Understanding Scalability and Fine-Grain Parallelism of Synchronous Data Parallel Training

Jiali Li*, Bogdan Nicolae†, Justin Wozniak†, George Bosilca*

*The University of Tennessee, Knoxville, USA
Email: jli111@vols.utk.edu,bosilca@icl.utk.edu
†Argonne National Laboratory, USA
Email: {bnicolae,woz}@anl.gov

***Abstract*—In the age of big data, deep learning has emerged as a powerful tool to extract insight and exploit its value, both in industry and scientific applications. With increasing complexity of learning models and amounts of training data, data-parallel approaches based on frequent all-reduce synchronization steps are increasingly popular. Despite the fact that high performance computing (HPC) technologies have been designed to address such patterns efficiently, the behavior of data-parallel approaches on HPC platforms is not well understood. To address this issue, in this paper we study the behavior of Horovod, a popular data parallel approach that relies on MPI, on Theta, a pre-Exascale machine at Argonne National Laboratory. Using two representative applications, we explore two aspects: (1) how performance and scalability is affected by important parameters such as number of nodes, number of workers, threads per node, batch size; (2) how computational phases are interleaved with all-reduce communication phases at fine granularity and what consequences this interleaving has in terms of potential bottlenecks. Our findings show that pipelining of back-propagation, gradient reduction and weight updates mitigate the effects of stragglers during all-reduce only partially. Furthermore, there can be significant delays between weight updates, which can be leveraged to mask the overhead of additional background operations that are coupled with the training.**

***Index Terms*—deep learning, data-parallel training, behavior analysis**

I. Introduction

Deep learning applications are rapidly gaining traction both in industry and for scientific computing. A key driver for this trend has been the unprecedented accumulation of big data, which exposes plentiful learning opportunities thanks to its massive size and variety. Unsurprisingly, there has been significant interest to adopt deep learning at very large scale on supercomputing infrastructures in a wide range of scientific areas: fusion energy science, computational fluid dynamics, lattice quantum chromodynamics, virtual drug response prediction, etc.

Under these circumstances, learning models are increasingly becoming more complex and exhibit deeper structures, up to the point where serial training becomes unfeasible. One solution to this challenge is synchronous data-parallel training [1], which relies on a tightly coupled parallelization strategy that involves frequent all-reduce synchronization steps. Specifically, this is an iterative process: replicas of the model are trained in parallel with different batches, which produces different gradients that are averaged using all-reduce and then used to independently update the weights of each replica. Given the same initial weights, the averaging of the gradients produces the same weight updates on all replicas, therefore keeping all weights synchronized.

However, such an approach is not without challenges, as frequent all-reduce operations can become a bottleneck that hinders scalability. In a quest to scale beyond a few nodes, recent efforts such as Horovod have stated considering MPI as the underlying communication framework that provides efficient all-reduce implementations. Although promising, this does not magically solve the problem of adopting data parallel training on supercomputing infrastructures.

One challenge in this context is the inherent complexity of modern deep learning approaches: even without data parallelism, they are capable of taking advantage of multi-core and hybrid architectures by parallelizing the processing of batches at fine granularity. One common strategy is layer-wise pipelining, which can be done both during feed-forward of samples and back-propagation of gradients. Specifically, during back-propagation, once the gradients for a layer have been computed, the weight updates for all successive layers can be performed in parallel with the computation of the gradients for the previous layer. However, if gradients are averaged using all-reduce operations, stragglers will negatively impact the potential to leverage pipelining, even if the actual all-reduce operations are fast by themselves.

Another important challenge is the need to couple training with additional pre-processing [2] (e.g., a transformation of the samples such as rescaling of images) or post-processing [3]. For example, one important post-processing pattern is check-pointing, which can be used for a variety of use cases: tracking the evolution of a model to understand the influence of samples on its weights; transfer of learning where a model partially or fully trained for one problem is partially trained and used for inference in a another problem; fault tolerance based on restart. Ideally, such operations need to have a minimal impact on the learning process by leveraging idle shared resources.

