Accelerating Collective Communication in Data Parallel Training across Deep Learning Frameworks

Joshua Romero[1], Junqi Yin[2], Nouamane Laanait[2,*], Bing Xie[2], M. Todd Young[2], Sean Treichler[1], Vitalii Starchenko[2], Albina Borisevich[2], Alex Sergeev[3,†], Michael Matheson[2]
[1]NVIDIA, Inc. [2]Oak Ridge National Laboratory [3]Carbon Robotics

Abstract

This work develops new techniques within Horovod, a generic communication library supporting data parallel training across deep learning frameworks. In particular, we improve the Horovod control plane by implementing a new coordination scheme that takes advantage of the characteristics of the typical data parallel training paradigm, namely the repeated execution of collectives on the gradients of a fixed set of tensors. Using a caching strategy, we execute Horovod's existing coordinator-worker logic only once during a typical training run, replacing it with a more efficient decentralized orchestration strategy using the cached data and a global intersection of a bitvector for the remaining training duration. Next, we introduce a feature for end users to explicitly group collective operations, enabling finer grained control over the communication buffer sizes. To evaluate our proposed strategies, we conduct experiments on a world-class supercomputer — Summit. We compare our proposals to Horovod's original design and observe 2× performance improvement at a scale of 6000 GPUs; we also compare them against tf.distribute and torch.DDP and achieve 12% better and comparable performance, respectively, using up to 1536 GPUs; we compare our solution against BytePS in typical HPC settings and achieve about 20% better performance on a scale of 768 GPUs. Finally, we test our strategies on a scientific application (STEMDL) using up to 27,600 GPUs (the entire Summit) and show that we achieve a near-linear scaling of 0.93 with a sustained performance of 1.54 exaflops (with standard error +- 0.02) in FP16 precision.

1 Introduction

The recent successes of Deep Neural Networks (DNNs) have encouraged continued investment across industries and domain sciences. Ranging from the traditional AI (e.g., image processing, speech recognition), to pharmaceutical and biomedical sciences (e.g., drug discovery), and to fusion, combustion and nuclear energy (e.g., disruption predictor, nuclear power plant) [29–34], more and more applications are actively exploiting ever-larger DNNs for production use.

With the growing applications of ever-larger DNNs, data parallelism in DNN training faces unprecedented challenges when synchronizing gradients[1] throughout distributed training runs. Deep learning (DL) frameworks, such as PyTorch [5] and TensorFlow [7], can exploit data parallelism for DNN training. In such a training run, an application creates multiple replicas of a model and distributes the replicas among a group of accelerators (e.g., CPUs, GPUs, TPUs, etc). Each accelerator executes on a different portion of training data across a number of *iterations*; at each iteration, it performs forward/backward pass computations independently, but synchronizes gradients (typically via global averaging) among the accelerators before applying weight updates (§2.1). In particular, accelerators synchronize tensors (multi-dimensional arrays) of gradients for the same set of parameters to ensure a globally consistent state for the model replicas.

This work advances collective communication in data parallel training. We propose several enhancements to Horovod [3] [25], a generic communication library designed to be independent to the framework runtimes, enabling its use across numerous popular DL frameworks with the same underlying backend implementation. Our ideas were motivated by two observations on Horovod. First, we observed that Horovod's core design is not scalable (see Figure 3) as it relies on a coordinator-worker control plane to orchestrate collective operations. At larger scales, this design choice leads to the single coordinator becoming overwhelmed and leaves the application runtime dominated by the orchestration process. Second, we found that Horovod's buffering mechanism (*Tensor Fusion*) fails to reliably generate optimal buffer sizes for efficient network bandwidth utilization (§2.2).

[*]Nouamane Laanait conducted this research when he was with Oak Ridge National Laboratory.

[†]Alex Sergeev conducted this research when he was with Uber, Inc.

[1]Centralized training (also called synchronous training) synchronizes gradients among accelerators; decentralized training (asynchronous training) synchronizes parameters [13] [14] [21]. This work optimizes centralized training and discusses gradient synchronization accordingly.

To address these inefficiencies, we improve the control plane with a new coordination scheme that takes advantage of characteristics of a typical data parallel training paradigm, namely the repeated execution of collectives on a fixed set of gradients (§2.1). Using a caching strategy, we execute Horovod's existing coordinator-worker logic only once during a training run, replacing it with a more efficient decentralized orchestration strategy using a globally intersected bitvector for the remaining training duration (§3.1). Moreover, we introduce a feature for end users to explicitly group collective operations within Horovod, enabling finer grained control over the communication buffer sizes used for reductions.

While the implementation details vary, most DL-based communication libraries use similar design principles to optimize the performance of gradient synchronization. First, these libraries will employ mechanisms to facilitate overlapping of gradient synchronization and backward pass. That is, rather than waiting for gradients of all parameters to be computed and then synchronizing them across accelerators altogether at once, gradients will be synchronized actively as they are computed during the backward pass. Second, rather than launching a synchronization operator (e.g., AllReduce) for each gradient individually, the libraries employ bucketing/packing/fusion strategies (e.g., torch.DDP [18], tf.distribute [6], Horovod) to aggregate the gradients of multiple parameters and execute AllReduce on larger communication buffers for improved bandwidth utilization.

The contributions described in this work are mainly enhancements specific to Horovod, overcoming inefficiencies in its framework-agnostic design and original coordinator-worker strategy. The framework native communication libraries, like tf.distribute and torch.DDP, are closely integrated within their respective frameworks with access to internal details. With access to these details, the implementation of well-organized and performant communication and similar advanced features like grouping are simpler in these libraries. While the implementation details in this paper are Horovod specific, the proposed grouping technique is generally applicable to any other collective communication libraries.

In particular, we summarize our contributions as follows:

1. We implement a light weight decentralized coordination strategy by utilizing a response cache to enable Horovod to reuse coordination-related information collected at application runtime, accelerating the orchestration process.

2. We enable grouping to provide end users with explicit controls over tensor fusion in Horovod.

3. Our developments are incorporated in Horovod and publicly available in Horovod v0.21.0.

4. We conduct experiments to evaluate our solution on a world-class supercomputer — Summit. The results show that: 1) our solution outperforms Horovod's existing strategies across scales consistently. 2) Compared to the framework native communication libraries such like tf.distribute and torch.DDP, we achieve comparable and/or better performance across scales consistently. Compared to a PS (parameter server)-based communication library BytePS [24], we achieve 20% better performance using up to 768 GPUs. 3) we further evaluate our solution on a scale up to 27,600 GPUs (the entire Summit) and show that we achieve near-linear scaling of 0.93 with a sustained performance of 1.54 exaflops (with standard error +- 0.02) in FP16 precision.

2 Background and Motivation

2.1 Data Parallelism in DNN Training

For data parallelism in distributed DNN training, a typical application run usually executes an iterative learning algorithm (e.g., SGD) among a number of GPUs; each GPU works on an identical replica and the same set of parameters of a DNN model. Here, a parameter is the bias or weight of a DNN layer; the value of a parameter or the value of a parameter's gradient is a multi-dimensional array, referred to as a *tensor*. In the run, a training dataset is partitioned into one or more equal-sized *batches*; each batch is processed on a different GPU. After a run starts, the model replicas, parameters, and the data structures of tensors are all fixed and determined.

During an iteration, each GPU updates parameters of a model replica by the following computational procedure: 1. the forward pass to compute loss. 2. the backward pass to compute gradients of the parameters. 3. the optimization step to update the parameters. In order to ensure model replicas are updated identically and remain in a globally consistent state, the gradients between GPUs are synchronized via averaging before updating parameters; this is referred to as *centralized learning*. Decentralized learning [13] [14] [21] maintains local consistency based on communication graphs [2] and synchronizes parameters. Moreover, for both centralized and decentralized learning, GPUs synchronize the same set of parameters/gradients across iterations. In this work, we focus on centralized learning and discuss collective communication in gradient synchronization/reduction.

Observation ①. For a DNN training run on a DL framework, the model replicas and parameters are all fixed. Across iterations in the run, GPUs repeatedly synchronize the same set of tensors for parameters/gradients.

2.2 Communication Libraries for Gradient Synchronization

2.2.1 Framework-native Libraries

For data parallel training, the key communication operations that occurs are AllReduce operations which average gradients among GPUs. Within an iteration, the framework processes

[2] In decentralized learning, GPUs are structured into a communication graph (e.g., ring or torus); each GPU only synchronizes among its local neighbors on the graph.

on GPUs each generate a set of gradients during the backward pass that must be globally reduced before being used to update the model parameters.

DL frameworks typically use dependency graphs to schedule compute operations, the use of which may result in non-deterministic ordering of operations. This is because in general, the order of operations through the compute graph that satisfies all dependencies is not unique. As a result, the order of operations executed can vary across framework processes within a single iteration, or even between iterations on a single process. This leads to problems in handling gradient communication between processes, as the operations generating gradients may occur in varied orders across processes. If each framework process naively executes a blocking AllReduce on the gradients in the order they are produced locally, mismatches may arise leading to either deadlock or data corruption. A communication library for DL must be able to manage these non-deterministic ordering issues to ensure that AllReduce operations are executed between processes in a globally consistent order.

The framework-native communication libraries (e.g., tf.distribute and torch.DDP) are designed to be closely integrated within the framework and have direct access to internal details, such as the model definition and expected set of gradients to be produced each iteration. Access to this information enables these libraries to directly discern the communication required during an iteration and more easily implement a performant communication schedule. For example, torch.DDP is a wrapper around a model in PyTorch, and utilizes the information contained in the model about gradients to determine how to schedule AllReduce operations during an iteration. While access to this information can simplify the implementation of these communication libraries, it ties their implementations strictly to the frameworks they were designed to support.

2.2.2 Framework-agnostic Libraries

In contrast to the framework-native communication libraries, a framework-agnostic library avoids any reliance on internal framework details and makes communication scheduling decisions based on information deduced during runtime. This design choice enables the library to operate across numerous frameworks, but the lack of access to internal information presents unique challenges. This section discusses the design of Horovod, a framework-agnostic communication library.

Horovod is a generic communication library developed to execute collective communication in data parallel training on GPUs, CPUs and TPUs, and with support for various DL frameworks. It serves as a high-level communication library that leaves network routing details (e.g., network reordering) handled by lower-level libraries, such as MPI, etc. Without loss of generality, this section discusses how Horovod integrates with MPI and TensorFlow on GPUs. Assuming this scenario, a distributed training run has N identical model replicas, and is executed on N GPUs managed by Horovod with MPI and TensorFlow. In the run, each GPU serves as both an MPI rank and a TensorFlow process[3], which conducts computations for a model replica across iterations, with Horovod providing communication routines to synchronize gradients across TensorFlow processes.

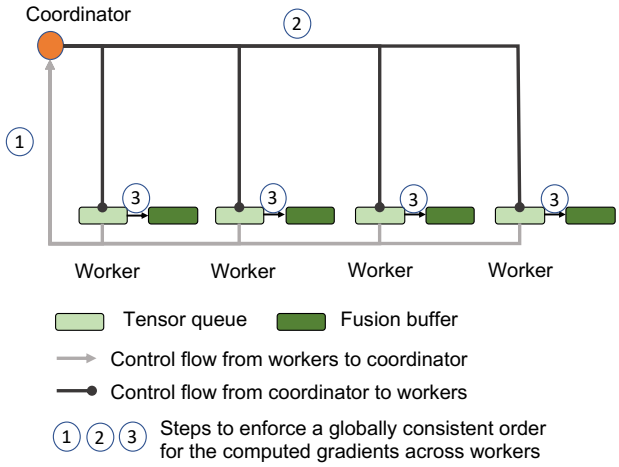

Figure 1: Coordinator-worker control model in Horovod's original design. The coordination progresses in three steps (see details in §2.2.2): First, the coordinator gathers the lists of requests from all workers; Second, the coordinator processes the request lists, and then generates and broadcasts a response list when observing one or more common requests from all workers; Third, after receiving the response list, each worker proceeds to execute collective operations.

This work introduces new techniques to Horovod after v0.15.2. In this section, we summarize the existing strategies based on v0.15.2. We use the terms *rank*, *process*, and GPU to refer to MPI rank, TensorFlow process, and their hosting GPU in turn, and use the terms *coordinator* and *worker* to refer to the Horovod threads spawned from the processes.

Similar to the framework-native libraries, Horovod must deal with the non-deterministic ordering of computations (discussed in §2.2.1). As it is agnostic to frameworks and lacking the knowledge of framework internal details, Horovod's design uses a control plane to resolve the non-deterministic ordering issue, where a coordinator-worker communication model is adopted to orchestrate collective communication and ensure a globally consistent order of execution.

Figure 1 presents a simple diagram of Horovod's control plane, with four threads each launched in a DL framework process. Particularly, the thread in Rank 0 serves as both the *coordinator* and a *worker*, and the other threads each serve as a different worker on a different GPU. During the course of a training run, the coordinator and workers execute the control logic periodically, with each execution referred

[3]For Horovod with TensorFlow, it is possible to use multi-GPUs per rank. But in production use, most users let each rank use a different GPU.

to as a *cycle*. In Horovod, the time between two sequential cycles is a configurable parameter with a default setting of 1 ms. To ensure synchronous cycles across Horovod threads, the communication operations in control plane (e.g., gather, broadcast) are blocking.

When a cycle starts, the coordinator first gathers lists of *requests* from all workers. Each request contains the metadata (e.g., tensor name, operation) that defines a specific collective operation on a specific tensor requested to be executed by the framework. The requests are collected from the worker's local tensor queue and are structured as a *request list*.

Next, the coordinator processes the request lists and counts the submissions of each request (identified by tensor name) from workers. When the coordinator observes that a common request has been submitted by all workers, it prepares that request for execution by generating a corresponding *response*. The coordinator generates a list of responses and broadcasts the list to all workers. Here, each response contains the metadata (e.g., tensor names, data type, collective operation) that is used by the Horovod backend to execute a collective operation (e.g., AllReduce). Optionally, before broadcasting, the coordinator will preprocess the response list, aggregating multiple compatible responses into larger *fused* responses, a process referred to as *Tensor Fusion* in Horovod documentation.

After receiving the response list, each worker proceeds to execute collective operations, one operation per response in the received response list. The portion of the Horovod backend executing collective operations is referred to as the *data plane*. For each response, a worker will access required input tensor data from the framework, execute the requested collective operation, and populate the output tensors for the framework's continued use. A key characteristic of this design is that the order of execution for collective operations is defined by the order of responses in the list produced by the coordinator. As such, a globally consistent ordering of collective operation execution is achieved across workers.

At a high-level, Horovod's design can be described as a set of mailboxes, where each worker is free to submit request for collectives in any order to their assigned mailbox, and eventually retrieve the desired output. The control plane is responsible for coordinating these requests across mailboxes, ensuring that only requests submitted by all workers are executed and are executed in a globally consistent order. One observation from this analogy is that Horovod's design is inherently unaware of any aspects of DL training, in particular that in typical DL workloads, a fixed set of gradient tensors will be repeatedly AllReduced during the course of a training run (discussed in §2.1). As a result, Horovod's design unnecessarily communicates redundant information to the coordinator at every iteration, leading to poor scalability.

Beyond coordination alone, tensor fusion may cause inefficiency in the data plane. Ideally, the tensor fusion process will generate well balanced fused responses throughout training, yielding larger sized communication buffers for improved network utilization. In practice, as the tensor fusion is closely tied to cycle that runs at an arbitrary user-defined tic rate, the resulting communication buffer sizes can be highly dynamic and varied, even when comparing iteration to iteration in a run. Figure 2 presents the fused AllReduce message sizes on ResNet50 as an example to illustrate the performance of tensor fusion. In summary, it is possible to have the Horovod cycles occur at favorable times during the training iteration, where the collective responses are well distributed across the Horovod cycles running during the iteration, resulting in correspondingly well-balanced fused communication message sizes. On the other hand, the Horovod cycles can occur at unfavorable times during the iteration, with some cycles completing with a few or even just one available collective response, yielding less efficient communication on smaller buffers. In the worst case, a single trailing gradient tensor for the iteration can be missed by all previous cycles run during the iteration, inducing additional latency equal to the user-defined cycle time, just to reduce a single gradient tensor.

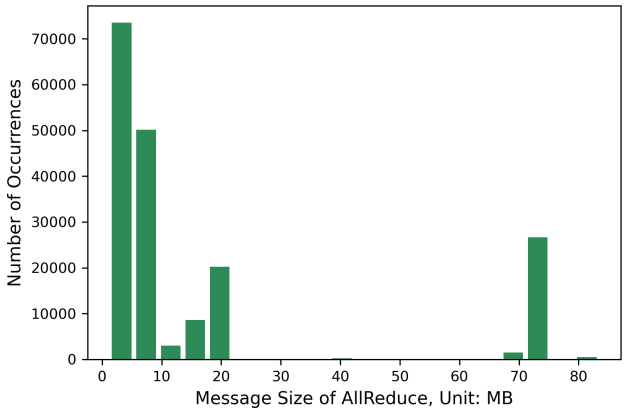

Figure 2: Histogram of AllReduce message size in Horovod's original design of Tensor Fusion. We present the results of a training run of ResNet50 with 96 GPUs on Summit (§4.1).

We report the detailed information about the original design of Horovod's control plane in the supplementary materials (Section 1), including pseudo code listings for Horovod coordinator-worker coordination logic and Horovod cycle, and the data structures for request list and response list.

Observation (2). The dynamic nature of tensor fusion can fail to generate buffer sizes for efficient network utilization. Thus, we are motivated to introduce a more explicit and strict control mechanism for tensor fusion that can improve performance.

2.2.3 Hierarchical Approach in Horovod

Kurth et al. [15] were the first to observe the scaling issue in Horovod's control plane. In particular, the coordinator-worker coordination strategy was found to be highly inefficient. When increasing the number of workers, the time cost of the communication and processing grows linearly since the coordinator needs to communicate/process the request list

from each worker. Especially at large scale, the cost of this coordination strategy was found to quickly dominate the training runtime. Their proposed solution was to introduce a hierarchical tree-based variant of the original coordinator-worker control model, using a hierarchy of coordinators splitting up the coordination tasks. It is clear that this hierarchical control strategy outperforms the original control plane with a logarithmic complexity, but at the same time, it suffers from the same issue as the original strategy does: the hierarchical coordination strategy redundantly communicates metadata for repeated operations across iterations in a training run.

Beyond the hierarchical coordination strategy, the authors also introduced a hybrid/hierarchical AllReduce in Horovod's data plane. Even with these improvements, their approach was not able to achieve efficient scaling with Horovod, requiring the introduction of a *gradient lag*. With gradient lag enabled, the gradients of a previous iteration are used to update weights in the current step, providing a longer window for overlapping the slower communication at scale with computation.

We present the hierarchical control plane in detail in the supplemental materials (Section 1) and discuss the performance of the hierarchical approach in Section 2.3.

Observation ③. Although existing Horovod solutions adopt different coordination strategies, they both fail to take advantage of characteristics of DL workloads and repeat the same metadata communications in the control plane across iterations in a training run.

2.3 Discussions on Horovod Performance

We focus on understanding the performance of existing Horovod solutions, including `Horovod_MPI`, `Horovod_NCCL`, and the hierarchical AllReduce (`Hierarchical_AllReduce`). Here, `Horovod_MPI` refers to the Horovod implementation with MPI for both the coordinator-worker communication in the control plane and AllReduce in the data plane. `Horovod_NCCL` refers to the implementation that uses MPI for control plane communication and NCCL for AllReduce in the data plane. In particular, NCCL v2.4.0 was used in this experiment, with tree-based communication algorithm options available along with existing systolic ring algorithm. `Hierarchical_AllReduce` represents the solution using MPI for the control plane communication and MPI+NCCL for the AllReduce in the data plane. In all three solutions, the coordinator-worker communication uses the control plane as shown in Figure 1. Moreover, all these solutions are available in Horovod [3].

We conducted experiments on STEMDL (See supplementary materials Section 3), a scientific application developed to solve a long-standing inverse problem on scanning transmission electron micro-scopic (STEM) data by employing deep learning. The DNN model in STEMDL is a fully-convolutional dense neural network with 220 million parameters; each GPU generates/reduces 880MB of gradients at an iteration. We ran the experiments on Summit supercomputer (§4.1), where each Summit node contains 6 GPUs.

We first consider the scalability results, shown in the left subfigure of Figure 3. It is clear that, after introducing the tree-based communication algorithms, `Horovod_NCCL` is able to deliver the best performance for all scales. When we increase the number of GPUs, `Horovod_NCCL` expands its lead in system throughput. For example, when using 6000 GPUs, it outperforms `Hierarchical_AllReduce` and `Horovod_MPI` by 3.2× and 5.4×, respectively.

Figure 3 (right subfigure) also reports the GPU utilization of the Horovod solutions across scales. The results show that, across all tested configurations, the GPU utilization is below 55%. When increasing the number of GPUs, the GPU utilization decreases progressively. We observed a much lower GPU utilization with 6000 GPUs (see Figure 6). This indicates that, although the NCCL-based AllReduce delivers good performance, the entire gradient reduction procedure in Horovod (e.g., coordination and execution) is highly inefficient. It leaves GPU resources underutilized and compromises system throughput. In this work, we argue that the inefficiency originates from both the control plane and AllReduce and introduce techniques (discussed in §3) to overcome these issues.

We limit the evaluation on `Horovod_MPI` to 1536 GPUs as we see noticeably poor performance. We skip the evaluations of the hierarchical tree-based coordinator-worker communication (Figure 1 in supplementary materials) and the gradient lag proposed in the hierarchical approach (§2.2.3), as they are currently neither included as part of Horovod nor publicly available. To summarize, Kurth et al. reported in [15] that, the entire hierarchical approach obtained the parallel efficiencies of $\sim$ 60% when using fully synchronous gradient reduction, only achieving above 90% on the Summit supercomputer with gradient lag enabled. In particular, researchers showed that gradient lag sometimes yields low training accuracy, and concluded that, without carefully tuning the related hyperparameters, this type of techniques is not generally applicable to DNN training [9, 10, 20]. Moreover, we show that our solution obtains up to 93% of parallel efficiency on Summit using a fully synchronous gradient reduction (discussed in §4.4), 1.5× better than the performance of the hierarchical approach without gradient lag reported in [15].

3 Boosting Collective Communication in DNN Training with Caching and Grouping

This work proposes to advance collective communication in centralized learning across various DL frameworks. We introduce new techniques to Horovod to improve its scalability and efficiency in both the control plane and the data plane. For the control plane, we develop a strategy to record the coordination information on the repeated requests for the same collective operations on the same parameters across

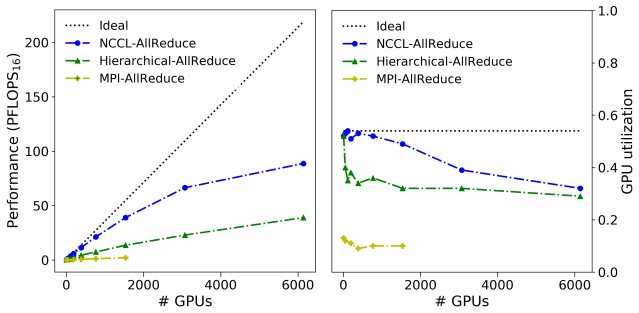

Figure 3: Performance and GPU utilization of existing Horovod strategies on STEMDL workload.

iterations in a training run (discussed in §2.1). In particular, we develop a light weight decentralized coordination strategy by utilizing a *response cache*. This cache introduces a means for Horovod to store the metadata about the repeated collective requests at each worker locally and bypass the redundant coordinator-worker communication entirely after the cache is populated. Moreover, we introduce *grouping* as a feature to Horovod's data plane. With grouping enabled, a user can request grouped collective operations for specific tensor groups, enforcing explicit control over Horovod's tensor fusion. We later show in experiments (§4) that, these two techniques can lead to significant performance improvement and obtain near-linear scaling in the production runs on a world-class supercomputer. Our techniques are adopted by Horovod and are publicly accessible in v.0.21.0.

In general, our proposals are built within Horovod's existing control logic (discussed in §2.2.2): we execute cycles to coordinate collective communication in DNN training; in our system, blocking communications are used to ensure synchronous cycles across workers and the network routing details (e.g., network reordering) are managed by lower-level communication libraries, such as MPI. Additionally, our modifications support both MPI and Gloo [2] libraries for control plane communication. We discuss the performance evaluation using MPI for control plane communication and either MPI or NCCL for data plane communication in Section 4.

3.1 Orchestrating Collective Communication with Caching

In contrast to the framework-native communication libraries like tf.distribute or torch.DDP, Horovod is designed to be generic. It utilizes lightweight bindings into frameworks to allow the Horovod runtime to process gradient reduction, and has no access to any data associated with the framework runtimes (e.g., iteration, parameters, models, etc.). In particular, Horovod interacts with DL frameworks via custom framework operations that enable the frameworks to pass a tensor and requested collective operation to the Horovod backend, and receive the output tensor after the collective is executed. These custom operations are defined for each supported framework, as the mechanisms to share tensor data can vary between frameworks, but otherwise the remainder of the code base is generic. This design choice enables Horovod to work across numerous DL frameworks using the same underlying code, but at the same time, this generic design leads to the inefficiency at scale with its centralized coordinator-worker control plane.

As is summarized in **Observation** ①, in a typical data parallel training run, there is a fixed set of gradients that needs to be AllReduced across iterations. Horovod's existing coordinator-worker design does not take advantage of this aspect of the workload, and will redundantly process the same collective communication requests through the coordinator at each iteration (**Observation** ③). Although this design choice allows Horovod to be dynamic and service any collective request submitted from workers, it is unnecessarily inefficient for the typical use case with a fixed set of repeated collective operations.

This section introduces a caching strategy that enables Horovod to capture and register repeated collective operations at runtime. With the cached metadata, we build a decentralized coordination among workers, replacing the existing strategy with significant performance improvement.

3.1.1 Response Cache

As Horovod does not have direct access to the framework-runtime metadata (e.g., iteration, tensors), any pattern of collective operations launched during a training run must be deduced at runtime based on prior collective requests observed. In order to capture the metadata about repeated collective operations, we introduce a *response cache* to Horovod. This cache can be used to identify repeated operations, as well as store associated *response* data structures generated by the coordinator to be reused without a repeated processing through the coordinator-worker process.

Each worker maintains a response cache locally. To construct the cache, Horovod threads will use the existing coordinator-worker control plane implementation. Specifically, workers send requests to the coordinator and receive a list of responses from the coordinator to execute. Instead of executing the collective operations immediately and destroying the response objects, the workers first store the response objects in a local cache, where each unique response is added to a linked-list structure. Additional tables are kept mapping tensor names to response objects in the cache as well as integer position indices in the linked list. A key characteristic of the cache design is that its structure is fully deterministic based on the order that response entries are added to the cache. In this design, the cache is populated using the list of responses received by the coordinator when a collective request is first processed. As the coordinator design already enforces a global ordering of responses, responses are added

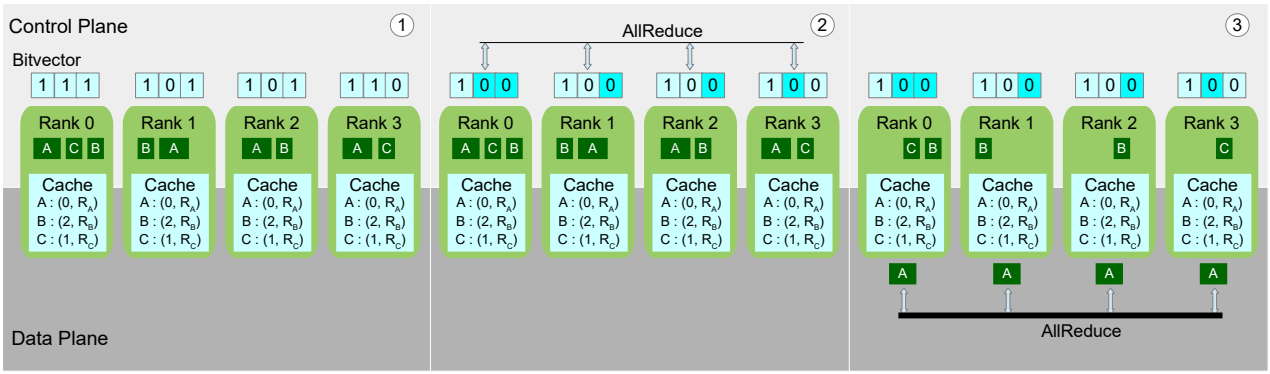

Figure 4: Illustration of AllReduce with Caching. We depict an example with 4 workers (0, 1, 2, 3) reducing 3 tensors (A, B, C). The strategy works in three steps: 1. Each worker populates a *bitvector*, setting bits according to entries in the *response cache* and the pending tensors in their local queues. 2. Workers synchronize the bitvectors via a global set intersection to identify common pending tensors. In this example, the bit associated with tensor A is shown as common across the workers. 3. Tensor A is sent to the data plane for AllReduce. When the AllReduce operation is done, Tensor A is removed from the queues on all workers.

to the cache on each worker in a globally consistent order which in turn ensures caches on each worker remain identical across workers. The data structure for each entry in the cache is the same as a response list discussed in Section 2.1. The cache implemented has a user-configurable capacity, with a default size of 1024 unique responses.

Using a combination of the cached responses and the globally consistent structure of the caches, a lightweight decentralized coordination scheme is enabled, as illustrated in Figure 4.

3.1.2 Cache-based Coordination with Response Cache and Bitvector

Once the response cache is created, it is utilized together with a bitvector to implement a lightweight decentralized coordination scheme. To achieve this, we take advantage of the fact that the response cache is constructed in a way that guarantees global consistency across workers. As a result, the structure of the response cache, in particular the index position of cached response entries, can be used to maintain a global indexing scheme of requests that are repeated that can be leveraged for coordination. We present the strategy in Figure 4, report the corresponding pseudo code in Algorithms 1 and 2, and summarize its procedure below.

1. At the start of a cycle, each worker performs the same operations as it does in the original design: it retrieves the pending requests from its local tensor queue, yielding a *RequestList*.

2. Each request in *RequestList* is checked against the response cache. If the request has an associated entry in the cache, the position of the cached entry is added to a set, *CacheBits*. Otherwise, this request does not have an associated cached entry and a flag is set to indicate that an uncached (i.e. previously unobserved) request is pending.

Algorithm 1 Horovod cycle with caching

```
1:  procedure RunCycleOnce
2:      RequestList ← PopMessagesFromQueue()
3:      CacheBits_g, UncachedInQueue_g ← CacheCoordination(RequestList)
4:      UncachedRequestList ← []
5:      for M in RequestList do
6:          cached ← ResponseCache.cached(M)
7:          if cached then
8:              bit ← ResponseCache.GetCacheBit(M)
9:              if bit ∉ CacheBits_g then
10:                 PushMessageToQueue(M)   ▷ Replace messages corresponding to uncommon bit positions to framework queue for next cycle
11:             end if
12:         else
13:             UncachedRequestList.append(M)   ▷ Collect any uncached messages
14:         end if
15:     end for
16:     ResponseList ← ResponseCache.GetResponses(CacheBits_g)   ▷ Retrieve cached responses corresponding to common bit positions
17:     if not UncachedInQueue_g then   ▷ All messages cached, skip master-worker coordination phase
18:         FusedResponseList ← FuseResponses(ResponseList)   ▷ Tensor Fusion
19:     else   ▷ Use master-worker coordination to handle uncached messages
20:         FusedResponseList ← MasterWorkerCoordination(UncachedRequestList, ResponseList)
21:     end if
22:     for R in FusedReponseList do
23:         ResponseCache.put(R)   ▷ Add response to cache
24:         PerformOperation(R)   ▷ Perform collective operation
25:     end for
26: end procedure
```

3. Each worker populates a bit vector, *BitVector*, setting bits corresponding to values in *CacheBits*. It also sets a bit to indicate whether it has uncached requests in its queue. The bit vectors across workers are globally intersected using an

Algorithm 2 Decentralized coordination with response cache and bitvector

```
1:  procedure CACHECOORDINATION(RequestList)
2:      CacheBits ← {}, UncachedInQueue ← False
3:      for M in RequestList do                          ▷ Check for cached messages
4:          cached ← ResponseCache.cached(M)
5:          if cached then
6:              bit ← ResponseCache.GetCacheBit(M)
7:              CacheBits.insert(bit)                    ▷ Collect bit positions for cached entries
8:          else
9:              UncachedInQueue ← True                   ▷ Record uncached message exists
10:         end if
11:     end for
12:     BitVector ← SetBitvector(CacheBits, UncachedInQueue) ▷ Set bits in local bitvector
13:     BitVector_g ← Intersect(BitVector)               ▷ AllReduce using binary AND op to get global bitvector
14:     CacheBits_g, UncachedInQueue_g ← DecodeBitVector(BitVector_g)  ▷ Get common bit positions and flag
15:     return CacheBits_g, UncachedInQueue_g
16: end procedure
```

AllReduce with the binary AND operation, resulting in a globally reduced bitvector, $BitVector_g$. Through this operation, only bits corresponding to requests that are pending on all workers remain set, while others are zero.

4. Each worker decodes $BitVector_g$, collecting indices of any remaining set bits to form $CacheBits_g$, the set of cache indices corresponding to requests currently pending on all workers. Additionally, it extracts whether any worker has pending uncached requests in queue.

5. Each request in *RequestList* is checked against the entries in $CacheBits_g$. If the request has an associated cache entry but has a position not in $CacheBits_g$, this means that only a subset of workers have this cached request pending. This request is pushed back into the internal tensor queue to be checked again on a subsequent cycle. If the request has an associated cache entry with a position in $CacheBits_g$, this means that the request is pending on all workers and is ready for communication. The associated response is retrieved from the cache and added to the *ResponseList*. If the request is not cached, it is added to a list of uncached requests that needs to be handled via the coordinator-worker process.

6. If there are no uncached requests pending on any worker, the coordinator-worker process is completely skipped and workers proceed to process locally generated *ResponseLists* composed of response entries from the cache. Otherwise, uncached requests are handled via the coordinator-worker process, with the coordinator rank generating a *ResponseList* containing the cached response entries along with new responses corresponding to the uncached requests.

It is worth highlighting that with this cache-based control, the coordinator-worker logic is only executed during cycles where previously unobserved requests are submitted to Horovod. In cycles where all requests are cached (i.e. repeated), the coordinator-worker control plane is never executed. For a typical DL workload with a fixed set of gradients to reduce every iteration, the response cache will eventually contain entries corresponding to this entire set. As a result, the poorly scaling coordinator-worker process will be skipped for all training iterations, except the first one, where all requests are initially observed and placed into the cache.

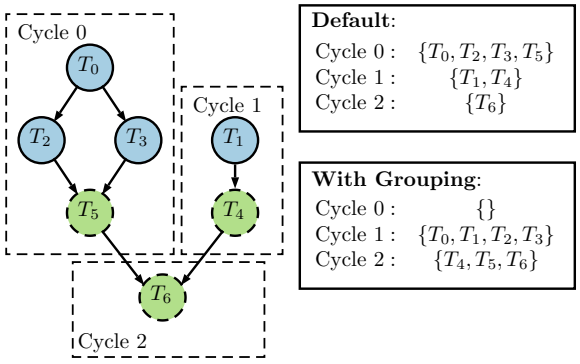

Figure 5: Illustration of Grouping. A task graph with nodes that generate requests T_n is depicted on the left, with the dashed boxes indicating requests visible to `Horovod` at 3 subsequent cycles. The nodes are colored to depict assignment to two groups (blue/solid borders and green/dashed borders). By default, a worker will submit all requests observed in a cycle to be processed/executed which can yield unbalanced sets of requests. With grouping enforced, requests are only submitted when complete groups are available.

3.2 Grouping

The response cache described in the previous section addresses inefficiencies in the Horovod control plane. In this section, we describe a method to improve the data plane performance of Horovod through explicit grouping of AllReduce operations. In particular, we introduce a feature to Horovod that enables users to submit grouped collective operations, allowing explicit control over Horovod's tensor fusion (§2.2.2).

As is shown in Figure 5, in place of submitting individual collective requests per tensor, a user can submit a grouped collective (e.g. `hvd.grouped_allreduce`) for multiple tensors. Collective requests submitted within a group are treated as a single request in Horovod's control plane; that is, no request in the group is considered ready for the data plane until all requests in the group are submitted. As a result, the tensors within a group are guaranteed to be processed by the data plane during the same cycle and fused, along with any other responses ready for execution during the cycle.

This new grouping mechanism can be used to control how gradient AllReduces are scheduled during an iteration. In particular, the gradient AllReduce requests for a single iteration can be assigned to one or more groups to explicitly control the fused communication buffer sizes that Horovod

generates for gradient reduction, avoiding issues that can arise using the default dynamic fusing strategy as described in Section 2.2.2. To ease use, this functionality is exposed to users via a new argument, `num_groups` to Horovod's high-level `DistributedOptimizer` wrapper. By setting this argument, the set of gradient tensors to be AllReduced within the iteration are evenly distributed into the number of groups specified. In the implementation described here, the gradients lists are split into groups of equal number of tensors, without consideration of buffer size.

Beyond this basic splitting, advanced users can achieve more optimal data plane communication performance by manually tuning the distribution of gradient tensors across the groups, to target fusion buffer sizes for improved network efficiency and/or achieving better overlap of communication and computation. We discuss the performance with different grouping configurations in Section 4.

We note that the framework native communication libraries like torch.DDP also support gradient fusion/bucketing and expose options to split gradient reduction into groups of approximately fixed message size. These native implementations generally leverage access to framework-level details, like information about the constructed model, to form these groups. As Horovod does not have access to these framework-level details directly, this grouping mechanism provides a means to provide such information via associating sets of tensors coming from the model to groups.

4 Experiment

4.1 Environment Setup

Hardware. We performed all experiments on Summit supercomputer [27] at the Oak Ridge Leadership Computing Facility. As the 2nd fastest supercomputer in the world, Summit is a 148.6 petaFLOPS (double precision) IBM-built supercomputer, consisting of 4,608 AC922 compute nodes with each node equipped with 2 IBM POWER9 CPUs and 6 NVIDIA V100 GPUs. Summit is considered as ideally suited for Deep Learning workloads, due to its node-local NVMe (called burst buffer) and Tensor Cores on V100 for faster low-precision operations. Moreover, its NVLink 2.0 and EDR InfiniBand interconnect provides 50 GB/s and 23 GB/s peak network bandwidths for intra-node and inter-node communication.

Software. The techniques proposed in this work are implemented based off Horovod v0.15.2 and have been incorporated in v0.21.0. We measured the performance with two communication backends, including NCCL v2.7.8 and Spectrum MPI (a variant of OpenMPI) v10.3.1.2. To evaluate the performance of our proposals across DL frameworks and to compare against the state-of-the-art communication libraries, we integrated our solutions in Horovod with TensorFlow (v2.3.1) and PyTorch (v1.6.0). We compared our solutions to tf.distribute in TensorFlow v2.4 (TensorFlow supports grouping since

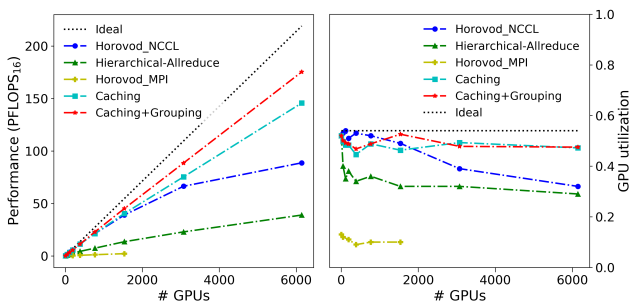

Figure 6: Performance and GPU utilization of Horovod's strategies. We compare our new techniques to the existing Horovod implementations using STEMDL (see Figure 3).

v2.4), torch.DDP in PyTorch v1.6.0, and BytePS (v0.2.5). In particular, BytePS is a deep learning framework that adopts PS (parameter server) as its communication model. BytePS is considered as an alternative to Horovod in a cloud environment. For tf.distribute and torch.DDP, we conducted the experiments with both NCCL and MPI; for BytePS, we conducted experiments simply with NCCL as BytePS does not support MPI. We configure BytePS in co-locate mode with one server and one worker per Summit node. We choose this configuration because it is recommended by the BytePS team as the best practice for high-performance computing (HPC) [1]. Moreover, we evaluated the scalability of our techniques with STEMDL, where the results are from an earlier incarnation of this work based on Horovod v0.15.2 built with NCCL v2.4, but the conclusions are similar.

Workloads. We evaluated our solution on GPU-based workloads. Starting with the STEMDL workload (message size 880MB per GPU), we compared our new techniques to the existing Horovod strategies (see Figures 3 and 6) with TensorFlow. We then broadened the experiments to compare with tf.distribute, torch.DDP, and BytePS on Resnet50 (102MB per GPU). Finally, we demonstrated our approach on ResNet50 and two more popular networks: EfficientNet-B0 (21MB per GPU) and VGG19 (574MB per GPU). We limit our interest in communication and use random synthetic data (of dimension (224, 224, 3)) as input to avoid impacts of I/O performance on the results. The training is in single precision with batch size of 64. We conducted the scalability experiments on the production code STEMDL using TensorFlow. We briefly discuss STEMDL in Section 2.3, report its source code in a GitHub repository (listed in Availability) and leave detailed documentation in Section 3 of the supplementary materials.

4.2 Evaluations on Horovod's Strategies

This section evaluates the performance of various strategies in Horovod. We compare the performance of caching and grouping to the existing strategies across scales. Figure 6 reports the results, in which we follow the definitions about the

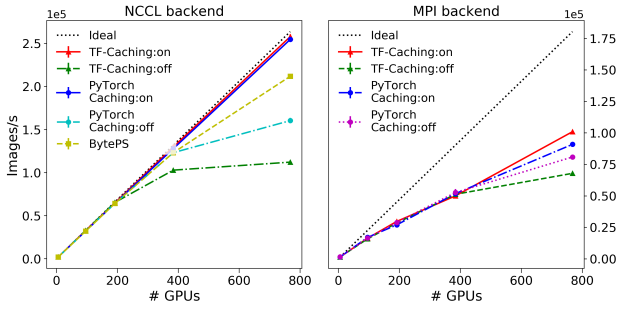

Figure 7: Performance of caching on ResNet50. We evaluate Horovod with caching enabled and disabled with both NCCL (left) and MPI (right) backends, and also compare the results to the performance of BytePS with NCCL (left).

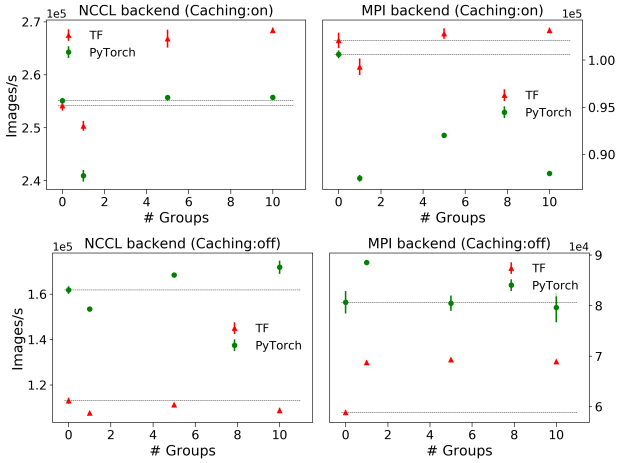

Figure 8: Performance of grouping on ResNet50. We evaluate Horovod with varied grouping configurations on 768 GPUs with caching enabled (top) and disabled (bottom) and with NCCL (left) and MPI (right) backends.

existing strategies given in Section 2.3 and name the results of our techniques as `Caching` (cached-based coordination enabled) and `Caching+Grouping` (both caching and grouping enabled), respectively. Similar to Figure 3, we focus on analyzing performance (left subfigure) and GPU utilization (right subfigure). Here, performance refers to the the floating-point operations performed per second (FLOPs). It is clear that our solutions outperform the existing strategies across scales consistently. When increasing the number of GPUs in use, the performance advantage grows rapidly. In particular, at the scale of 6000 GPUs, `Caching+Grouping` and `Caching` obtain 1.97× and 1.64× GPU performance improvement, and equally 1.48× utilization improvement, over the Horovod baseline in `NCCL-AllReduce`. Accelerated by our techniques, 175 petaFLOPS in FP16 precision (more detailed discussion can be seen in supplementary materials Section 2) can be delivered with less than a quarter of Summit.

We conclude that our techniques achieve better performance than the existing strategies, especially at scale.

4.3 Evaluations across Frameworks and Communication Libraries

Next, we evaluate caching and grouping with both TensorFlow and Pytorch, and compare our techniques to tf.distribute, torch.DDP, and BytePS.

4.3.1 Caching and Grouping across Frameworks

We first analyze the caching performance on Horovod with TensorFlow and Pytorch. Figure 7 presents the results. It suggests that, for the results with both NCCL and MPI, the caching-enabled Horovod (`TF-Caching:on` and `PyTorch Caching:on`) first delivers equally good performance; and when increasing the number of GPUs to 384 and more, the caching-enabled Horovod delivers better performance consistently with both TensorFlow and Pytorch. In particular, compared to the caching-disabled Horovod (`TF-Caching:off` and `PyTorch Caching:off`) with NCCL on 768 GPUs, the caching strategy achieves 2.5× (`TF-Caching:on`) and 1.6× (`PyTorch Caching:on`) performance improvement, respectively. Compared to the caching-disabled Horovod with MPI on 768 GPUs, the caching strategy achieves 1.53× and 1.15× performance improvement, respectively.

Figure 7 also presents the performance of BytePS (`BytePS`). It is shown clearly that BytePS delivers better performance than the cache-disabled Horovod consistently, and delivers equally good performance as the caching strategy does on the range of 6 GPUs — 384 GPUs, and delivers 20% lower performance than the caching strategy does on 768 GPUs. This suggests that, at larger scales, BytePS exhibits the scalability issue in typical HPC settings such as Summit. We leave the further study on the performance of BytePS on HPC clusters as future work.

Next, we report the grouping benefit in Figure 8. In the case with caching enabled (`Caching:on`) , comparing to the case without grouping (# groups = 0), the training throughput on 768 GPUs with Horovod (NCCL backend) obtains a decent 5% boost with 5 or 10 tensor groups for TensorFlow, although the gain for PyTorch is less significant. For the much slower MPI backend, the improvement becomes marginal or negative. When the caching is turned off (`Caching:off`) , there is a performance boost for PyTorch with the optimal group size, while for TensorFlow, it benefits mostly from grouping on the MPI backend. This indicates complicated interactions between the grouping and caching optimization.

