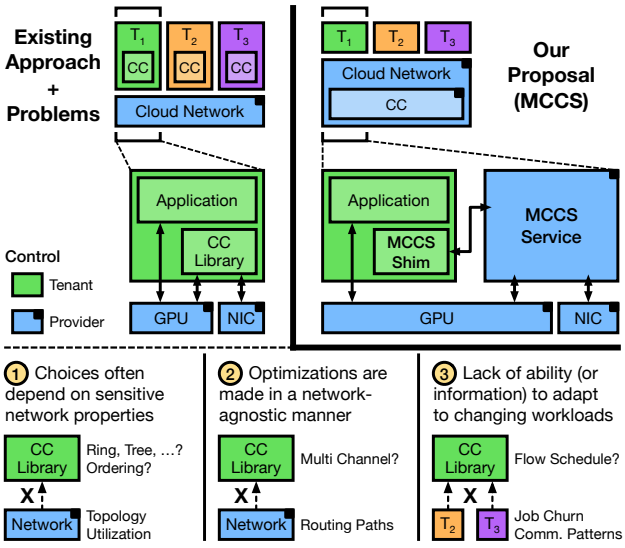


Figure 1: Comparison between existing approaches and our approach (MCCS) to collective communication.

1 Introduction

Collective communication is fundamental to supporting many distributed computing workloads. Many popular collective communication libraries exist today, including the NVIDIA Collective Communication Library (NCCL) [24], Intel MPI library [15], Open-MPI [12], and Gloo [11]. These libraries provide common collective communication primitives such as AllReduce and AllGather, which developers leverage by linking these libraries directly into their applications. Beneath their high-level APIs, these libraries implement primitives through various algorithms (e.g., AllReduce via Rings or Trees) along with heavily-optimized, and sometimes hardware-specific implementations (e.g., NVIDIA SHARP [26]).

Today, distributed workloads have increasingly moved to the cloud for the ease of infrastructure management and resource pooling. In a public cloud environment, however, existing collective communication libraries have shown several shortcomings. First, choosing the most efficient algorithm requires knowledge of the physical network topology and link utilization, which are not available to cloud tenants. As a consequence, this may lead to sub-optimal decisions between ring- and tree-based algorithms and their configurations (e.g., participant ordering in a ring). Second, current libraries (e.g., NCCL) decide the exact strategy at initialization time and will not change the chosen strategy once the job

starts. While this is fine for single-tenant settings (e.g., supercomputers), it is not ideal for multi-tenant settings because the best choice of algorithm depends on the other tenants' communication patterns. Finally, current libraries often make optimization choices in a manner that is agnostic to the underlying physical network configuration; however, these optimizations frequently rely on assumptions regarding the configuration. For instance, NCCL instantiates multiple TCP/RDMA connections between nodes to improve throughput by exploiting multiple network paths in parallel even though the connections may be routed via the same (shared) physical path.

In this paper, we propose a new approach that more tightly integrates collective communication with the cloud network instead of the applications. We call our approach MCCS, which is short for Managed Collective Communication as a Service. MCCS exposes traditional collective communication abstractions to applications, yet decouples the implementation from the applications themselves to extend more control to the cloud infrastructure provider. With MCCS, tenants are no longer responsible for implementing collectives, which often relied on information unavailable to the tenants themselves (e.g., physical configuration, properties of other tenants' applications). Meanwhile, MCCS extends significant flexibility to the cloud provider to support a variety of benefits: First, the provider can easily adopt custom, proprietary collective communication approaches without the need for changing existing user applications. Second, the provider can enforce fine-grained quality of service (QoS) policies at the level of collective operations. Third, the provider is no longer forced to choose between providing strong performance or the confidentiality of their proprietary infrastructure.

Achieving our goals for MCCS requires us to address several key challenges. First, we need to resolve the tension between decoupling collectives from the application while maintaining the existing interface. Second, collective communication is a group operation that requires synchronization among a number of components (e.g., application, service, hardware). Third, we need to support policies driven by the changing status of the cluster which can improve performance at both the logical-level (e.g., ring strategy) and physical-level (e.g., flow scheduling).

We implement a prototype for MCCS that targets applications using GPU-based computation and provide a lightweight shim library that connects applications with our system service. Our system can easily integrate existing collective algorithms implemented in NCCL through their CUDA kernels, as well as support more-customized algorithms. We evaluate the performance of MCCS using a small scale testbed and large scale simulations. Our testbed results have shown that MCCS consistently outperforms NCCL by up to 2.4x in terms of algorithm bandwidth, and improves training workloads in a multi-tenant improves by up to 34%. Our simulation results have demonstrated that MCCS enables an overall speedup of 3.4x on a large-scale cluster. Our source code is available at https://github.com/phoenix-dataplane/MCCS/.

In this paper, we make the following contributions:

- A new architecture for supporting collective communication in multi-tenant scenarios, shifting control from applications to the cloud network to improve performance.
- An approach to enable dynamic reconfiguration of collective implementations at runtime, which we leverage to demonstrate strong policies such as collaborative transfer scheduling across applications.
- A prototype implementation of MCCS targeting distributed machine learning workloads that is conceptually a drop-in replacement for NCCL, along with an evaluation that demonstrates the benefits via real-world traces on a testbed deployment and simulator.

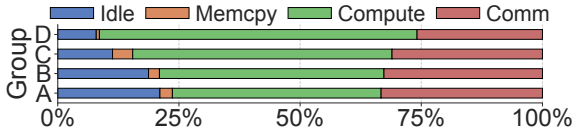


Figure 2: Training time breakdown of models from various product groups at a large social network company.

2 Background

We first describe how existing collective communication libraries work and their importance in deep learning. We then discuss why they are an ill fit for a multi-tenant network.

2.1 Collective Communication Libraries

Collective communication is a fundamental building block for parallel computing. Common collective operators include broadcast, reduce, allgather, reducescatter, and allreduce. To complete a collective operation, multiple participants perform a sequence of send and receive actions. The efficiency of these operations largely relies on algorithms that reduce network transmission. Typical collective communication libraries include NCCL [24], OpenMPI [12], and Gloo [11].

How does a collective communication library picks a strategy for a distributed job? A collective communication library has several built-in algorithms, and the library contains logic to select one of them based on a set of static factors like data length and the number of participants. Taking AllReduce as an example, OpenMPI uses several criteria to choose the most suitable collective algorithm, including a combination of factors such as the size of the data, the number of processes involved, the network architecture, and the bandwidth and latency requirement of the algorithm [14, 32].

The performance of collective communication has recently received significant attention in the research community and industry, due to the rise of distributed deep learning. Much efforts have been place in designing algorithms to improve the performance of collective operations such as AllReduce and AllGather [7, 29, 36, 40, 41, 44], as they play critical role in deciding the end-to-end deep learning performance.

Figure 2 shows a statistics of contributions of to the overall training time of models across four major product groups at one of the largest social network company in the world. Exposed (non-overlapping) computation, CPU↔GPU memory copy, communication time, and GPU idle time are measured. This breakdown confirms that data communication constitutes a significant portion of the training time.

2.2 Using Collective Communication Libraries in a Multi-Tenant Network?

In a multi-tenant datacenter network, traditional collective communication libraries face several challenges. Firstly, cloud networks often provide a simplified, black-box abstraction, where all tenant instances are connected through a big, virtual switch. However, these instances may actually be distributed across multiple physical racks, which are interconnected to upper-layer switches through multiple links, sometimes with oversubscription.