Therefore, it is important to understand the behavior of deep learning frameworks at fine granularity, such as to expose potential bottlenecks and opportunities to leverage idle resources. To this end, our paper contributes with a study that aims to quantify and explain important aspects that impact the behavior. We summarize our contributions as follows:

- We discuss and analyze important aspects that impact the performance and scalability of synchronous data parallel training. In particular, we emphasize the interplay between pipelining and all-reduce (Section II).
- We introduce a series of key metrics and illustrate how to extract them through an analytics framework, which we apply to fine-grain logs of events produced by Horovod [4], a popular synchronous data parallel training framework (Section III).
- We study CANDLE NT3 and ResNet-50 on the Theta supercomputer, exposing their sensitivity to various parameters with respect to performance and scalability, while zooming on the most efficient configurations to explain the interplay of pipelining and all-reduce through the key metrics. Based on these key metrics, we discuss opportunities for improvement and efficient coupling with pre/post processing (Section IV).

II. Background and Related Work

Deep Learning (DL) algorithms are a class of machine learning algorithms that are based on complex neural neural networks with a large number of layers (hence called deep). They have have been successfully applied in a wide range of tasks: image recognition, machine translation, forecasting [5]. Such algorithms have increasingly gained attention in high performance computing as a complement to simulations (e.g., identify regions of interest, select promising initial conditions, etc.).

DL algorithms primarily use gradient descent to update the weights. This is an iterative technique that works as follows. First, a training sample is used as the input of the neural network (first layer) to compute the output layer by layer until a prediction of the result is obtained (last layer). This step is called feed-forward. Then, the difference (gradients) between the predicted and actual result ("ground truth") is used to update the weights layer by layer up to the first layer. This step is called back-propagation. The goal is to converge to a minimum that is representative of all training samples and acts as an interpolation function for the whole problem. An important type of gradient descent is mini-batch gradient descent, where multiple training samples are used for feed-forward and the resulting average gradients for back-propagation. This speeds up the training process, both because fewer iterations are needed, and because there are fewer abrupt changes to the descent due to biased samples, which reduces the noise of finding the best direction to take. However, a batch size that is too large may get stuck in a local optimum, which leads to poorer generalization. Therefore, it is important to choose a number of training samples (batch size) that optimizes this trade-off.

Gradient descent is a computationally expensive technique. The explosion of data sizes and need to solve more complex problems have led to the introduction of deeper structures with more layers (e.g., complex residual networks with 1000+ layer, such as ResNet [6]). Therefore, gradient descent is not only more expensive to run simply because it needs to process more batches, but also because each batch itself is more expensive to process. To solve this problem, distributed DL algorithms have been developed that are capable of scaling horizontally on multiple compute nodes. They broadly fall into two classes.

Synchronous data-parallel DL algorithms leverage the idea of creating replicas of the learning model on each node and training each replica in parallel with a different batch. During feed-forward, this can be done in an embarrassingly parallel fashion. However, during back-propagation, the weights are not updated with the local gradients, but with global average gradients computed across all nodes using blocking all-reduce operations. This process is illustrated in Figure 1.

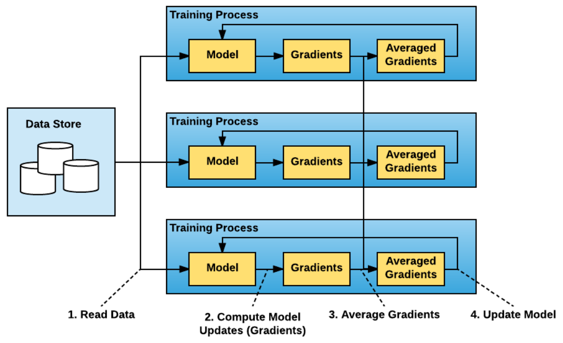


Fig. 1. Synchronous data-parallel training.

Asynchronous data-parallel DL algorithms aim to alleviate the overhead of synchronization due to blocking all-reduce in order to mitigate the problem of stragglers. A popular approach is to use a parameter server [7], which holds the most up-to-date version of the learning model. Then, same as for the synchronous case, each model replica is trained independently with different batches. However, instead of averaging the gradients globally, each node sends the local gradients to the parameter server, where they are used to update the weights. Since the replicas quickly go out of sync with the parameter server, they have to be refreshed periodically. However, this approach has two issues: (1) the parameter server can quickly become overwhelmed with gradient updates and request to refresh the learning model, which limits scalability; (2) the gradients are not computed based on the most up-to-date weights, which makes the convergence slower. Although there are recent promising efforts [8], the inherent complexity is still a barrier for wide adoption in practice. For this reason, we focus our study on synchronous data-parallel approaches.