To obtain a better understanding on the grouping behavior under different frameworks and communication fabrics, we plot the timing breakdown in Horovod for a 768-GPU training in Figure 9. For each iteration, the timing consists of two parts: coordination (control plane) and AllReduce (data plane). The timing for the AllReduce portion is further

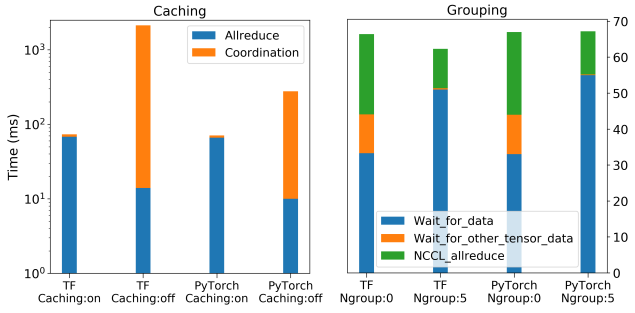

Figure 9: The inner timing breakdown in Horovod (NCCL backend) for a 768-GPU training with caching enabled and disabled (left) and grouping (# groups = 5) (right), respectively, during the training of ResNet50.

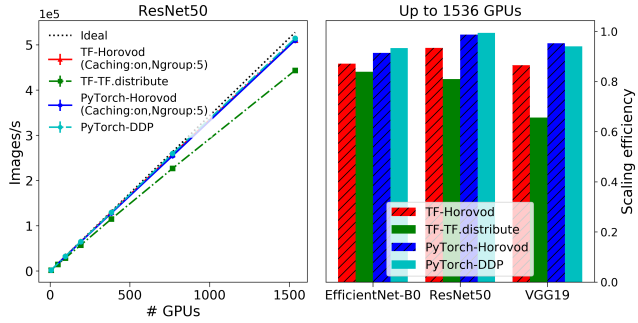

Figure 10: Scaling comparisons among Horovod, tf.distribute, and torch.DDP for the training of EfficientNet-B0, ResNet50, and VGG19. Training throughput (images/s) of ResNet50 (left). Scaling efficiency using up to 1536 GPUs (right).

split into wait (denoted [3] in Horovod as `WAIT_FOR_DATA` and `WAIT_FOR_OTHER_TENSOR_DATA` for time on waiting for framework to deliver gradient data and other data in the same fused collective, respectively) and actual communication (NCCL AllReduce). The case is slightly complicated for grouping. On one hand, the NCCL AllReduce time is almost cut in half because the grouped messages (orders of 10 MB) can better utilize network bandwidth; on the other hand, the wait time increases due to the coordination of groups. The overall performance of grouping depends on the trade-off between the aforementioned 2 factors. Too small number of groups (larger message and longer wait time) or too slow communication fabric (smaller or no gain in larger message communication) may result in worse performance with grouping, as indicated in Figure 8.

4.3.2 Evaluations across Communication Libraries

With both caching and grouping enabled, we compare the scaling efficiency of Horovod with tf.distribute and torch.DDP. To conduct a fair comparison, we ran all three libraries using a NCCL backend, and configure tf.distribute to use its AllReduce mode (`MultiWorkerMirroredStrategy`), similar to Horovod and torch.DDP. In contrast to the experiments with TensorFlow v2.3.1 reported in the previous sections, this section contains experiments run using tf.distribute in TensorFlow v2.4 as it supports a comparable grouping feature and is a more recent release. Moreover, we disabled the `broadcast_buffers` option in torch.DDP to ensure that no additional collective operations outside the gradient AllReduces are performed during testing. We set the bucket size/pack size for grouping in torch.DDP and tf.distribute to 25MB as it is the default configuration for torch.DDP.

We present the results in Figure 10. As is shown clearly in the left subfigure, using up to 1536 GPUs, Horovod delivers 93% and 96% of scaling efficiencies with TensorFlow and PyTorch, respectively, while tf.distribute and DDP achieve 81% and 97% of the efficiencies, respectively. To further illustrate the scaling on different communication volumes, we plot the scaling efficiency for EfficientNet-B0, ResNet50, and VGG19 (right subfigure). Our approach shows an average of 12% better scaling than tf.distribute and a comparable performance to DDP, across model sizes, and the advantage becomes bigger as communication volume increases.

To obtain a better understanding of the performance of the three libraries, we profiled the training of ResNet50 with the libraries using Nsight Systems [4] (an NVIDIA profiling tool) and observed how well the AllReduce operations overlap with computation within a training iteration for each library. The results (see details in supplementary materials Section 4) show that all three libraries group tensors for AllReduce to a similar number of large buffers per iteration (4 or 5). In particular, we observed >95% of AllReduce overlapped with computation when using Horovod and torch.DDP, and the number dropped to ∼75% when using tf.distribute.

We conclude that our solution performs well with both TensorFlow and PyTorch. Moreover, it delivers comparable and/or better performance than tf.distribute and torch.DDP, especially for large communication volumes.

4.4 Scaling Analysis on Production Code

This section evaluates the scaling efficiency of our solutions using a scientific DNN training code, STEMDL. The purpose of the section is to demonstrate a use case that stresses the communication layer of DL training at extreme scales (e.g. 27k GPUs). Our expectation is that if a communication implementation can scale well in this scenario, it should be well suited to many other workloads operating with far fewer tasks. Beyond scaling efficiency, we also evaluate the power consumption and overall performance of the production runs of STEMDL on the fully-scaled Summit, and leave the detailed documentation (e.g., the metrics and evaluations) to Section 2 in the supplementary materials, due to space limitations.

Figure 11 presents the scaling results. With both caching and grouping enabled, Horovod achieves a scaling efficiency

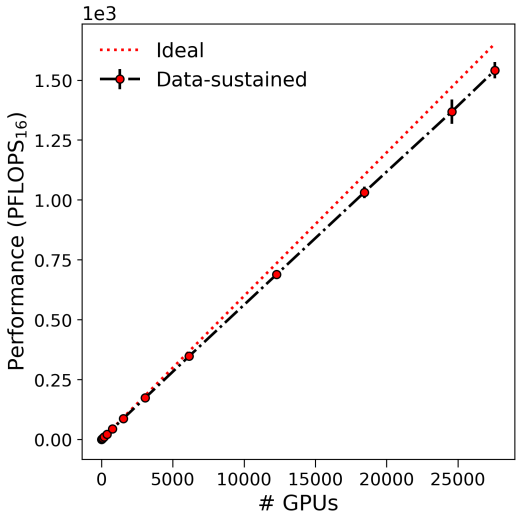

Figure 11: Scaling efficiency of STEMDL using up to 27,600 GPUs, the entire Summit.

of 0.93 at 27,600 GPUs and reach a sustained performance of 1.54 exaflops (with standard error +- 0.02) and a peak performance of 2.15 exaflops (with standard error +-0.02) in FP16 precision. Moreover, on a single GPU, our proposals attain 59.67 and 83.92 teraflops as the sustained and peak performance, respectively. It suggests that each GPU achieves 49.7% and 70% of the theoretical peak performance of a V100 (120 teraflops) as its sustained and peak performance. To the best of our knowledge, it exceeds the single GPU performance of all other DNN trained on the same system to date.

We conclude that our techniques can attain near-linear scaling on up to 27,600 GPUs.

5 Related Work

Other than collective AllReduce, another popular scheme for data parallelism is parameter server. Incorporated with many acceleration techniques such as hierarchical strategy, priority-based scheduling, etc, BytePS [12, 24] has shown better scaling performance than Horovod in a cloud environment where parameter servers run on CPU-only nodes, because the network bandwidth can be more efficiently utilized[4]. We compared our solutions with BytePS on a typical HPC setting and the results (see Figure 7) show that our techniques perform better in such settings.