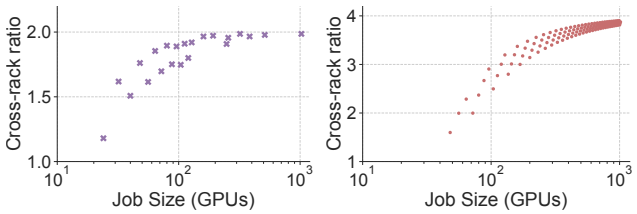
This lack of visibility into the physical topology can lead to suboptimal collective algorithm selection or configuration. For instance, in a ring-based collective algorithms where data transfer follows worker ranks (which are assigned by users), Without topology information, randomly assign ranks to workers in different racks could lead to the ring to cross racks back and forth multiple times, causing substantially more inter-rack traffic than necessary.

Figure 3 illustrates the network overhead introduced by non-optimal ring configuration with respect to job sizes. We have a production trace collected at one of the largest social network company, whose production cluster uses a spine-leaf architecture. Each rack connects two hosts, each with 8 GPUs and 8 NICs. We measure a job's network overhead using cross-rack ratio, where is the number of cross-rack flows of the collective ring used by the job, normalized to that of an optimal ring configuration. A ring configuration in the worst case introduces 2x cross-rack flows compared to the optimal one. The performance degradation would only grow as more hosts are placed under a rack. We simulate a cluster with the same scale as the company's and computes the expected cross-rack ratios with different job sizes, if ring ordering is randomly chosen and we assume jobs are perfectly packed to hosts. The worst case overhead becomes 4x in this scenario. We also find that the overhead grows with respect to the job size.

Further, in today's multi-path datacenters, the most commonly used network load balancing strategy is Equal-Cost Multi-Path (ECMP). However, ECMP may not efficiently handle multiple flows from a ring-based collective operation, potentially leading to congestion on a single physical path and reduced throughput.

NetHint [8] suggests letting a cloud provider expose its network topology and link utilization. This transparency will potentially enable a collective communication library adjust its choices of collective communication at runtime. However, this approach raises security and privacy concerns, because a cloud provider has incentive to maintain the confidentiality of its network topology and link utilization.

In summary, cloud tenants face a significant challenge in selecting an optimal collective communication algorithm due to the lack of visibility into the physical network's structure. Providing tenants with access to this information could introduce security risks for cloud providers, as it exposes sensitive details of their infrastructure. *A viable approach appears to be for the cloud provider to assume responsibility for choosing the collective communication strategy on behalf of the tenant.* Our system, MCCS, explores this approach.



(a) [Empirical] 2 hosts/rack **(b) [Simulated] 4 hosts/rack**

Figure 3: Number of cross rack flows normalized to optimal ring from both production trace and simulation.

3 Overview

MCCS is a new design of collective communication for the multi-tenant cloud setting. We have following goals. First, a cloud provider can decide the collective communication strategy for a cloud tenants. The cloud tenant calls a higher-level collective communication interface like AllReduce, instead of instantiating point-to-point data transfer by a collective communication library. Second, the tenant has no knowledge of what algorithm is chosen and does not know the cloud's network topology, link utilization, etc. The cloud provider hides all these sensitive information inside a cloud service. Third, the cloud provider can change the collective communication strategy without interrupting the running tenant to accommodate changes in the multi-tenant network (e.g., to accommodate a new tenant's workload, to leverage more available bandwidth as other tenants leave the network). Finally, our new design will enable the cloud provider to do other optimizations (e.g., joint optimization of collective communication and flow scheduling, having multiple tenants' workloads use the bandwidth in an interleaved fashion).

MCCS, realizes collective communication as a cloud service instead of an application library. At the same time, MCCS maintains a similar interface as traditional collective communication libraries such as NCCL. Figure 1 presents an overview of the various components of MCCS and its architecture different compared to traditional collective communication libraries. MCCS service runs as a trusted, user-space process with access to all GPUs and NICs on the host. User applications on the host only have access to one or more GPUs allocated to the application. Applications are compiled with MCCS shim library, which communicates with MCCS service using shared host and GPU memory. MCCS service is controlled by the cloud provider, and applications can only access it through its collective communication APIs.

MCCS service issues collective communication operations for GPU buffers on behalf of the cloud tenant. MCCS service decides the collective communication strategy and can change the strategy at runtime. Because MCCS service is controlled by a cloud provider, for every flow in the collective communication, MCCS service can also decide its network path in the cloud network using source routing or other path control schemes. Further, MCCS service can enforce QoS policies by controlling flow paths and the timing of collective communication operation execution.

4 Design

In this section, we break down and discuss the design of MCCS in three parts. First, we look at how MCCS meets the existing collective API while decoupling collectives from the application. Next,

we discuss how MCCS supports the data path for collective communication with support for runtime reconfigurability. Finally, we present our solution to decouple policy from mechanism and enable flexible management of collectives. While our design is agnostic to any particular collective communication implementation, we focus on the NCCL throughout this section; other GPU-based collective communication libraries such as RCCL [38] and oneCCL [28] maintain similar semantics and terminologies as NCCL.

4.1 Collective Interface

To support collective communication as a service, MCCS needs to: 1) provide an interface to applications for invoking collectives, and 2) enable synchronization between application computation and collective operations. One goal that constrains our design is a desire to maintain as close to the same interface as existing libraries, like NCCL, so that MCCS could act as a drop-in replacement.

NCCL provides APIs that build on core CUDA primitives for applications to issue collectives. A CUDA stream is similar to the notion of a thread, allowing applications to enqueue a sequence of operations (e.g., kernels) to be executed in-order by the GPU. An application can use multiple threads to express concurrency between operations by enqueuing them on different streams. When invoking a NCCL collective API, developers specify the stream that the collective will be enqueued on; this is used to capture the data dependency between a collective and the (prior) computations that generate the data for the collective to operate over.

To realize our design for MCCS, we need to address two challenges that relate to the inherent isolation between applications that leverage CUDA. First, similar to host memory, GPU memory between different processes is isolated by default. How can MCCS service GPU buffers of applications and implement collectives on them? Second, due to the fact that CUDA streams are limited within a single process, how can MCCS service's APIs maintain the same thread semantics for the application's CUDA calls, as respected by NCCL? We will discuss each of these in turn.

Memory Management. We address the memory management challenge by choosing to redirect control over GPU memory allocations and deallocations to the MCCS service. Our shim library provides APIs that applications can invoke directly to allocate memory that will be accessible to both the application and the MCCS service; the application can use the existing CUDA APIs to manage private buffers that are not directly used as part of collective operations. Alternatively, to minimize the changes to existing applications, we also support the redirection of all allocations to the MCCS service.

The MCCS shim issues an allocation request to the MCCS service over the shared memory command queue between the application and the service. A dedicated front-end engine for the given application will handle the request by internally allocating memory on the specified GPU device and obtain an inter-process memory handle to share with the application. The MCCS shim receives and opens this handle to obtain the underlying device pointer to the allocated memory, which it returns to the application. The application can then use these pointers freely for invoking compute operations, while for collective operations, MCCS shim passes an identifier for the memory allocation and an offset to MCCS service. The service will check whether the data buffer user passes is within a valid allocation before performing the operation. This process follows similarly for deallocation requests – the shim is responsible for closing the inter-process memory handle before forwarding the request to the MCCS service.

Synchronization. We address the synchronization challenge by designing an event-based mechanism that maintains the traditional semantics of CUDA streams, which enables the sequential ordering of dependent compute and collective operations. Since CUDA streams belong to a single application and cannot be shared (unlike GPU memory via inter-process handles), we need an alternative approach for MCCS. We leverage the CUDA event primitive, which provides the ability to enqueue a stream operation that blocks waiting on the notification of a event that was enqueued on a (different) stream. Unlike streams, events can be shared via inter-process handles.

MCCS addresses synchronization at the level of communicators, a standard abstraction used in collective communication libraries (e.g., NCCL) to specify a subset of n nodes, each assigned a unique rank in $[0, n)$, to take part in collective operations. While an application may use an arbitrary set of streams for managing its computation and collective operations, the MCCS service maintains one stream per communicator. When an application issues a collective operation for a given communicator, the MCCS service enqueues the communication kernels that implement the collective request on the associated stream.

To enable communication kernels on MCCS-managed streams to wait for compute kernels on application-managed streams to finish (and vice-versa), we need to introduce event management operations into the MCCS shim. When a communicator is created, the MCCS service also creates a corresponding event object and obtains an inter-process event handle that it returns to the MCCS shim (along with the communicator handle). The MCCS shim will use this event to enqueue an operation on the application's stream to block waiting for the collective to finish prior to executing any subsequent operations. Likewise, the MCCS shim also creates an event object for each application stream to share with the MCCS service. Instead of hooking directly into CUDA's stream management API, the MCCS shim creates events in an on-demand fashion whenever a new application stream is used for invoking a collective operation. The MCCS shim shares an inter-process event handle along with a stream identifier with the MCCS service, which it uses to enqueue an operation on the internal stream for the communicator to block waiting for any computation to finish prior to executing the subsequent communication kernel.

Now, we have developed an approach for decoupling the collective communication from the application while still providing the same interface from our MCCS service. Next, we will explore how MCCS actually implements the communication that underlies collective operations while enabling more manageability as compared to existing approaches.

4.2 Collective Communication

To support dynamic reconfiguration of the data path subject to policies, we need to decouple the data path setup from the communicator's initialization. In NCCL, most of the work that configures

the data path takes place when a communicator is created by an application, where NCCL first attempts to detect the intra-host topology (e.g., NVLink between GPUs) to figure out how to connect all of the intra-host GPUs, while also identifying the best NIC to use. After the intra-host topology is decided, each rank communicates with the root (i.e., rank 0) to form an AllGather TCP/IP-based ring for exchanging control information. At this point, NCCL can set up the underlying algorithm for implementing collective operations by constructing rings and trees, which it uses to establish peer-to-peer connections between nodes (e.g., consecutive ranks in the ring).

NCCL is mainly designed to run on infrastructure that is tightly controlled by the user (i.e., application developer), focusing only on optimizing the intra-host strategy while leaving the optimization of inter-host strategy to users. In particular, NCCL simply connects inter-host rings and trees according to the ordering of user-specified ranks when establishing the communicator; therefore, users must carefully design the GPU-to-rank mapping, which requires expert knowledge of the cluster topology (and is often error prone). In multi-tenant settings for public cloud, where the users and infrastructure provider are not the same principal, this becomes problematic. However, even intra-host optimization presents issues in public cloud settings as well due to virtualization. NCCL can potentially fail to optimize intra-host strategy because it relies on `sysfs` information to discover the PCIe topology of GPUs and NICs. Typically, modern virtualization approaches may hide such information from tenants (and thus from tenant-controlled collective libraries like NCCL). This challenge has been noted in other recent work on collective communication algorithms such as TACCL [41].

By decoupling the implementation of collectives from the applications, we are uniquely positioned to transform these prior challenges into opportunities for MCCS. First, we can leverage proprietary (and thus often confidential) topology information within the context of the MCCS service without revealing such information to the applications. This involves architectural challenges in terms of running collective communication strategies outside the tenant application's control and observability. Second, we can enable dynamic reconfiguration of collective strategies in response to changes in the cloud network (or the set of multi-tenant applications). This involves addressing a key challenge for enabling reconfiguration (e.g., at the level of ring orderings) that simultaneously achieves high-performance and ensures synchronization across the participating nodes within a communicator. We will discuss each of these in turn.

Multi-Tenant and Topology-Aware Architecture. For MCCS, we need to develop a new architecture to simultaneously support multiple applications sharing cluster, or even individual host, resources while also being able to exploit low-level network information (e.g., physical topology). Given that NCCL is focused on a single application, the implementation of collective communication for communicators consists of a "transport agent", which is responsible for managing the sending/receiving of inter-host collective communication traffic via available NICs based on GPU data buffers. We choose to decompose the role of the MCCS service into two main engines [1]: 1) a proxy engine, and 2) a transport engine.

The proxy engine is responsible for bridging the gap between high-level communicators and low-level resources. For each GPU on a given host, MCCS initializes a single proxy engine that handles all communicators which include that GPU in their ranks. When a collective is issued, the proxy engine manages the higher-level collective strategies and network configurations for how the collective communication will be implemented. For instance, this enables MCCS to optimize how inter- and intra-host rings are connected and ordered for improve resource utilization. Additionally, MCCS enables the incorporation of various collective strategies optimized for specific topologies, such as those proposed in recent research [7, 29, 41] or even proprietary strategies developed in-house by the provider. In all cases where communication takes place over intra-host communication channels (e.g., NVLink, host shared memory buffer), the proxy engine manages the setup and use of those channels directly.

For all inter-host communication, the proxy engine offloads the management to the transport engine. While conceptually similar to the transport agent in NCCL, the transport engine in the MCCS service is responsible for multiple applications simultaneously. Additionally, the transport engine is responsible for providing the underlying mechanisms for scheduling flows on network paths using existing path control techniques (e.g., source routing, policy-based routing). There may be one or more transport engines associated with each GPU to support more communication parallelism.

Dynamic Reconfiguration. The MCCS service exposes support for dynamic reconfiguration via a command that is made available to the provider (not the applications). A key goal of our design is to ensure that the performance overhead for performing a reconfiguration is low (since this otherwise reduces the benefit from implementing smart policies) and that there is zero (or negligible) performance overhead for collective operations when no reconfiguration is issued. At a high-level, this motivates our choice to support reconfiguration at the granularity of collective operations. Reconfiguration should be a coarse-grained scheduling decision in practice, reacting to events such as link utilization increasing due to traffic that is outside the scope of collectives managed by MCCS (e.g., fetching training data, background flows).

While it is straightforward for all nodes to agree on a configuration at initialization time, which necessarily takes place before collectives, this is much more challenging when implementing reconfigurations between collective operations. For an illustration of this, consider the example shown in Figure 4. Here we assume an application created a communicator consisting of three GPUs and that it issues a series of AllReduce (AR) collectives for that communicator. We differentiate between the launch of a collective, which shows the ring configuration, and the subsequent completion of a collective. At some point, a reconfiguration request (Req) is sent to each of the MCCS service instances running on different nodes; however, due to arbitrary network and processing delays, it is possible for the command to be received and processed at different times. Without appropriate synchronization, this could lead to

[1]We use the term "engine" to refer to a general wrapper around functions that can asynchronously operate over inputs to generate some outputs.

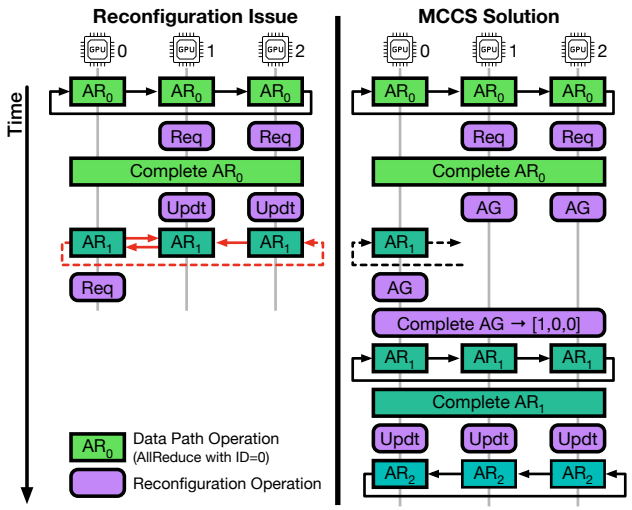


Figure 4: Example showcasing a potential synchronization issue in handling dynamic reconfigurations (left) and the MCCS protocol to address this (right).

correctness issues (shown on the left) in which rank 0 executes AR_1 from the perspective of the previous ring ordering, while ranks 1 and 2 perform updates (Updt) to handle the reconfiguration request prior to AR_1. We need to address this problem without requiring expensive synchronization operations on the fast path (i.e., between collectives when no reconfiguration takes place).

Our solution is to leverage the per-communicator control ring as the basis to construct an efficient barrier synchronization mechanism. Each proxy maintains a sequence number for the collectives over time, which inherently matches across all nodes in a communicator because each collective involves every node. After receiving a reconfiguration request, each proxy enqueues all subsequent collectives prior to issuing an AllGather (AG) collective on the control ring to exchange the sequence number corresponding to the last collective launched. Local updates will not be completed until the AllGather completes, which will provide all nodes with the sequence numbers for every other node; computing the maximum sequence number enables nodes to identify which collectives should precede any reconfiguration update (i.e., collective sequence number is less than or equal to the maximum).

Looking back at Figure 4, we see on the right how this synchronization prevents this correctness issue in this example. When the proxies for ranks 1 and 2 receive the reconfiguration request (Req), they issue the AllGather operation with their data containing 0 for the latest collective (AR_0) that was launched. Later, when the proxy for rank 0 receives the reconfiguration request, it also issues the AllGather operation; however, since it already launched AR_1, its data contains 1 for the latest collective. At this point, the AllGather operation completes, which allows the proxies for all ranks to determine that the maximum sequence number is 1; for ranks 2 and 3, this means that they should issue the queued AR_1 collective prior to updating the configuration. The updated ring ordering will be used in all future collectives until a another future reconfiguration request is issued. To update a configuration, the proxy engines will interact with the transport engines to close all existing peer-to-peer connections for the communicator and clean up corresponding resources. Afterwards, the new connections are instantiated based on the chosen strategy (e.g., rank ordering within a ring), similar to what is performed at the time of initialization.

In analyzing the performance implications of this design, we can make two observations. First, issuing a reconfiguration request can introduce some performance overhead, since collectives will be stalled until the AllGather for the reconfiguration is complete (i.e., until the last proxy receives and handles the request). Additionally, there is some overhead in tearing down and establishing new peer-to-peer connections. As we will demonstrate in the evaluation, the performance overhead for handling reconfigurations is rather small, and enables significant performance benefits from smart policies. Second, in the absence of a reconfiguration request, there is no performance overhead. Note that the proxy for rank 0 is able to launch the AR_1 collective before even being aware that the other ranks received a reconfiguration request – any synchronization via the control channel (or blocking) only occurs after a request is received.

At this point, we have the ability for applications to issue collectives and for the MCCS service to implement them while supporting reconfigurations at runtime while taking into account low-level topology information. Next, we will explore how MCCS enables flexible and expressive management according to provider-defined policies that build on top of the reconfiguration mechanisms that we just discussed.

4.3 Enabling Manageability

One of our key goals in MCCS is to cleanly decouple policy from mechanism. The design of our proxy and transport engines within the MCCS service enables management of both the control and data paths for collective communication. On the control path, the MCCS service can support different collective strategies for various applications as well as control network resource allocation (e.g., NICs per application, network routing). On the data path, the MCCS service can support fine-grained control of communication through augmentation of the transport engine to control the conditions for sending network traffic. Our architecture enables this through dynamic loading of provider-supplied logic that can handle policy decisions determined by an external controller.

To enable an external controller (e.g., centralized manager) to schedule the collective communication across all applications on the cluster, the MCCS service needs to provide an interface for exposing necessary information. For each application, this information is based on the set of active communicators, including the set of GPUs (and hosts) that make up the ranks within the communicator, and the current configuration of collective strategy (e.g., ring configuration) and network resources (e.g., flow mapping). Additionally, the MCCS service can perform fine-grained tracing of collectives issued by applications to determine properties of their computation and communication patterns. The controller consumes this data to make a policy decision.

Next, we look at several concrete examples of scheduling and quality-of-service (QoS) policies, which will also be used in our evaluation of MCCS. While these examples are admittedly straightforward, they effectively illustrate MCCS's system capabilities beyond what today's collective communication library can offer. MCCS can also incorporate topology-optimized collective algorithms from MSCCL [7, 41], while they only apply to a single-tenant environment.

Topology-aware collaborative scheduling. We explore the following two heuristics to enable the joint optimization of the algorithmic strategy at the collective level and flow assignment at the network layer.

Example #1: Locality-aware ring configuration. The ordering of how the hosts are chained in the ring collective algorithm directly dictates the overall communication pattern. If many flows have to go through links above the leaf level (assuming a Clos network topology), severe congestion could occur due to oversubscription. Hence, our goal is to minimize the number of cross-rack / cross-pod flows. We apply a greedy algorithm to configure the ring ordering for each communicator (application). We group the participant hosts by their locality (e.g., under the same rack, under the same pod) and then connect them in a sequential order. The algorithm takes the set of participant GPUs for each communicator obtained by MCCS service management APIs, and sends an optimized ring ordering back to MCCS service.

Example #2: Best-fit fair flow assignment (FFA). Once the ring configuration for all applications are optimized, the communication patterns between hosts and hence the set of flows can be determined. Still, using the standard ECMP approach to map flows into network routes could lead to significant overall collective performance degradation and inconsistency due to flow collision. Our goal is to maximize the aggregated collective performance of all applications, and ensure fairness between different applications. We use a slightly modified version of the greedy heuristics proposed in Hedera [1], where for each flow we assign it the path that has minimal excess bandwidth demand. We round-robin between flows from different jobs for fairness. For example, if two applications are both performing collectives using hosts on rack A and B. There are 2 routes between A and B and each application have 2 flows from A to B. FFA would assign each route a flow from both application. FFA takes the collective strategy configuration of all communicators as input. As communication patterns solely depend on the collective strategy, FFA knows all flows (RDMA connections) in the network. It then assigns each flow a route ID, where the mapping is issued to MCCS service.

QoS features. MCCS enables priority control at both coarse-grained resource allocation and fine-grained communication.

Example #3: Priority flow assignment (PFA). We modify FFA to allow some routes to be reserved for high priority applications. We first fit flows of low priority applications using only non-reserved routes, and flows of high priority applications are assigned best routes from all available ones. In our example, PFA can dedicate one of the two routes between rack A and B to the prioritized application.

Example #4: Traffic scheduling (TS). With priority flow assignment, we can dedicate networks links to some of the highest priority applications. However, we may still have some applications sharing links. MCCS could enforce a traffic schedule to control when each application can send out traffic. In our implementation, we apply a simple time window based approach inspired by CASSINI [35] to interleave traffic. TS invokes MCCS tracing API and requests a trace of a prioritized application. TS then analyzes the idle cycles of the application when it is not issuing collectives. TS sends a time interval schedule to MCCS service. Transport engines in MCCS service then allow other applications to send traffic only when the prioritized application is idle.

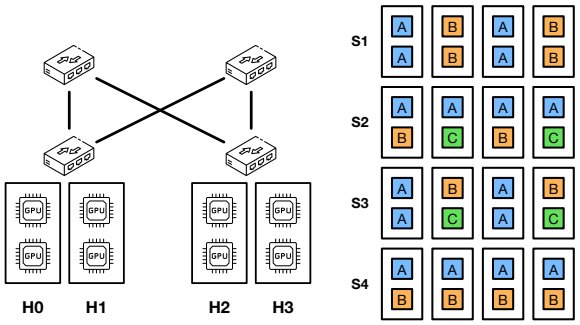
5 Implementation

MCCS is implemented in 13.5K lines of Rust: 1.5K for the shim library and IPC implementation, 6K for control and management planes, and 6K for transport engine and transport protocols.

Collective CUDA kernels and transports. We adapt CUDA kernels from NCCL v2.17.1 for computation and intra-host communication. We modify the kernels so that the communicator resources on the kernel side (e.g., ring buffers) can be set up by proxy engines in the MCCS service. We focus on ports of NCCL's ring AllReduce and AllGather kernels; however, it is straightforward to implement other collective operations, P2P communication, and other algorithms (e.g., tree algorithms). For transport protocols, we implement support for channels using host shared memory and RDMA; other channels, such as NVLink, can also be integrated.

Internal engine scheduling. Our engines are designed similar to asynchronous futures in Rust. A pool of runtimes is used to execute the engines, where each runtime corresponds to a kernel thread. Engines can be scheduled on either a dedicated runtime or a shared one. Runtimes without active engines can sleep to release the CPU. Currently, we dedicate a runtime to each engine. Compared with NCCL, which only uses an additional thread per GPU for the transport agent, our prototype would use 2 more threads for the frontend and proxy engines. However, we note that if multiple applications use the same GPU, they will share a proxy engine. We do not focus on CPU usage optimization as a core goal in our prototype, and we could implement better engine scheduling strategies to lower CPU utilization (e.g., frontend and proxy sharing a runtime if only one application uses the GPU).

Management. We leverage policy-based routing at the switch to achieve explicit route control for implementing FFA and PFA. Based on the assigned route ID for each RDMA connection, MCCS service modifies the UDP source port of ROCEv2 packets. The source port is not used by ROCEv2 protocol, hence we install a routing policy on switch that maps flows to routes based on the UDP source port specified by MCCS service. To implement TS, we currently use a hard-coded logic directly embedded in the transport engines, and we manually profile applications offline. We note that such TS scheduling logic could be easily integrated into a dynamic library function loaded by the transport engine, while the communication trace of applications can be retrieved from the MCCS management API.



(a) Testbed topology (b) Multi apps setups

Figure 5: Testbed topology and multiple applications evaluation setups.

For our evaluation, we consider the case in which all tenants utilize MCCS for collective communication. However, this is not strictly required. Even if only a subset of tenants use MCCS, MCCS can still collaboratively schedule the collectives of that subset, while treating other flows as background flows (and adapt to them).

6 Evaluation

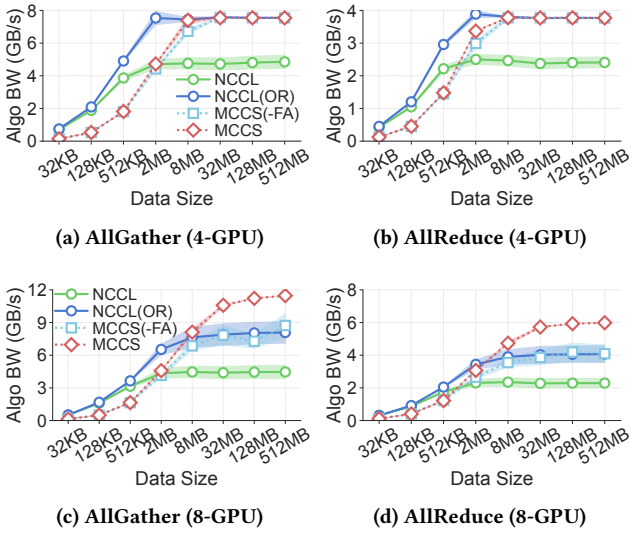
We evaluate the capabilities of MCCS using a small-scale testbed. We also conduct large-scale simulations to quantify the performance benefits of collaborative scheduling enabled by MCCS for large compute clusters.

6.1 Testbed Setup and Workloads

Figure 5a presents the setup of our testbed. We have four nodes in our testbed, each equipped with 2 NVIDIA RTX 3090 GPUs and a 100 Gbps Mellanox ConnectX-5 NIC. Using a single 100 Gbps Mellanox SN2100 switch, we emulate a spine-leaf topology with 2 leaf switches and 2 spine switches through self-wiring. Four nodes are placed under two racks, where each rack corresponds to a leaf switch. The links between the switches are limited to 50 Gbps, while the links between each host and the leaf switches are limited to 100 Gbps. This means that the over-subscription ratio of our testbed is 2. On each host, we use IB traffic class (TC) and rate limit each TC to emulate two 50 Gbps virtual NICs (one per GPU).

We use AllReduce and AllGather benchmarks to evaluate how MCCS can improve the collective performance in both the case of a single application and the case of serving multiple applications at the same time. For the single-application scenario, we use two setups: a 4-GPU setup where one GPU and one 50 Gbps NIC on each host is used, and an 8-GPU setup where all two GPUs and two 50 Gbps NICs are used. To show the effectiveness of MCCS in a multi-tenant environment, we construct 4 setups on our testbed. These setups include applications with different sizes and different placements, as shown in Figure 5b.

In addition to AllReduce and AllGather benchmarks, we evaluate training workloads using a traffic generator with profile traces. The traffic generator is implemented with Rust using the MCCS library. To collect the traces, we used PyTorch [30] v2.1.0, DeepSpeed [37] v0.10.3 and Megatron-LM [42] to profile a VGG-19 model [43] with data parallel training, and a 2.7B parameters GPT model [6] with tensor parallel training.



(a) AllGather (4-GPU) (b) AllReduce (4-GPU)

(c) AllGather (8-GPU) (d) AllReduce (8-GPU)

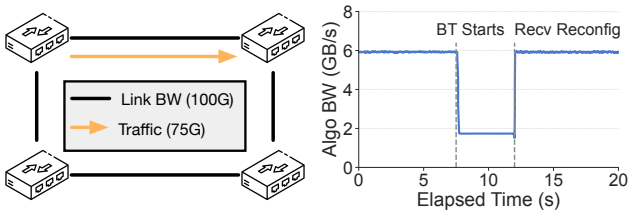
Figure 6: [Single application]: Algorithm bandwidth of AllReduce and AllGather. The shaded areas represent 95% percentile intervals.