A. *Synchronous data-parallel optimizations*

DL algorithms take advantage of multi-core and hybrid architectures to pipeline the gradient computation with the weight updates. Specifically, once the gradients up to a given layer have been computed, they can be applied to update the weights of all other upper layers, while concurrently computing the gradients for the next lower layer. This is called layer-wise pipelining. Over time, several runtimes that implement such ideas have become popular: Tensorflow [9], Caffe [10], Torch [11]. They are optimized for compute nodes that feature multi-core (e.g., Intel Xeon/KNL) or hybrid (e.g., CPUs + GPUs) architectures.

Therefore, implementing a synchronous data-parallel approach exactly as illustrated in Figure 1 using a single large all-reduce operation for all gradients of all layers would be suboptimal, because it would cancel the benefits of pipelining. To address this issue, all-reduce operation can be performed at fine granularity in parallel with the gradient computation and back-propagation [12]. This is illustrated in Figure 2.

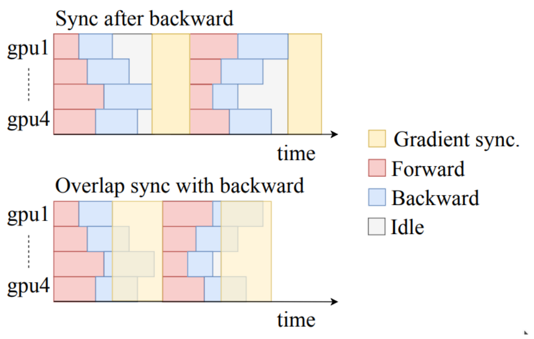


Fig. 2. Synchronous data parallel optimizations: Serial vs. pipelined gradient computation (backward) and all-reduce + averaging (gradient sync)

However, unlike the sequential base case, the weight updates to the upper layers need to be performed using the averaged gradients, while the computation of the gradients of the next lower layer needs to be performed with the local gradients. This implies the all-reduce and averaging of the gradients must be performed on a copy rather than in-place. Furthermore, the averaging of the copied gradients must finish before they can be applied to update the weights.

B. Horovod

Many popular implementations of synchronous data-parallel training emerged: Distributed Tensorflow, Distributed Torch, CNTK [13], MaTEx (Machine Learning Toolkit at Extreme Scale) [14], CaffeonSpark [15], FireCaffe [16]. Some of these runtimes can use MPI as the underlying communication layer that provides an optimized all-reduce implementation, which is a natural fit for supercomputing architectures.

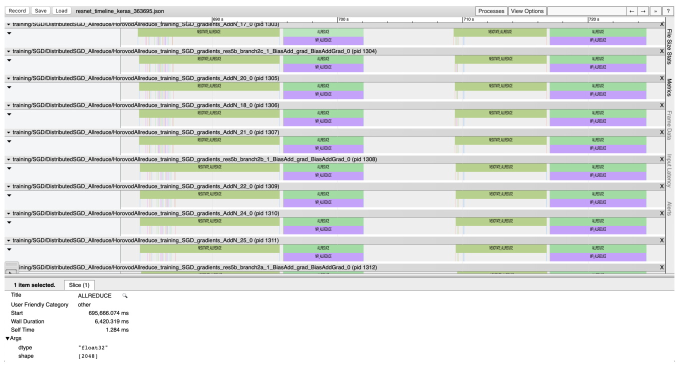
However, much of the DL ecosystem has evolved around Python. High-level libraries such as Keras [17] were specifically designed to hide the complexity of interacting with DL runtimes directly. In this context, Horovod, a relatively recent implementation that aims to continue this tradition emerged. Specifically, Horovod hides the details of parallelization by wrapping around high-level classes typically instantiated by DL users. Using this approach, there are only minimal changes necessary to make convert a serial base code into a synchronized data-parallel code. