Alleviating Load Imbalance in Data Processing for Large-Scale Deep Learning

Sarunya Pumma,[*] Daniele Buono,[†] Fabio Checconi,[†] Xinyu Que,[†] and Wu-chun Feng[*]

[*] Virginia Tech, Blacksburg, VA, USA; {sarunya, wfeng}@vt.edu
[†] IBM T.J. Watson, Yorktown Heights, NY, USA; {dbuono, fchecco, xque}@us.ibm.com

***Abstract*—Despite its growing importance, scalable deep learning remains a difficult challenge. Scalable deep learning is constrained by many factors, including those deriving from *load imbalance*. For most deep-learning software systems, multiple data-processing components—including neural network training, graph scheduling, input pipeline, and gradient synchronization—execute simultaneously and asynchronously. Such execution can cause the various data-processing components to contend with one another for the hardware resources, leading to severe load imbalance and, in turn, degraded scalability. In this paper, we present an in-depth analysis of state-of-the-art deep-learning software, TensorFlow and Horovod, to understand their scalability limitations. Based on this analysis, we propose four novel solutions that minimize resource contention and improve performance by up to 35% for training various neural networks on 24,576 GPUs of the Summit supercomputer at Oak Ridge National Laboratory.**

***Index Terms*—scalable deep learning, load imbalance, resource contention, TensorFlow, Horovod**

I. Introduction

Neural network learning has been around for decades as a method to classify complex data and trends, although it has only recently gained significant traction [15]. One of the main reasons for its recent popularity is the advancement of hardware technology [6, 7, 16, 20, 26, 38] that enables *deep* neural network learning (i.e., deep learning or DL) with massive quantities of data. On top of this DL hardware, many DL software frameworks have been rapidly developed to incorporate the cited trends. Most modern DL frameworks, such as TensorFlow [1], allow for parallel and distributed training. However, these frameworks suffer from scalability limitations, particularly on large supercomputing systems.

Most recent efforts to improve the scalability of DL have targeted network I/O optimization, that is, improving gradient synchronization [4, 35, 39]. Consequently, a number of communication plugins exist for parallel and distributed DL. Horovod [25] is one of the most widely used communication libraries because of its ease of use and good out-of-the-box performance. However, Horovod's inability to scale to large supercomputing systems is a known problem [33].

In this paper, we first study the scalability limitation in TensorFlow with Horovod (henceforth referred to as TensorFlow/Horovod) on large supercomputing systems and analyze the root cause of such limitation. Our analysis shows that the scalability limitation is not caused by the native performance of the hardware or software ecosystem itself but, instead, is an artifact of subtle resource contention issues that slows down the execution of some processes thus resulting in ***straggler processes*** or ***imbalance*** in the amount of time spent on computing by the different processes (i.e., load imbalance). The parallel DL ecosystem comprises multiple data-processing components, which typically run simultaneously and asynchronously. We observe that poor coordination between these components can cause ***high hardware resource contention***, which leads to load imbalance and prevents DL training from achieving high scalability on large-scale systems.

Because of the graph-based computational model used in modern DL frameworks, such load imbalance occurs only in portions of the graph where the resource demand is higher than the available hardware resources. This imbalance then propagates the resource contention into future iterations of the computation, creating further imbalance and further slowing down the overall computation. Due to the subtlety of this problem, the DL community continues to overlook this scalability limitation in DL frameworks.

Based on our analysis, we propose, design, and implement four techniques that enable different data-processing components to share the available computational resources more effectively in TensorFlow/Horovod. We then evaluate the performance of our proposed solutions in training several real-world deep neural networks and demonstrate improvements of up to 35% when using 24,576 GPUs of the Summit supercomputer at Oak Ridge National Laboratory [13].

It is important to note that this paper is *not* about improving the accuracy of the training itself but rather about understanding and alleviating scalability issues in large-scale DL. As such, we leverage well-known accuracy-improvement techniques showcased in [36, 37]—where the authors demonstrate scaling to batch sizes of around 32,768 while sustaining the best known accuracy of $\sim$75% for ImageNet training—to tune our neural networks. On a related note, while [23] claims to be able to sustain good accuracy with batch sizes as large as 131,072, we have not been able to reproduce that claim. Nevertheless, we note that there is significant ongoing research in the community to allow DL workloads to use very large batch sizes without losing accuracy [11, 28, 34], and that these batch size numbers are expected to go up dramatically in the next few years.

The rest of the paper is organized as follows. In Section II, we present background information on data processing in parallel DL. In Section III, we analyze the performance of TensorFlow/Horovod and their scalability issues. In Section IV, we describe the design and implementation of our proposed solutions. A detailed evaluation of our solutions while training

This work was done when Sarunya Pumma was a summer intern at IBM T.J. Watson.

various deep neural networks is presented in Section V. Other literature related to this paper is presented in Section VI. We summarize our work in Section VII.

II. Data Processing in Parallel Deep Learning

Deep neural network (DNN) learning is a complex computational method that consists of multiple data-processing components. Most modern DL frameworks assign these components to run on the available computational devices, e.g., CPUs and GPUs, simultaneously and asynchronously so as to increase resource utilization and computational throughput. Without proper coordination, however, these data-processing components compete with each other for resources, such as CPU cycles, memory bandwidth, network bandwidth, and access to the direct memory-access (DMA) engine.

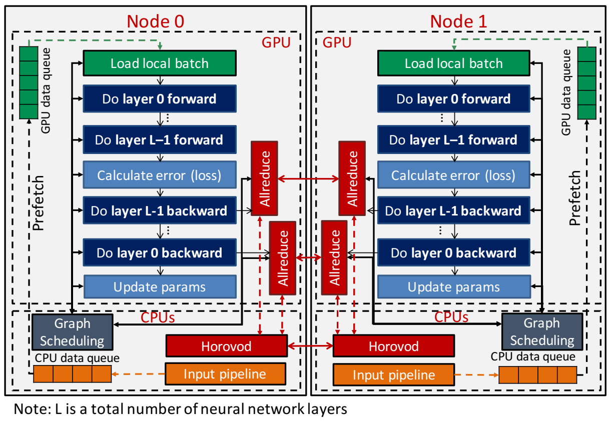


Fig. 1. Data-processing components of deep learning.

In this paper, we adopt a multilevel parallel DL model that provides *data parallelism* across nodes as well as within the node via multiple GPUs on each node. In other words, a full replica of the DNN is trained with a different batch of input data on each GPU. Each GPU then uses *model parallelism* to further parallelize the DL training. Figure 1 shows the data-processing components of our data-parallel environment. In the figure, we show only one GPU per node for simplicity, but the actual system that we use in our experiments has multiple GPUs per node. The model comprises four main data-processing components:

1) **Graph Scheduling (occurs on the CPUs):** Each kernel/operation in the DNN is dispatched to run on the GPU by the graph scheduler that is driven by the CPU threads. Any delay in graph scheduling can slow the DNN training.

2) **Neural Network Training (occurs on the GPUs):** The core computation associated with the DNN training (i.e., forward and backward computations) occurs on the GPUs.

3) **Gradient Synchronization (occurs on both the CPUs and GPUs):** The gradient synchronization between all GPUs is performed via the Allreduce operation during the backward computation of the training. Similar to other operations, the Allreduce operation is dispatched to run on the GPU by the graph scheduler. (Although the Allreduce operation in Figure 1 is depicted as executing on the GPUs, it uses both CPU and GPU resources.) The gradient transfers are scheduled and managed by Horovod, more details of which are presented in Section III-B.

4) **Input Pipeline Processing (occurs on the CPUs):** Input pipeline processing in the DL system involves a number of steps including file I/O, data shuffling, data augmentation, data prefetching, and host-to-device data transfer. In our experiments, we enable data batch prefetching and pipelining to avoid data-movement bottlenecks that might occur.

Of these various data-processing components, the gradient synchronization is of particular interest because of its dependence on both CPU and GPU resources. Without sophisticated coordination, it can potentially compete for both CPU and GPU resources with the other data-processing components described above. Thus, in this work, we focus on minimizing the interactions between the gradient synchronization component and the other data-processing components on the CPUs (i.e., graph scheduling and input pipeline processing).

III. TensorFlow/Horovod Performance Analysis

Here we profile and analyze the data processing components in TensorFlow/Horovod on a large-scale system.

A. Experimental Setup

We first articulate our experimental setup, from which we gather our experimental data for subsequent performance analysis.

Experimental testbed: We use Summit,[1] a supercomputer at Oak Ridge National Laboratory, as our experimental testbed. Summit has 4,608 nodes connected via Mellanox EDR 100-Gbps InfiniBand. Each node has two sockets of IBM POWER9 CPUs (total of 44 cores), six NVIDIA Tesla V100 GPUs, 512 GB of memory, and 1,500 GB NVMe (short for Non-Volatile Memory Express) which is used as our data storage. Each socket connects 22 cores, three GPUs, and 256 GB of memory. For each node, two cores (one per socket) are isolated for operating system tasks and cannot be used by user applications. We use six processes per node because Horovod restricts each process to drive at most one GPU. Each process has exclusive access to seven cores and one of the GPUs that is located on the same socket. Processes are limited to accessing only 256 GB of memory within their socket.

DL frameworks and software stack: We use TensorFlow v1.14.0-rc0 and Horovod v0.16.3 as our DL framework and communication subsystem, respectively. We use CUDA v10.1.168, CUDNN v7.6.1, and NCCL v2.4.7 (with GPUDirect RDMA) as the drivers for TensorFlow and Horovod. We compile the TensorFlow computation graph using the Accelerated Linear Algebra (XLA) compiler [2], which, in turn, disables any overlap between the computation and communication in the overall execution of a program. We performed experiments with both XLA-enabled and XLA-disabled benchmarks. In our experiments, the loss of performance due to the lack of overlap between the computation and communication

[1]Ranked one on Top500, November 2019 (https://www.top500.org/lists/2019/11)

was typically equal to the gain in performance from using XLA. So there was no significant difference in performance whether XLA was enabled or disabled. We chose to use the XLA-enabled version in most of our experiments for ease of analysis.

Benchmarks: We use *tf_cnn_benchmarks*,[2] one of the most well-known convolutional neural network (CNN) training benchmarks. All experiments use the ImageNet[3] dataset. Our analysis and evaluations are conducted on various CNNs, including five variants of ResNet [12] (sizes 18, 34, 50, 101, and 152), AlexNet [15], GoogLeNet [30], Inception-v3 [31], and VGG16 [27]. All our experiments use mixed-precision floating-point computations [14].

Experimental configuration: In all of our experiments, the training is run for 500 iterations with an additional 10 "warm-up" iterations that are not included in the performance results. All experiments are run three times, and the average performance is shown.

B. Understanding Horovod and its Background Thread

Horovod is a communication plugin for distributed DL frameworks, including TensorFlow [1], PyTorch [24], and MXNet [8]. It provides new communication operation classes with similar semantics as the native communication operations in these DL frameworks. Because of this, users can construct *computation graphs*, which are the representations of the DNNs, by simply replacing the native communication operations with Horovod operations.

Horovod relies on several highly optimized data-movement libraries, such as the Message Passing Interface (MPI) [32], NVIDIA Collective Communications Library (NCCL),[4] IBM Distributed Deep Learning Library (DDL) [9], Intel Machine Learning Scaling Library (MLSL) [29], and Facebook Gloo[5] for communication, thus allowing for better data-transfer performance and scalability. Each Horovod operation takes one input tensor and produces one output tensor. Typical DL computations require processing more than one tensor. Consequently, computation graphs generally contain multiple Horovod operations.

All Horovod operations are *nonblocking asynchronous*. They are *nonblocking* in that a Horovod operation will always return in a finite amount of time, irrespective of the state of other processes in the system. Specifically, when a Horovod operation is executed in the computation graph, the DL framework's graph scheduler thread enqueues a communication request into the Horovod's internal request queue and then returns. Horovod operations are *asynchronous* in that once the operation is enqueued, the DL framework is no longer responsible for its completion. The progress and completion of the operation are asynchronously handled by Horovod. To achieve this, within each operating system (OS) process,

[2]https://github.com/tensorflow/benchmarks.git

[3]http://www.image-net.org/challenges/LSVRC/2012/

[4]https://developer.nvidia.com/nccl

[5]https://github.com/facebookincubator/gloo

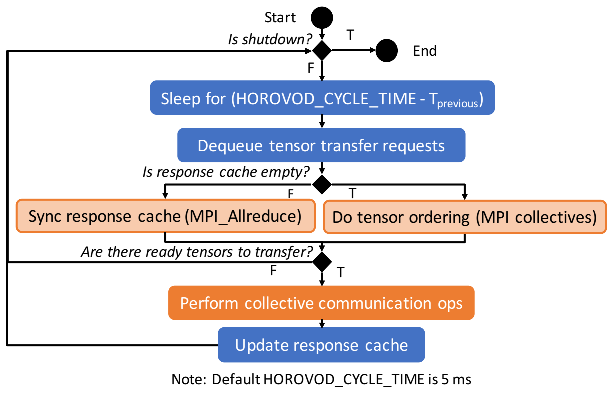

Start
Is shutdown?
T
End
F
Sleep for (HOROVOD_CYCLE_TIME - $T_{previous}$)
Dequeue tensor transfer requests
Is response cache empty?
F
T
Sync response cache (MPI_Allreduce)
Do tensor ordering (MPI collectives)
Are there ready tensors to transfer?
F
T
Perform collective communication ops
Update response cache
Note: Default HOROVOD_CYCLE_TIME is 5 ms

Fig. 2. Horovod background thread workflow.

Horovod creates a background thread whose primary purpose is to perform the data transfers associated with the various Horovod operations in that process. The background thread periodically checks the request queue (into which the graph scheduler had enqueued communication requests), issues data transfers for the tensors associated with the enqueued requests, and executes completion callbacks to the DL framework once the transfers are completed.

Tensor transfer requests associated with Horovod operations that have no dependencies with one another in the computation graph could be enqueued simultaneously, for example, by different graph scheduler threads processing the graph. Therefore, even when all processes are executing the same graph, the operations in the graph, including Horovod operations, could be executed out of order, thus making the order of data transfer requests nondeterministic. Because Horovod relies on other collective communication primitives, it has to ensure that data transfers for different tensors are performed in the same order on all processes. Consequently, the background threads on these processes have to perform an additional **tensor-ordering consensus protocol** to determine a globally consistent order of data transfers.

The provided implementation of the tensor-ordering consensus protocol in the Horovod background thread is unfortunately inefficient. Figure 2 shows the high-level workflow of this protocol. The background thread executes an infinite loop of progress checks on tensor data transfers. We refer to each such loop as a "cycle." To prevent the background thread from monopolizing a CPU core for progress checks, there is sleep time inserted between cycles. The sleep time is *HOROVOD_CYCLE_TIME* $- T_{previous}$, where *HOROVOD_CYCLE_TIME* is the maximum sleep time or the maximum cycle time threshold (which is a user input; default is 5 ms) and $T_{previous}$ is the execution time of the previous cycle excluding the sleep time. If $T_{previous}$ is larger than *HOROVOD_CYCLE_TIME*, the background thread will not sleep; otherwise, it will sleep, and upon waking up, the background thread dequeues the tensor transfer requests from the request queue and attempts to create a global ordering for them through a consensus protocol.

The consensus protocol is straightforward; one of the background threads is assigned as the "master background thread"

(MPI rank zero). All background threads use MPI collective operations to send the transfer request details of their ready tensors to the master background thread. The master background thread, in turn, looks through the list of ready tensors from all the background threads, forms an ordered list of tensors that are ready on all the background threads, and sends this list back to all the background threads. Once this tensor-ordering consensus protocol has completed, each background thread fuses its local tensors and performs data transfer based on the order received from the master background thread.

This tensor-ordering consensus protocol is heavyweight and often causes severe performance degradation, especially on the master background thread. To address this issue, recent versions of Horovod (since v0.16.2) have introduced a tensor-ordering cache called a "response cache," which can be reused across cycles. This cache, which is a data structure for storing tensor information and tensor order for future use, is initially empty. Once the tensor-ordering consensus protocol occurs, each background thread locally stores the tensor request information and ordering scheme in its response cache. In the next cycle, the response cache is *not* empty, so the heavyweight consensus protocol can be avoided, but the background threads still need to synchronize their caches via *MPI_Allreduce*, to determine which tensors in the cache are ready to be transferred because this list can change from cycle to cycle. In other words, the response cache reduces the amount of work done by the master background thread, but it does *not* (and cannot) remove the synchronization needed between the background threads.

An important aspect to understand here is that the cache synchronization is a "worst-case" requirement. Typically, the cache is *not* empty, and there are no tensors ready to be transferred. Thus, in most cycles, the background thread sleeps and then does an empty *MPI_Allreduce* for the cache synchronization. Because the arrival of tensor transfer requests is nondeterministic, each background thread still needs to participate in every *MPI_Allreduce* even if it has no new tensor transfer requests in order to prevent deadlocks.

C. Scalability Analysis

Figure 3 shows the results of weak scaling with TensorFlow/Horovod using the ResNet50 network (relative to linear scaling). Our baseline for comparison is XLA-enabled TensorFlow/Horovod (hereafter called TensorFlow/Horovod), but in this graph we also show XLA-disabled TensorFlow/Horovod performance for completeness. The data-processing throughput of TensorFlow/Horovod is approximately 3.3 times *worse* than linear scaling on 24,576 GPUs; that is, the scaling loss is 69.7%.

To identify the source of this scaling loss, we profile the GPU execution, as shown in Figure 4, and classify the overall time into two parts: computation (denoted by "Forward & backward pass execution time") and communication (denoted by "HorovodAllreduce time"). The figure shows that the computation time stays relatively constant as the number of GPUs increase but that the communication time increases nearly

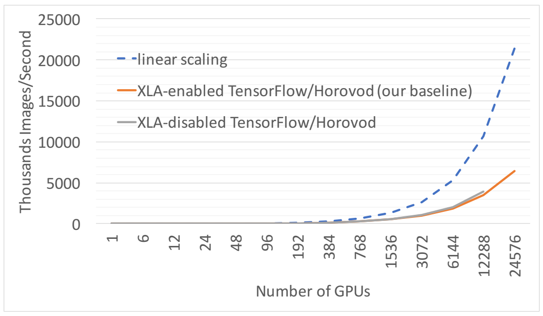


Fig. 3. Weak scaling of TensorFlow/Horovod compared with linear scaling.

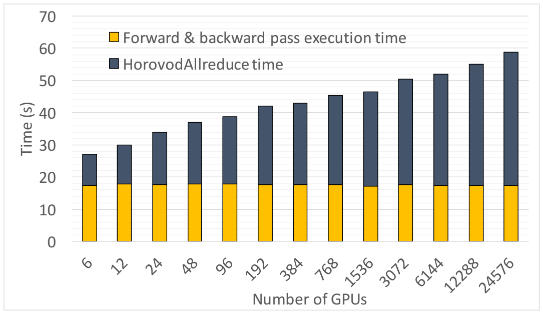


Fig. 4. TensorFlow/Horovod GPU time breakdown. (Using XLA disables any overlap between the computation and communication, as explained in Section III-A).

linearly. In the worst case, HorovodAllreduce consumes up to 70.3% of the overall GPU execution time, which accounts for virtually all of the scaling loss noted above.

Next, we analyze the GPU time during HorovodAllreduce, as shown in Figure 5. The figure shows the HorovodAllreduce time separated into three parts: (1) *ncclAllReduce*, where the GPU can be either idle (waiting for other GPUs to send data) or busy (calculating the summation of data), (2) *memory copy*, where the GPU is considered to be idle as it is using the DMA engine, but not the computation units, and (3) the *Horovod background thread overhead*, where the GPU is idle and waiting for the host to finish its work. Our profiling shows that the GPU is idle for at least 67% of the HorovodAllreduce time (i.e., summation of the background thread overhead and memory copy time). The majority of this idle time is due to the *Horovod background thread overhead*, which includes cycle latency (i.e., the sleep between cycles), tensor stalling (i.e., waiting for tensors to be ready for transfer on all processes), and tensor ordering.

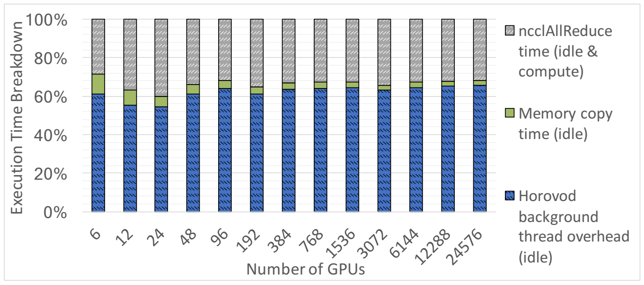


Fig. 5. TensorFlow/Horovod HorovodAllreduce time breakdown.

D. Investigating the Horovod Background Thread

As noted in Section III-B, the Horovod background thread spends most of its time alternating between sleeping and performing an often empty *MPI_Allreduce*. When there are tensors to be transferred, it calls the collective communication operations. The workflow of the Horovod background threads was designed for scenarios where all background threads are fairly synchronized. In such a scenario, as shown in Figure 6, the background threads spend most of their time sleeping. Thus, they would not compete for resources with other data processing components, and consequently, would not create any further performance imbalance in the computation. In reality, however, this is not always the case.

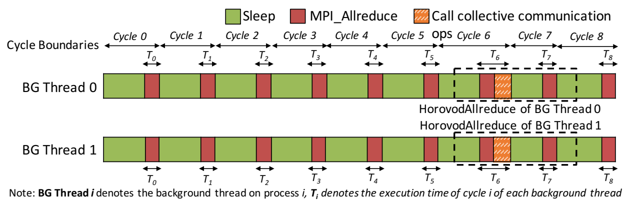

🟩 Sleep
🟥 MPI_Allreduce
▨ Call collective communication ops
Cycle Boundaries
Cycle 0
Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Cycle 8
T_0 T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8
BG Thread 0
HorovodAllreduce of BG Thread 0
HorovodAllreduce of BG Thread 1
BG Thread 1
Note: BG Thread *i* denotes the background thread on process *i*, T_i denotes the execution time of cycle i of each background thread

Fig. 6. A perfect synchronization of Horovod background threads.

In real DNN training, even when all processes are computing on exactly the same computation graph, there can be a slight imbalance in their execution time or the state of the various threads in the system (e.g., which threads are executing at a given point in time). Such load imbalance is expected but generally small and uninteresting. However, the *cascading effect* of such small imbalances is of particular interest as it makes up virtually all of the HorovodAllreduce time.

Consider a case with two processes, where both processes are computing on the same computation graph, but the state of the execution or that of the various threads is not exactly identical on both processes. For these processes, when the background threads are ready to be scheduled by the OS, the two background threads might have to wait for vastly different amounts of time to get scheduled. This difference in actual scheduling time depends on when the OS decides to preempt the other currently executing threads (i.e., the input pipeline and graph scheduling threads) and to execute the background thread. This wait time for preemption can be as high as tens of milliseconds on modern Linux versions. We call this scenario, where some background threads take longer to be scheduled than the other background threads, as "oversleep."

Figure 7 demonstrates the cascading effect of load imbalance that propagates from one cycle to the next. In the figure, *BG Thread i* denotes the background thread on process *i*. In *Cycle 0*, *BG Thread 0* arrives at the *MPI_Allreduce* function first and consequently takes longer to complete the operation because it is waiting for *BG Thread 1*, that oversleeps, to call *MPI_Allreduce*. In this case, *BG Thread 1* is a **straggler** thread.

In the next cycle (i.e., *Cycle 1*), *BG Thread 0*'s previous cycle time T_0 is larger than the maximum cycle time threshold, *HOROVOD_CYCLE_TIME*, and thus Horovod would not let it sleep at all, as described in Section III-B. *BG Thread 0* would then issue *MPI_Allreduce* right away. In contrast, T_0 of the straggler thread is smaller than *HOROVOD_CYCLE_TIME* causing it to sleep in *Cycle 1*. This action would cause the

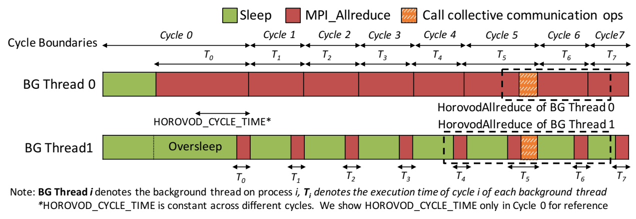

🟩 Sleep
🟥 MPI_Allreduce
▨ Call collective communication ops
Cycle Boundaries
Cycle 0
Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle7
T_0 T_1 T_2 T_3 T_4 T_5 T_6 T_7
BG Thread 0
HorovodAllreduce of BG Thread 0
HOROVOD_CYCLE_TIME*
HorovodAllreduce of BG Thread 1
BG Thread1
Oversleep
Note: BG Thread *i* denotes the background thread on process *i*, T_i denotes the execution time of cycle i of each background thread
*HOROVOD_CYCLE_TIME is constant across different cycles. We show HOROVOD_CYCLE_TIME only in Cycle 0 for reference

Fig. 7. Horovod background thread oversleep problem.

straggler thread to be delayed in reaching its *MPI_Allreduce* in the next cycle as well, further exacerbating the load imbalance.

In addition, while *MPI_Allreduce* waits for other processes to arrive, it **spin waits**, thus consuming CPU cycles and potentially slowing down other data-processing components, namely, input pipeline processing and graph scheduling. For input pipeline processing, the impact is minimal because the outcome of the input pipeline is used in the next training iteration (recall that data prefetching is enabled), and the delay does not stall the current iteration. For graph scheduling, however, this slowdown can cause the forward and backward computation on the GPU to be delayed. This delay causes some GPUs (e.g., the GPU associated with process 0 in Figure 7) to execute the gradient synchronization late, thus making the imbalance show up in the HorovodAllreduce time as the Horovod background thread overhead.

IV. Proposed Solutions

In this section, we present four solutions to alleviate the load imbalance in distributed DL processing, based on our performance analysis of TensorFlow/Horovod in Section III.

A. Horovod-GS: Global Sleep Time Optimization

The oversleeping of some background threads causes the processes to issue *MPI_Allreduce* at different points in time, as noted in Section III-D; in other words, the processes are "out of sync" in calling *MPI_Allreduce*. To address this issue, we propose *Horovod-GS*. In *Horovod-GS*, each background thread, instead of using just its local knowledge to figure out how long it needs to sleep, uses a **globally coordinated sleep time** to ensure that all processes use the same sleep time every N cycles. This prevents the load imbalance from propagating beyond N cycles. The intent here is to separate the load imbalance from the actual data transfer time in *MPI_Allreduce*.

Before sleeping, each background thread computes its local sleep time using the same formula as in the original Horovod (*HOROVOD_CYCLE_TIME*$-T_{previous}$). Then, all processes determine the globally minimum sleep time by using another *MPI_Allreduce*. Once the minimum sleep time is received, each background thread sleeps for the globally minimum sleep time. The rest of the workflow is the same as the original Horovod. This approach prevents the load imbalance effect from propagating to subsequent cycles.

As noted above, the second *MPI_Allreduce* that we introduce in *Horovod-GS* occurs once every N cycles. So, theoretically, a smaller value for N creates higher synchronization overhead while reducing load imbalance, whereas a large value of N provides the opposite tradeoff. Empirically, we found that $N = 1$ delivered the best performance on our system—although, depending on the dataset and the system, this value may need to be tuned appropriately.

B. Horovod-NBCS: Nonblocking Cache Synchronization

While Horovod-GS can reduce the load imbalance, it cannot completely prevent the background thread from competing for resources with other components. Specifically, the imbalance in the first *MPI_Allreduce* still remains and typically consumes the most time. Thus, the Horovod background thread still spends a significant amount of time occupying the CPU cores and competing for resources with the input pipeline and the graph scheduling components.

Thus, we propose Horovod-NBCS (i.e., nonblocking cache synchronization), where we seek to limit the time spent inside *MPI_Allreduce* in order to free up computational resources for the other two data-processing components. To do so, we leverage nonblocking MPI collective operations, specifically, *MPI_Iallreduce* and *MPI_Test*, for the response cache synchronization. While this approach does *not* avoid the out-of-sync problem between processes, the processes no longer compete for resources with other components because the time spent inside each MPI call is finite (as guaranteed by the MPI standard for all nonblocking operations) and typically small.