One promising direction is to further reduce the communication volume via compression [8, 11, 26, 35, 36], decentralized learning [13, 14, 19], or staled/asynchronized communication [9, 10, 20]. The compression techniques include quantization, sparsification, sketching, etc, and the combined method [22] has shown 2 orders of magnitude in communication volume reduction without loss of accuracy. For decentralized learning, depending on the communication graphs for model replicas, the communication complexity is reduced to $O(Deg(graph))$ independent of scale. Staled/asynchronized communication can boost the communication performance by relaxing the synchronization requirement across model replicas, which usually comes with some cost in model convergence. These developments are orthogonal to our approach, and in principle, our techniques can apply on top of them.

Beyond proposals for improving collective communication in DNN training. Kungfu [23] is proposed to auto-tune the parameters in both DNN models and DL frameworks based on runtime monitoring data. This effort is complementary to ours: we propose techniques in Horovod with introduction of parameters that may benefit tremendously from appropriate tuning. Another significant recent study [28] proposed Drizzle to improve large scale streaming systems with group scheduling and pre-scheduling shuffles. Similar to our approach, Drizzle reused scheduling decisions to reduce coordination overhead across micro-batches. But different to our decentralized coordination proposal, Drizzle amortized the overhead of centralized scheduling.

6 Conclusion

We have shown that by introducing a new coordination strategy and a grouping strategy we exceed the state of the art in scaling efficiency. This opens up, in particular, opportunities in exploiting the different levels of parallelism present in many systems (e.g. intra-node vs inter-node) such as Summit to train even larger DNN models.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Shivaram Venkataraman, for their invaluable comments that improved this paper. This research was partially funded by a Lab Directed Research and Development project at Oak Ridge National Laboratory, a U.S. Department of Energy facility managed by UT-Battelle, LLC. An award of computer time was provided by the INCITE program. This research also used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

Availability

The proposed techniques have been upstreamed to the Horovod main distribution [3]. The code for full Summit distributed training and the software for data generation are made public [16, 17].

[4] In current ring-based AllReduce (as implemented in NCCL), each model replica sends and receives $2(N-1)/N$ times gradients (N being number of GPUs), so the total message volume transferred in network per model is 2x of the gradient volume for large N.

References

[1] BytePS Best Practice. https://github.com/bytedance/byteps/blob/master/docs/best-practice.md.

[2] Gloo. https://github.com/facebookincubator/gloo.

[3] Horovod. https://github.com/horovod/horovod.

[4] Nvidia Nsight. https://developer.nvidia.com/nsight-systems.

[5] PyTorch. https://pytorch.org/.

[6] tf.distribute in TensorFlow. https://www.tensorflow.org/api_docs/python/tf/distribute.

[7] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[8] Naman Agarwal, Ananda Theertha Suresh, Felix Yu, Sanjiv Kumar, and H. Brendan McMahan. cpSGD: Communication-efficient and differentially-private distributed SGD. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*, 2018.

[9] Suyog Gupta, Wei Zhang, and Fei Wang. Model accuracy and runtime tradeoff in distributed deep learning: A systematic study. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, 2017.

[10] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS'13)*, 2013.

[11] Nikita Ivkin, Daniel Rothchild, Enayat Ullah, Vladimir braverman, Ion Stoica, and Raman Arora. Communication-efficient distributed SGD with sketching. In *Advances in Neural Information Processing Systems (NIPS'19)*, 2019.

[12] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, 2020.

[13] Anastasia Koloskova*, Tao Lin*, Sebastian U Stich, and Martin Jaggi. Decentralized deep learning with arbitrary communication compression. In *Proceedings of the International Conference on Learning Representations (ICLR'20)*, 2020.

[14] Anastasia Koloskova, Sebastian U Stich, and Martin Jaggi. Decentralized stochastic optimization and gossip algorithms with compressed communication. In *Proceedings of the 36th International Conference on Machine Learning (ICML'19)*, 2019.

[15] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, Prabhat, and Michael Houston. Exascale deep learning for climate analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)*, 2018.

[16] Nouamane Laanait, Michael A Matheson, Suhas Somnath, Junqi Yin, and USDOE. STEMDL. https://www.osti.gov//servlets/purl/1630730, 9 2019.

[17] Nouamane Laanait, Junqi Yin, and USDOE. NAMSA. https://www.osti.gov//servlets/purl/1631694, 8 2019.

[18] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch distributed: Experiences on accelerating data parallel training. *Very Large Data Bases Conference (VLDB'20)*, 2020.

[19] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. Pipe-SGD: A decentralized pipelined SGD framework for distributed deep net training. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*, 2018.

[20] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS'15)*, 2015.

[21] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous decentralized parallel stochastic gradient descent. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, 2018.

[22] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *Proceedings of the International Conference on Learning Representations (ICLR'18)*, 2018.

[23] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, 2020.

[24] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, 2019.

[25] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[26] Ryan Spring, Anastasios Kyrillidis, Vijai Mohan, and Anshumali Shrivastava. Compressing gradient optimizers via count-sketches. In *Proceedings of the 36th International Conference on Machine Learning (ICML'19)*, 2019.

[27] Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, Al Geist, James Sexton, Jim Kahle, Christopher J. Zimmer, Scott Atchley, Sarp Oral, Don E. Maxwell, Veronica G. Vergara Larrea, Adam Bertsch, Robin Goldstone, Wayne Joubert, Chris Chambreau, David Appelhans, Robert Blackmore, Ben Casses, George Chochia, Gene Davison, Matthew A. Ezell, Tom Gooding, Elsa Gonsiorowski, Leopold Grinberg, Bill Hanson, Bill Hartner, Ian Karlin, Matthew L. Leininger, Dustin Leverman, Chris Marroquin, Adam Moody, Martin Ohmacht, Ramesh Pankajakshan, Fernando Pizzano, James H. Rogers, Bryan Rosenburg, Drew Schmidt, Mallikarjun Shankar, Feiyi Wang, Py Watson, Bob Walkup, Lance D. Weems, and Junqi Yin. The design, deployment, and evaluation of the coral pre-exascale systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'18)*, 2018.

[28] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, 2017.

[29] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. Characterizing output bottlenecks in a supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'12)*, 2012.

[30] Bing Xie, Jeffrey Chase, David Dillow, Scott Klasky, Jay Lofstead, Sarp Oral, and Norbert Podhorszki. Output performance study on a production petascale filesystem. In *HPC I/O in the Data Center Workshop (HPC-IODC'17)*, 2017.

[31] Bing Xie, Yezhou Huang, Jeffrey Chase, Jong Youl Choi, Scott Klasky, Jay Lofstead, and Sarp Oral. Predicting output performance of a petascale supercomputer. In *Proceedings of the International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC'17)*, 2017.

[32] Bing Xie, Sarp Oral, Christopher Zimmer, Jong Youl Choi, David Dillow, Scott Klasky, Jay Lofstead, Norbert Podhorszki, and Jeffrey S Chase. Characterizing output bottlenecks of a production supercomputer: Analysis and implications. *ACM Transactions on Storage (TOS'20)*, 2020.

[33] Bing Xie, Zilong Tan, Phil Carns, Jeff Chase, Kevin Harms, Jay Lofstead, Sarp Oral, Sudharshan Vazhkudai, and Feiyi Wang. Applying machine learning to understand write performance of large-scale parallel filesystems. In *the 4TH International Parallel Data Systems Workshop (PDSW'19)*, 2019.

[34] Bing Xie, Zilong Tan, Phil Carns, Jeff Chase, Kevin Harms, Jay Lofstead, Sarp Oral, Sudharshan S Vazhkudai, and Feiyi Wang. Interpreting write performance of supercomputer I/O systems with regression models. In *Proceedings of the 36th IEEE International Parallel and Distributed Processing Symposium (IPDPS'21)*, 2021.

[35] Min Ye and Emmanuel Abbe. Communication-computation efficient gradient coding. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, 2018.

[36] Yue Yu, Jiaxiang Wu, and Longbo Huang. Double quantization for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems (NIPS'19)*, 2019.