Baselines: We compare MCCS with **NCCL** (v2.17.1), which is not network topology aware and cannot perform inter-host ring optimization. To quantify the performance overhead of MCCS, we manually configure the inter-host ring used by NCCL with the results from our locality-aware ring configuration algorithm to serve as one of the baselines. We denote this baseline as **NCCL(OR)**, i.e., NCCL with optimal ring.

6.2 Improving Single Application

We first evaluate how the performance of a single application can be improved with topology-aware scheduling capability enabled by MCCS. We run AllReduce and AllGather benchmarks of different data sizes (measured by output buffers). We report the algorithm bandwidth [25] measurement, which is calculated as output buffer size divided by execution time. To evaluate the system overhead introduced by MCCS, we also evaluate MCCS without our flow assignment algorithm, and instead rely on ECMP for routing. Figure 6 shows the results in the 4-GPU and 8-GPU setups. Our full solution is denoted as **MCCS**. We also evaluate a version of MCCS without doing flow assignment: **MCCS(-FA)**.

MCCS's system-level performance overheads can be calculated by comparing MCCS(-FA) and NCCL(OR), which both use the optimal ring from our ring configuration algorithm. MCCS has negligible system-level performance overheads when data size is above 8 MB. On 4 GPUs, MCCS(-FA)'s algorithm bandwidth is 63% lower than NCCL(OR) on 512 KB AllGather (which corresponds to 128 KB input per GPU) and 51% lower on 512 KB AllReduce, but the performance difference decreases to 9.7% for 8 MB AllGather and 0.75% for 8 MB AllReduce. The reason is that for large messages, MCCS's performance is bottlenecked by the collective communication's



(a) Example scenario (b) Ring reconfiguration

Figure 7: [Single application]: Showcase of adapting to background flows.

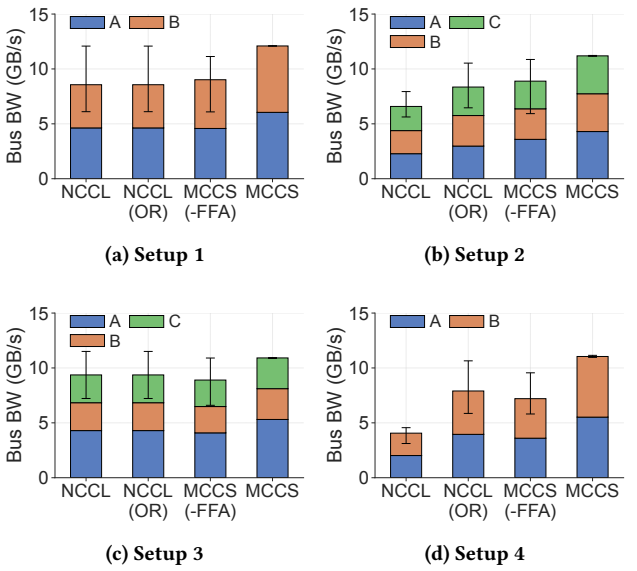
data transfer. For small messages (less than 8 MB), MCCS suffers from performance penalties due to the latency overhead introduced on the datapath. The communication between the application and the MCCS service, as well as between the internal engines of the MCCS service, incurs an overall latency of 50-80 us.

NCCL's performance is the worst because NCCL itself does not know the best ring configuration. Comparing NCCL and NCCL(OR), we find collective algorithm optimizations play a crucial role in achieving high performance. NCCL(OR) is 56% better than NCCL on the 4-GPU setup and 78% better on 8-GPU the setup for 512 MB AllReduce.

To understand how ECMP plays a role in AllReduce and All Gather performance, let's look at the 8-GPU case in Figure 6 because the 8-GPU scenario has cross-rack traffic. For 512 M AllReduce, MCCS outperforms MCCS(-FA) and NCCL(OR) by 46%. Note that all three approaches use optimal ring configurations. The key reason is that flow assignment is fundamental to avoiding flow collision in ECMP, so only optimizing the collective algorithm insufficient.

MCCS enables joint optimization of collective communication algorithm and flow scheduling by a cloud provider. Combining both ring configuration and flow scheduling techniques, MCCS delivers an 1.6x speed-up on the 4-GPU setup and a 2.4x speed-up on the 8-GPU setup on average for 8 MB-512 MB AllReduce and AllGather compared to NCCL.

Dynamic changes of collective communication strategies to adapt to background flows. Here we also showcase the capability to reconfigure an application's collective strategy at runtime without interrupting the application. We use an example scenario to demonstrate this feature. We leave the monitoring of background flows to external components. For instance, a switch agent can be configured to report to a centralized manager when there are persistent large flows that are not managed by MCCS. The centralized manager can then send a new configuration to MCCS service. With our testbed, we emulate a topology shown in Figure 7a, where each of the server is connected to a switch, and the four switches are linked as a ring. We instantiate an 8-GPU AllReduce job, the the AllReduce job uses a ring algorithm that connects hosts clockwise. As shown in Figure 7b, at time 7.5 s, a background flow of 75 Gbps between two switches in the clockwise direction, the available capacity for the AllReduce job drops to 25 Gbps. However, the switch links counterclockwise is not affected. If the collective strategy configuration is not adjusted, the AllReduce algorithm bandwidth drops from 5.9 GB/s to 1.7 GB/s. MCCS enables the application to recover its collective performance by transparently reverse the ring when the background flow starts. After reconfiguration command is issued (at time 12 s) by the external centralized manager, the AllReduce bandwidth immediately recovers to 5.9 GB/s.



(a) Setup 1 (b) Setup 2

(c) Setup 3 (d) Setup 4

Figure 8: [Multi applications]: Application bus bandwidth. Error bars represent 95% percentile intervals.

6.3 Improving Multiple Applications

Next, we evaluate how MCCS improves the overall performance with a centralized view of all applications and collaborative schedule their collective communication. Figure 8 shows 128 MB AllReduce performance in the 4 different setups, as described in §6.1. We report bus bandwidth [25] of each application, which is normalized version of the algorithm bandwidth. Here we use bus bandwidth because it is independent of collective algorithm and the number of participants. It reflects the hardware peak bandwidth for inter GPU communication. The aggregated bus bandwidth of all applications indicates the overall network utilization, while the proportion each application gets allocated reflects fairness of allocation. For ablation study, we also compare with the baseline of MCCS without fair flow assignment. We denote this baseline as **MCCS(-FFA)**.

For all setups, MCCS (with FFA) not only achieves the highest aggregated bus bandwidth but also ensures fairness across applications. It outperforms NCCL by 75% on average. All applications in setups 1, 2, and 4 use the same amount of NICs per host, so they should have identical inter-host GPU communication performance. MCCS therefore equally distributes the bandwidth between different applications. In setup 3, application A uses 2 GPUs and 2 NICs per host, while B and C use only 1 per host. Therefore, application A's inter-host collective performance should be 2 times that of applications B and C. Again, MCCS achieves fair allocation as the bus bandwidth distribution among A, B and C is close to 2:1:1. Using ECMP fails to guarantee fairness among applications. For instance, in setup 3, the performance ratio between applications A and B for MCCS(-FFA) is 1.7:1 instead of 2:1.

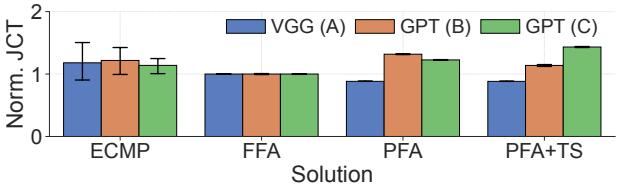


Figure 9: [Training workloads]: Job completion time using different scheduling and QoS strategies. A has the highest priority, followed by B, while C is the lowest. Error bars represent 95% percentile intervals.

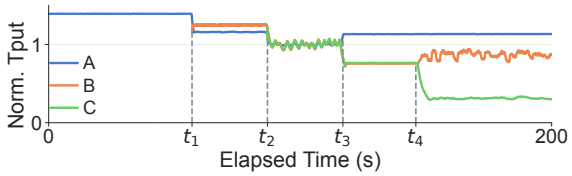


Figure 10: Normalized training throughput with dynamic job arrivals and QoS.

6.4 Training Workloads with QoS

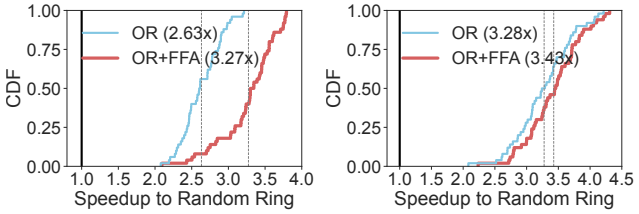
We evaluate MCCS using GPT and VGG training traces. We use setup 3 for our evaluation. We assume A, B, C represent three tenants sharing the cluster. A is assigned 4 GPUs to train a VGG model from scratch on a large dataset, while B and C are assigned 2 GPUs each to finetune GPT models.

Fair scheduling speed-ups every workload. Using our traffic generator on MCCS to simulate the workloads, we report their job completion time (JCT) in Figure 9 under different scheduling approaches. The JCT of each workload is normalized to its respective value under fair flow assignment (**FFA**). We find that ECMP routing degrades every workload. Besides having high performance variance across 10 trails, it also leads to 18%, 22%, 14% slower job completion on average, for A, B, C respectively.

QoS capabilities enable workload prioritization. Running the workloads from all three tenants at the same time inevitably result in contention of network resources. Even with fair flow scheduling, the performance of a workload would still degrade, compared to dedicate the entire network for that workload by running it independently. In this case, the infrastructure administer may prefer prioritizing some tenants. We showcase our two QoS techniques in §4.3 to demonstrate MCCS's capabilities for enabling QoS through controlling both coarse-grained resource allocation and fine-grained communication.

We assume an administrator wants to prioritize A over both B and C. Using priority flow assignment (**PFA**), we dedicate one of the two routes between the two racks to A, with B and C sharing the other one. PFA speeds up A's training by 13% compared to FFA and 34% compared to ECMP.

With A prioritized using PFA, B and C now shares a single bottlenecked route, so their performance degrades. If the administer wants to further prioritize B over C without affecting A, flow assignment no longer works no remaining routes are available that we can dedicate B to. Fine-grained communication-level QoS mechanism needs to be utilized. Hence, in this scenario, we apply our time window based traffic scheduling (**PFA+TS**) to prioritize B. Compared with PFA, in PFA+TS tenant B's training is sped up by 16%.



(a) Random placement **(b) Compact placement**

Figure 11: [Simulations]: MCCS's speedup of AllReduce completion time compared with random ring. The numbers in the legend and the vertical dashed lines represent the average speedups across jobs relative to random ring.

Dynamic policy enforcement. We demonstrate MCCS's flexibility in policy enforcement, by showing how network administers can adapt their QoS policies based on current cluster status with dynamic application arrivals. We illustrate the training throughput of A, B, C in Figure 10, where they arrive sequentially. The throughput is normalized to their values under FFA. A already occupies the cluster at the start, which is followed by B's arrival at t_1. As A has two 50G NICs per host, it can utilize all the 100G switching capacity of the network when there are no other tenants share the network. After B arrives, A's throughput is decreased by 17%. Then, C arrives at t_2 , and all three of them share the network using FFA. The throughput of A now drops further by 14%. There are also some fluctuations in the throughput of all applications, which could attribute to network congestion. After t_3, the administer prioritizes A over B and C using PFA, A's performance therefore improves by 13%. At time t_4, the administer the further prioritizes B over C using TS, the throughput of B is increased by 18%. The fluctuations after t_3 is introduced by our time window based TS.

6.5 Simulations

We evaluate how a larger scale deployment can benefit from MCCS via simulations. We compare among three solutions (1) random ring selection, (2) optimal ring (OR) selection, and (3), OR with fair flow assignment (FFA). In OR, we always create optimal rings, with the number of rings equal to the number of network multi-path choices. In OR+FFA (representing MCCS), we assign each ring to each of the path in the network.

We simulate a cluster of 768 GPUs. We have 16 spine switches and 24 leaf switches fully connected. Each leaf switch has 4 hosts connect to it. Each host has 8 GPUs and 8 NICs. All the network links and NICs are 200 Gbps. The oversubscription of the network is 2, which is identical to our testbed setting. Our flow-level simulator assumes per-flow fairness. For the workload and job arrival pattern, we adopt a similar setting as the distributed data-parallel deep

learning experiment in NetHint [8]. We run 50 jobs of ResNet-50 of model size 100 MB in each experiment. The job sizes are either 16 or 32 GPUs with equal probability. We consider two types of job placement. Random placement means the simulator allocate randomly GPUs to a job. Compact placement means the simulator assigns GPUs that belong to the same rack to a job whenever possible. The jobs arrival follows a Poisson distribution with the lambda set to 200 ms. We run each experiment 5 times and report the average speedup for each job's AllReduce completion time.

Figure 11 shows the CDF of performance improvement of using MCCS compared to using NCCL. For random placement, OR and OR+FFA speed up the collective communication by 2.6x and 3.3x compared with using random rings. With flow assignment, each job can maximize the utilization of the inter-rack network bandwidth. This is because for each network path, we assign a ring to utilize it. Without FFA, the flows within a job can collide on the same physical path. In compact placement setting, OR and OR+FFA still outperform random ring by 3.3x and 3.4x. However, FFA does not add much to OR because the job almost never span more than two racks, and the link capacity of even a single path between two racks would suffice the traffic demand. We observed that the schedule computation takes within 1 ms on average for a job size of 32 GPUs and scales linearly with the job size. The rescheduling occurs only when a job joins or exits.

7 Related Work

Integrating collective communication into the network. There are several prior efforts in integrating collective communcation into the network. ATP [21], SwitchML [39], and PANAMA [10] propose offloading AllReduce operations to in-network hardware to enable multi-tenant distributed machine learning. The key difference is that MCCS targets at the public cloud environment, where these works all require tenant applications to be trusted. In these works, a misbehaving or malicious application can circumvent the cloud provider designed collective communication strategy, and this will require well-behaving tenants to adjust their strategies accordingly. In MCCS, all the collective operations are managed through the MCCS daemon. Another difference is that MCCS's performance gain is not from in-network gradient aggregation but from dynamic adjustment to collective communication strategy.

Exposing public cloud network information for tenants to pick collective communication strategies. A separate line of work focuses on letting tenant acquire information about the physical network of the cloud provider in order to pick collective communication strategies. NetHint [8] presents an approach that the cloud provider periodically exposes a hint, containing a subset of the physical network topology and link utilization, to help the tenant pick collective communication strategy. However, a cloud provider may have security and privacy concerns of exposing their physical network topology and network utilization to cloud tenants to prevent adoption. PLink [23] and Choreo [20] let tenant applications measure their VM-level network bandwidth in order to pick collective communication strategies or decide on job scheduling. These approaches are not guaranteed to be accurate, because reverse engineering the network configurations from a single tenant's observation is generally hard. Further, in both approaches, tenants are making their own decisions on collective communication. In comparison, MCCS controls all tenants' collective communication.

Choosing collective communication strategies based on network topology and bandwidth. Optimizing collective communication strategies for particular network topology and bandwidth configuration is a standard task for developers running large-scale workloads on supercomputers [9, 17]. For machine learning workloads, several prior works have focused on improving collective performance [31, 41, 44]. These works all focus on the single-tenant scenarios. Our work focuses on the multi-tenant public cloud setting. We need to deal with challenges of dynamically changing collective communication strategies, which is not a concern in single-tenant scenarios.

Quality of Service (QoS) in a multi-tenant network. How to let multiple tenants share a cloud network with QoS guarantees is an old topic. A cloud datacenter network often uses a combination of congestion control [3, 13, 34, 46], load balancing [2, 18, 27, 45], and various types of rate limiting techniques [4, 5, 16, 19, 22, 33]. These works focuses on how to share bandwidth given a set of point-to-point network demand. The optimizations MCCS addresses is on having multiple collective communication operations share the bandwidth by selecting collective communication strategies (e.g., the ordering of nodes in an AllReduce ring for each tenant), which is a different and complementary problem.

8 Conclusion

This paper explores a new service-based approach to collective communication called MCCS. MCCS allows a cloud provider to select collective communication strategies for cloud tenants and enable the cloud provider to enforce QoS policies on collective communication operations. Collective communication strategies selected by the cloud provider improves tenant performance because the strategies is picked with the knowledge of the underlying cloud network characteristics (i.e., topology, utilization) and can adapt when network characteristics changed. Our testbed and simulation-based evaluations have shown that MCCS improves tenant collective communication performance by up to 2.4x compared to state-of-the-art collective communication libraries, while adding more management features including dynamic adjustment of collective communication algorithm, quality of service, and network-aware traffic engineering. *This work does not raise any ethical issue.*

Acknowledgments

We thank our shepherd Yuliang Li and other anonymous reviewers for their insightful feedback. Our work is partially supported by NSF grant CNS-2238665 and by gifts from Adobe, Amazon, IBM, and Meta.

References

[1] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. 2010. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*.

[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *SIGCOMM*.

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *SIGCOMM*.

[4] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. 2014. End-to-end Performance Isolation Through Virtual Datacenters. In *OSDI*.

[5] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards Predictable Datacenter Networks. In *SIGCOMM*.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-shot Learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.

[7] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2021. Synthesizing Optimal Collective Algorithms. In *PPoPP*.

[8] Jingrong Chen, Hong Zhang, Wei Zhang, Liang Luo, Jeffrey Chase, Ion Stoica, and Danyang Zhuo. 2022. NetHint: White-Box Networking for Multi-Tenant Data Centers. In *NSDI*.

[9] Mathijs Den Burger, Thilo Kielmann, and Henri E Bal. 2005. Balanced Multicasting: High-Throughput Communication for Grid Applications. In *ACM/IEEE Conference on Supercomputing (SC)*.

[10] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. 2021. In-Network Aggregation for Shared Machine Learning Clusters. In *MLSys*.

[11] Gloo 2023. Collective Communications Library with Various Primitives for Multi-Machine Training. https://github.com/facebookincubator/gloo. (2023).

[12] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. 2005. Open MPI: A Flexible High Performance MPI. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, 228–239.

[13] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *SIGCOMM*.

[14] Olaf Hartmann, Matthias Kühnemann, Thomas Rauber, and Gudula Rünger. 2005. Adaptive Selection of Communication Methods to Optimize Collective MPI Operations. In *PARCO*.

[15] intelmpi 2024. Intel MPI Library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html. (2024).

[16] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. 2013. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*.

[17] Nicholas T Karonis, Bronis R De Supinski, Ian Foster, William Gropp, Ewing Lusk, and John Bresnahan. 2000. Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance. In *IPDPS*.

[18] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. 2017. Clove: Congestion-Aware Load Balancing at the Virtual Edge. In *CoNEXT*.

[19] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, and Amin Vahdat. 2019. PicNIC: Predictable Virtualized NIC. In *SIGCOMM*.

[20] Katrina LaCurts, Shuo Deng, Ameesh Goyal, and Hari Balakrishnan. 2013. Choreo: Network-Aware Task Placement for Cloud Applications. In *IMC*.

[21] 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 *NSDI*.

[22] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. 2014. Application-Driven Bandwidth Guarantees in Datacenters. In *SIGCOMM*.

[23] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. 2020. PLink: Discovering and Exploiting Locality for Accelerated Distributed Training on the Public Cloud. In *MLSys*.

[24] nccl 2023. The NVIDIA Collective Communication Library (NCCL). https://developer.nvidia.com/nccl. (2023).

[25] NVIDIA. 2023. Performance Reported by NCCL Tests. https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md. (2023).

[26] nvidiasharp 2024. NVIDIA Scalable Hierarchical Aggregation and Reduction Protocol (SHARP). https://docs.nvidia.com/networking/display/sharpv300. (2024).

[27] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless Datacenter Load-balancing with Beamer. In *NSDI*.

[28] oneccl 2023. oneAPI Collective Communications Library (oneCCL). https://github.com/oneapi-src/oneCCL. (2023).

[29] Siddharth Pal, Liangyu Zhao, Jason Fantl, Joud Khoury, Arvind Krishnamurthy, and Prithwish Basu. 2023. Efficient All-to-All Collective Communication Schedules for Direct-Connect Topologies. *arXiv preprint arXiv:2309.13541* (2023).

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems* 32 (2019).

[31] Y Peng, Y Zhu, Y Chen, Y Bao, B Yi, C Lan, C Wu, and C Guo. 2019. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *SOSP*.

[32] Jelena Pješivac-Grbović, George Bosilca, Graham E. Fagg, Thara Angskun, and Jack J. Dongarra. 2007. MPI Collective Algorithm Selection and Quadtree Encoding. *Parallel Comput.* (2007).

[33] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. 2013. ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing. In *SIGCOMM*.

[34] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. 2011. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM*.

[35] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. 2024. CASSINI: Network-Aware Job Scheduling in Machine Learning Clusters. In *NSDI*.

[36] Saeed Rashidi, William Won, Sudarshan Srinivasan, Srinivas Sridharan, and Tushar Krishna. 2022. Themis: A Network Bandwidth-Aware Collective Scheduling Policy for Distributed Training of DL Models. In *ISCA*.

[37] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[38] rccl 2023. ROCm Communication Collectives Library (RCCL). https://github.com/ROCm/rccl. (2023).

[39] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *NSDI*.

[40] Daniele De Sensi, Tommaso Bonato, David Saam, and Torsten Hoefler. 2024. Swing: Short-cutting Rings for Higher Bandwidth Allreduce. In *NSDI*.

[41] 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 *NSDI*.

[42] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[43] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014).

[44] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. 2020. Blink: Fast and Generic Collectives for Distributed ML. In *MLSys*.

[45] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. 2017. Resilient Datacenter Load Balancing in the Wild. In *SIGCOMM*.

[46] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM*.