We note that, even though this approach uses nonblocking cache synchronization, it guarantees cache consistency by ensuring that the cache synchronization has completed before performing another one. Thus, the cache content of all background threads is identical in each cycle.

C. Horovod-SCP: Static CPU Resource Partitioning

With Horovod-SCP, we address the load imbalance problem via a simple static partitioning of resources. Specifically, to avoid contention between the different data-processing components, we partition the available cores into groups such that each group of threads that executes a different data-processing component gets a different set of cores. This guarantees that there is no contention between the different data-processing components, thus potentially alleviating load imbalance.

As we will see in Section V, Horovod-SCP successfully reduces the contention between different data-processing components to alleviate load imbalance. Despite the impressive performance gains, however, we view Horovod-SCP as a somewhat of a workaround. Specifically, while Horovod-SCP does alleviate the biggest cause for load imbalance, it comes with several shortcomings.

First, the static partitioning of CPU resources means that any variation in processing needs that arise during the execution of the DL workflow cannot be dynamically resolved. For instance, the background thread only needs to be active for a small part of the total execution, but having a dedicated core means that that core cannot be used for other data-processing components when the background thread is idle. This can impact the overall performance if the other data-processing components starve for CPU resources. Second, even with a dedicated core, load imbalance cannot be fully avoided. This is because, as described in Section III-D, even when all processes are computing on exactly the same computation graph, there can be a slight imbalance in their execution time. Because of this slight imbalance, how long each background thread spends in the *MPI_Allreduce* can be different, which would cause different threads to sleep for different amounts of time in the next cycle, which would further increase the load imbalance. Thus, future work will study and integrate dynamic partitioning, as appropriate.

D. Horovod-TOPO: Graph Topology Exploitation

While the previous solutions can help reduce the time the background thread spends competing for computational resources, they are still fundamentally limited by the way Horovod performs tensor ordering. In particular, they rely on the most generic possibility where the tensor transfers can be issued in any arbitrary order. However, this is not true in reality and over-generalizes the TensorFlow workflow.

TensorFlow uses a graph processing workflow, and the order in which tensors are issued depends on the graph structure. Tensors that are *logically concurrent* (i.e., belong to graph nodes with no dependency between them) can be issued in any order. When Horovod sees such a tensor request, it can wait for the other logically concurrent tensor requests to be issued without creating deadlock. In contrast, for two tensors whose corresponding graph nodes have a dependency between them, there is a guaranteed ordering where the second tensor cannot be issued before the first tensor operation has completed.

Thus, our Horovod-TOPO solution seeks to eliminate the original tensor ordering and the response cache synchronization by performing a one-time TensorFlow graph analysis. The core idea of Horovod-TOPO is to analyze the TensorFlow graph and the dependencies between the graph nodes to form a **partial logical ordering** of tensor data-movement requests. The generated ordering is a *logical ordering* because some tensor requests are logically concurrent and can be issued in any arbitrary order by the graph scheduling threads. The generated ordering is *partial* because the graph dependencies restrict the reordering of some tensor requests. Based on this partial logical ordering, we can then precompute a **tensor fusion scheme** that determines which tensors can be fused together so that the data-transfer requests can be larger, thus amortizing data-transfer overhead. Once the tensor fusion scheme is determined, it is stored within Horovod and utilized for all future computation iterations. Thus, this topological graph analysis needs to be done only once and never repeated.

In TensorFlow, normally only a subgraph is executed at any given time. A subgraph is identified by the user with "fetches," which are nodes in the graph whose outputs will be obtained from the execution, i.e., fetches are sink nodes of the subgraph. On the first execution of the subgraph, we assign an identification number (ID) to every Horovod operation in the subgraph. We then adopt a traditional reverse depth-first search algorithm to traverse from fetches to the root nodes to identify all Horovod operations and their dependencies. The time complexity of this algorithm is $O(V+E)$, where V and E are the number of nodes and the number of edges in the subgraph, respectively. The ID represents the chronological order in which the operations are executed. To account for operation dependencies, parent operations are assigned a smaller ID than are children operations. Among sibling operations, IDs are assigned based on the order that they are added to the graph. For parallel nonsibling operations, we assign IDs to the operations according to their depth in the subgraph. (If the depths are the same, the ID assignment is arbitrary.)

Together with the tensor order, we determine the tensor fusion model (i.e., which tensors should be fused before communicating) the first time that a subgraph is executed. We follow Horovod's original approach to fuse only HorovodAllreduce's tensors and to cap the fusion buffers at *HOROVOD_FUSION_THRESHOLD* (64 MB by default).

Figure 8 shows an example of our tensor ordering and tensor fusion. The fetches in this example are *Allreduces 1*, *3* and *4*. From the figure, we assign the parent operations to have a smaller ID than their children (e.g., *Allreduces 0* and *2*). The parallel nonsibling operations are assigned IDs based on their depth in the subgraph (e.g., *Allreduce 4* has a larger ID than *Allreduce 1*). *Allreduces 0* and *1* can be fused as they are logically concurrent, likewise for *Allreduces 3* and *4*. In contrast, *Allreduces 0* and *2* cannot be fused because they share a dependency. Likewise, *Allreduces 2* and *3* have to be in different fusion buffers.

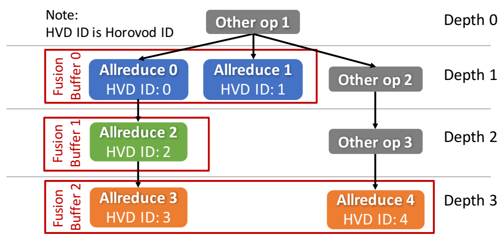


Fig. 8. Example of tensor ordering and tensor fusion.

During the actual execution of the TensorFlow graph, the background thread performs a tensor transfer only if the transfers of all tensors with smaller IDs have been issued as shown in Figure 9. We use one request array per fusion buffer for storing tensor transfer requests from the TensorFlow's graph scheduler threads. Each tensor request has its designated slot based on the Horovod ID in the request array. The queues and variables that are shared between the different threads are managed by using C++11 std::atomics, with some portions of the code optimized for IBM POWER9 CPU hardware atomics and memory ordering/consistency semantics.

While Horovod-TOPO cannot guarantee that the data transfer requests for logically concurrent tensors will always *arrive* in the same order, it does guarantee that (1) the data transfer requests are *issued* in the same order on the different background threads and (2) the background threads do not have to wait indefinitely before issuing a data transfer request. Background threads wait for additional tensor requests to be

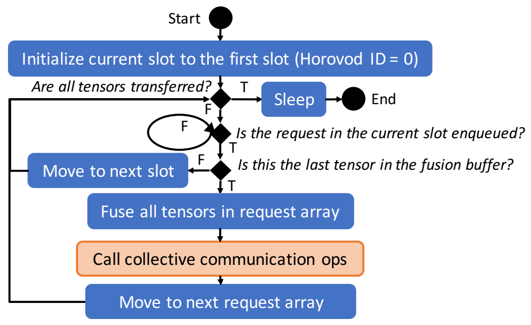

Start
Initialize current slot to the first slot (Horovod ID = 0)
Are all tensors transferred?
T
F
Sleep
End
F
Is the request in the current slot enqueued?
T
Move to next slot
F
Is this the last tensor in the fusion buffer?
T
Fuse all tensors in request array
Call collective communication ops
Move to next request array

Fig. 9. Horovod background thread's workflow in Horovod-TOPO.

issued only when the corresponding graph nodes are logically concurrent, and thus, the difference in their arrival is bounded by a finite amount of time. Due to space limitations, we elide the mathematical proof that shows that Horovod-TOPO is deadlock-free.

Because Horovod-TOPO assumes that the dependencies between the tensors to be transferred remains unchanged throughout the execution, it supports only static computation graphs. As TensorFlow is a declarative-style DL framework, that is a computation graph is defined prior to the computation takes place, it has traditionally only supported static graphs. Limited support for dynamic computation graphs has been recently added to TensorFlow v.2.0. In case of imperative-style DL frameworks like PyTorch, both static and dynamic computation graphs are supported, but they are dynamically created and freed during runtime, i.e., the graph structure is not known until the graph is executed. To leverage Horovod-TOPO for static graphs in PyTorch, Horovod-TOPO has to compute tensor dependencies during the first execution of the graph. Then, it can use such information to compute the tensor ordering and fusion schemes once the first iteration execution has completed.

V. Experiments and Results

We evaluate the performance of our proposed solutions and compare them against that of the original TensorFlow/Horovod implementation.

A. Evaluation of Proposed Solutions on ResNet50 Training

We first measure the weak-scaling performance of our various solutions using the ResNet50 network and the ImageNet dataset. In this experiment, we use a fixed local batch size (i.e., number of samples per GPU in one iteration) of 32; the global batch size increases proportionally with the number of GPUs. Figure 10 shows the data-processing throughput for the different approaches (i.e., original Horovod and our four optimized versions, namely GS, NBCS, SCP, and TOPO). All four optimizations outperform TensorFlow/Horovod by up to 10%, 16%, 18%, and 21%, respectively.

Figure 10 also shows that for runs with the number of GPUs $\geq$ 1,536, our global batch size becomes large enough that the inference accuracy drops. Despite this drop in accuracy, we highlight the following two points: (1) our proposed solutions are also applicable to smaller global batch sizes and deliver

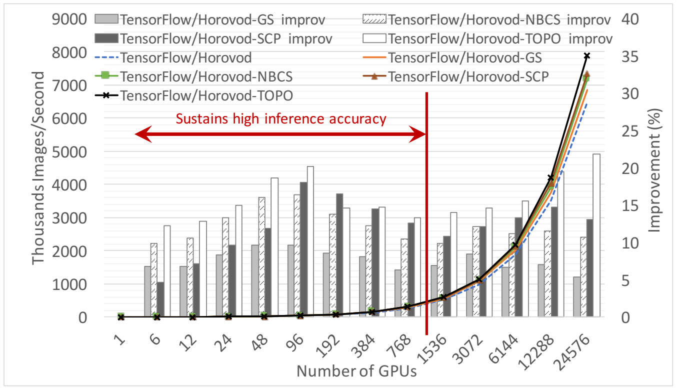


Fig. 10. Weak-scaling results on ResNet50: image-processing rates (images/second) and percentage improvement in performance.

significant performance improvements even in such cases, and (2) as noted in Section I, the general trend in the research community seems to be towards algorithmic improvements that allow for larger batch sizes, thus indicating the increasing importance of studying the scalability of DL frameworks on large supercomputing systems.

Because TensorFlow/Horovod-TOPO delivers the best performance gain, we further analyze its performance in order to understand the improvement. Specifically, we compare the GPU time breakdown of TensorFlow/Horovod-TOPO with that of the original TensorFlow/Horovod in Figure 11. The improvement with TensorFlow/Horovod-TOPO is mainly from the reduction of the Horovod background thread overhead. On 24,576 GPUs, this overhead shrinks from $\sim$46% of the execution time to 3.4%. We note that some load imbalance still remains in the execution, as evidenced by the increase in the time taken by ncclAllReduce. This load imbalance does not appear as part of the Horovod background thread overhead because the background thread's workflow is now completely nonblocking. Instead, it shows up in the ncclAllReduce time. Investigating this load imbalance is outside the scope of this paper but part of our future work.

Figure 12 presents our strong-scaling results. Here we use a fixed global batch size of 24,576, which still sustains state-of-the-art accuracy. The local batch size is scaled proportionally with the number of GPUs. We use at least 96 GPUs in this experiment to ensure sufficient memory. The performance results show similar trends as weak scaling: our optimizations improve performance by up to $\sim$19%. We experience a slight drop in the image processing rate after 1,536 GPUs because the local batch size becomes too small (i.e., $\leq$ 8) and the communication time, which increases with the number of GPUs, dominates the overall time.

B. Horovod-TOPO's Performance on Other Neural Networks

Below we analyze the performance of Horovod-TOPO while training a variety of neural networks.

1) Graph Parsing Overhead: As noted in Section IV-D, Horovod-TOPO traverses the TensorFlow computation subgraph prior to the first execution of the subgraph to obtain tensor dependencies and the tensor fusion model. To understand how much impact the graph parsing has on the overall performance of training various neural networks, we compute the node and edge counts in the subgraph and measure the graph traversal overhead, as shown in Table I.

TABLE I
COMPUTATION GRAPH CHARACTERISTICS AND HOROVOD-TOPO'S GRAPH TRAVERSAL OVERHEAD. (THE OVERHEAD IS NOT INCLUDED IN THE RESULTS IN SECTION V-A.)

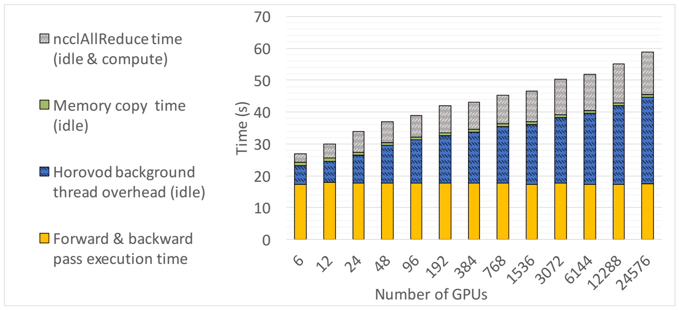
CNN Name	Node Count	Edge Count	Traversal Overhead	
			ms	Percent[a]
ResNet18	1,714	2,565	32	0.18
ResNet34	2,866	4,325	86	0.37
ResNet50	3,988	6,042	159	0.04
ResNet101	7,558	11,499	594	1.49
ResNet152	11,128	16,956	1,274	2.33
AlexNet	655	893	6	0.02
GoogLeNet	3,499	4,970	182	0.89
Inception-v3	6,146	9,211	382	1.12
VGG16	1,091	1,479	17	0.03

[a]Percentage in the execution time of the complete ImageNet training on 12,288 GPUs.

As expected, the graph traversal time increases proportionally with the node and edge counts. In most cases, the total traversal overhead is a few tens or hundreds of milliseconds. The overhead is the highest for ResNet152, which takes around 1.2 seconds for the graph traversal. To put this in perspective, the overall execution time of the training runs is typically on the order of tens of minutes to even hours. For ResNet152, for example, our graph traversal overhead accounts for a mere 2.33% of the total execution time of a complete ImageNet training. Thus, we conclude that Horovod-TOPO incurs an insignificant amount of overhead for the graph parsing.

2) Weak-scaling Evaluation: Figure 13 illustrates the image-processing rates and improvement percentage while using TensorFlow/Horovod-TOPO compared with TensorFlow/Horovod for various neural networks. For all the experiments, the input pipeline is identical. For simplicity in discussion, we define the term "**load-imbalance susceptible window**" or "**LIS window**," which refers to the period where the graph scheduling component and the input pipeline are both active but tensor transfers have not yet occurred. During the LIS window, load imbalance has the highest impact on the graph scheduling, which in turn has a direct impact on the core neural network training on the GPUs. When the input pipeline is not active, the load imbalance that is caused by the Horovod background thread does not have a significant impact on the graph scheduling because the cores are mostly free. In our evaluation, for the ResNet networks, we observe 5–21% performance gain. The smaller ResNets tend to achieve larger improvement because the input pipeline computation amount is fixed and thus smaller ResNet networks have a larger LIS window (as a fraction of the total execution time) than the larger networks. The LIS windows of ResNets 18, 34, 50, 101, and 152 are approximately 46%, 55%, 48%, 30%, and 23%, respectively.

Similar to ResNets, DNNs that have a large LIS window tend to show better performance improvements. For example, GoogLeNet, which has an LIS window of 56%, achieves the



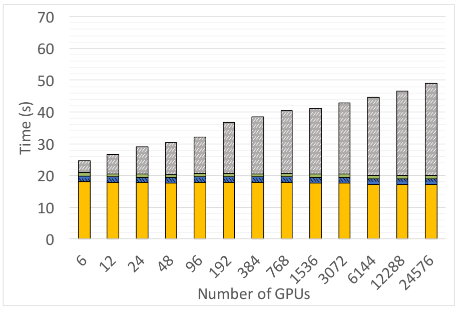


Fig. 11. GPU time breakdown of ResNet50 training: (a) TensorFlow/Horovod; (b) TensorFlow/Horovod-TOPO.

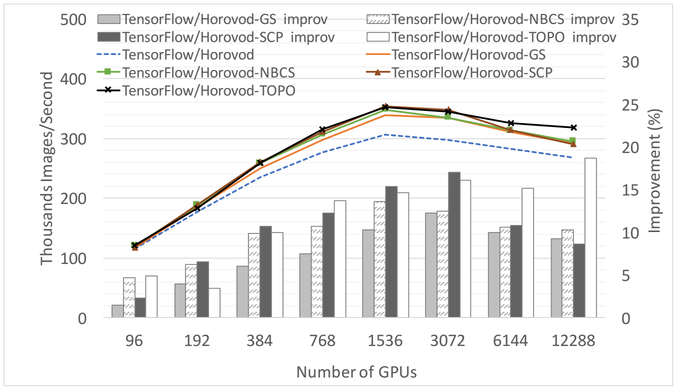


Fig. 12. Strong-scaling results on ResNet50: image-processing rates (images/second) and percentage improvement in performance.

best performance improvement with an average improvement of 25% and a maximum improvement of 35%. Inception-v3 has a slightly smaller LIS window of 42% and correspondingly achieves a smaller performance gain of 10–18%. VGG16 has a relatively small LIS window of 32%, limiting its performance gain to 2–6%.

For AlexNet, Horovod-TOPO achieves slightly worse performance (∼6%) than the original Horovod. This is because the computation graph of AlexNet is *very small* (see Table I) and the overhead associated with managing the tensor ordering and fusion buffers hurts performance more than the benefit of the reduced cache synchronization. While additional (engineering) optimizations to Horovod-TOPO to improve how the tensor ordering and fusion buffers are managed could be made, they would simply bring the performance of Horovod-TOPO in line with that of original Horovod.

VI. Related Work

Allreduce communication has been well studied in the past [3, 11, 14, 34, 35]. Our work, however, is different from this existing literature in that it does not directly target the Allreduce communication itself, but rather the imbalance between the different processes that indirectly affects the Allreduce communication. An approach that is similar to Horovod-TOPO was proposed in [14]. Importantly, though, this existing work does not directly handle tensor ordering—in fact, it ignores tensor dependencies and forces tensor transfers to be in the reverse order of the layers (i.e., assuming that tensor transfers only happen in the backpropagation phase). We believe that this approach is not applicable to many graph structures. For example, such method would not work with the synchronization of batch-normalization statistics in [35] that takes place in both forward and backward passes and the statistics have dependencies between each other. In contrast, our work analyzes the actual dependency structure in the graph and creates a tensor ordering schedule that would work for any static graph.

There exist several communication protocols and libraries that are developed and optimized for distributed DL. TensorFlow uses Google's Remote Procedure Call (gRPC) [21], which operates on top of TCP, for communication. However, gRPC/TCP is not supported or performs poorly on most supercomputers. On the other hand, MPI has been well optimized for a broad range of communication protocols for supercomputing systems. Several DL frameworks [1, 8, 24] adopt MPI as a native communication protocol. Because GPUs are becoming the most popular accelerators for DL, various GPU-aware MPI implementations also exist [5, 19].

Cray Programming Environment Machine Learning Plugin [22] is built on top of MPI. Despite its impressive scaling, however, its operations are blocking synchronous (as opposed to Horovod's nonblocking asynchronous operations), making them susceptible to deadlock, and are invalid in some computation graphs. Aluminum [10], an asynchronous GPU-aware communication library, contains a novel latency-optimized Allreduce algorithm to improve the performance of communication that overlaps with computation.

MPI, NCCL, DDL, MLSL, and Gloo are low-level collective communication libraries for which users have to perform tensor ordering before using them to prevent deadlock. Horovod is a communication plugin that wraps around these high-performance collective communication libraries to provide an efficient tensor ordering approach for DL frameworks. Horovod has shown impressive performance in the literature [17, 25]. However, its loss of scalability on large-scale systems is well documented [33]. Kurth et al. [18] utilized Horovod-MLSL (MLSL backend to Horovod) and reported that it required additional thread binding to avoid the background threads from monopolizing CPU cores.

Overall, to the best of our knowledge, our work is the first to identify the contention between data-processing components as a cause of TensorFlow/Horovod's scalability limitation.

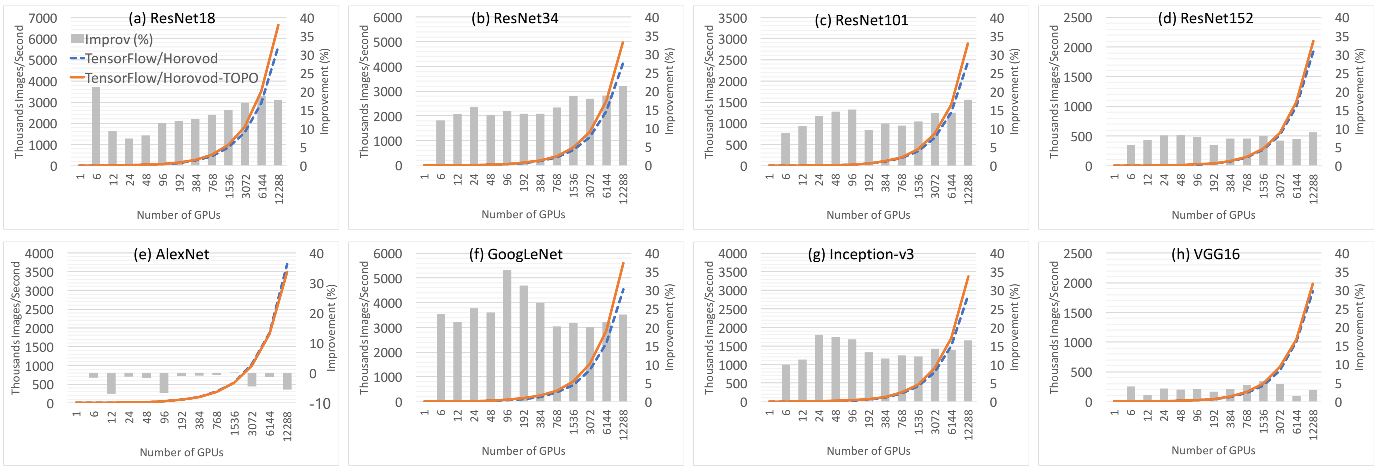


Fig. 13. Weak-scaling results on various DNNs (image-processing rates and improvement percentage): (a) ResNet18; (b) ResNet34; (c) ResNet101; (d) ResNet152; (e) AlexNet; (f) GoogLeNet; (g) Inception-v3; (h) VGG16. Note: the scale of the image-processing rate axis varies among graphs.

VII. Concluding Remarks

In this paper, we investigate the limitations in scaling deep-learning frameworks to large-scale supercomputing systems. Specifically, we analyze TensorFlow and Horovod—state-of-the-art DL software frameworks—and identify resource contention between data-processing components, which causes load imbalance, as the root cause of their scaling limitations. To address this scaling limitation, we propose four solutions that efficiently tackle such load imbalance and demonstrate up to 35% improvement in performance on 24,576 GPUs of the Summit supercomputer at Oak Ridge National Laboratory.

References

[1] M. Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.

[2] M. Abadi et al. A Computational Model for TensorFlow: An Introduction. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 1–7. ACM, 2017.

[3] T. Akiba et al. Extremely Large Minibatch SGD: Training ResNet-50 on Imagenet in 15 Minutes. *arXiv preprint arXiv:1711.04325*, 2017.

[4] D. Alistarh et al. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.

[5] A. A. Awan et al. Scalable Distributed DNN Training using TensorFlow and CUDA-Aware MPI: Characterization, Designs, and Performance Evaluation. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 498–507, May 2019.

[6] E. Azarkhish et al. Neurostream: Scalable and Energy Efficient Deep Learning with Smart Memory Cubes. *IEEE Transactions on Parallel and Distributed Systems*, 29(2):420–434, 2017.

[7] M. S. Birrittella et al. Intel® Omni-path Architecture: Enabling Scalable, High Performance Fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 1–9. IEEE, 2015.

[8] T. Chen et al. MXNET: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv:1512.01274*, 2015.

[9] M. Cho et al. PowerAI DDL. *arXiv preprint arXiv:1708.02188*, 2017.

[10] N. Dryden et al. Aluminum: An Asynchronous, GPU-aware Communication Library Optimized for Large-Scale Training of Deep Neural Networks on HPC systems. 9 2018.

[11] P. Goyal et al. Accurate, Large Minibatch SGD: Training Imagenet In 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.

[12] K. He et al. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[13] J. Hines. Stepping up to Summit. *Computing in Science Engineering*, 20(2):78–82, Mar 2018.

[14] X. Jia et al. Highly Scalable Deep Learning Training System with Mixed-precision: Training ImageNet in Four Minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[15] A. Krizhevsky et al. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira et al, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[16] T. Kurth et al. Deep Learning at 15PF: Supervised and Semi-supervised Classification for Scientific Data. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 7. ACM, 2017.

[17] T. Kurth et al. Exascale Deep Learning for Climate Analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 51:1–51:12, Piscataway, NJ, USA, 2018. IEEE Press.

[18] T. Kurth et al. TensorFlow at Scale: Performance and productivity analysis of distributed training with Horovod, MLSL, and Cray PE ML. *Concurrency and Computation: Practice and Experience*, 31(16):e4989, 2019.

[19] K. Manian et al. Characterizing CUDA Unified Memory (UM)-Aware MPI Designs on Modern GPU Architectures. In *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*, pages 43–52. ACM, 2019.

[20] S. Markidis et al. NVIDIA Tensor Core Programmability, Performance & Precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[21] A. Mathuriya et al. Scaling GRPC TensorFlow on 512 Nodes of Cori Supercomputer. *arXiv preprint arXiv:1712.09388*, 2017.

[22] A. Mathuriya et al. CosmoFlow: Using Deep Learning to Learn the Universe at Scale. In *SC*, 2018.

[23] K. Osawa et al. Large-Scale Distributed Second-Order Optimization Using Kronecker-Factored Approximate Curvature for Deep Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 12359–12367, 2019.

[24] A. Paszke et al. Automatic differentiation in PyTorch. In *NIPS-W*, 2017.

[25] A. Sergeev et al. Horovod: Fast and Easy Distributed Deep Learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[26] G. Shainer et al. The Development of Mellanox/NVIDIA GPUDirect over InfiniBand a new model for GPU to GPU Communications. *Computer Science-Research and Development*, 26(3-4):267–273, 2011.

[27] K. Simonyan et al. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[28] S. L. Smith et al. Don't Decay the Learning Rate, Increase the Batch Size. *arXiv preprint arXiv:1711.00489*, 2017.

[29] S. Sridharan et al. On Scale-out Deep Learning Training for Cloud and HPC. *arXiv preprint arXiv:1801.08030*, 2018.

[30] C. Szegedy et al. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[31] C. Szegedy et al. Rethinking the Inception Architecture for Computer Vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[32] R. Thakur et al. Optimization of collective communication operations in MPICH. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, February 2005.

[33] X. Wu et al. Performance, Energy, and Scalability Analysis and Improvement of Parallel Cancer Deep Learning CANDLE Benchmarks. In *Proceedings of the 48th International Conference on Parallel Processing*, page 78. ACM, 2019.

[34] M. Yamazaki et al. Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds. *arXiv preprint arXiv:1903.12650*, 2019.

[35] C. Ying et al. Image Classification at Supercomputer Scale. *arXiv preprint arXiv:1811.06992*, 2018.

[36] Y. You et al. Scaling SGD Batch Size to 32k For Imagenet Training. *arXiv preprint arXiv:1708.03888*, 6, 2017.

[37] Y. You et al. Imagenet Training in Minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, page 1. ACM, 2018.

[38] Y. You et al. Reducing BERT Pre-Training Time from 3 Days to 76 Minutes. *arXiv preprint arXiv:1904.00962*, 2019.

[39] W. Zhang et al. Staleness-aware Async-SGD for Distributed Deep Learning. *arXiv preprint arXiv:1511.05950*, 2015.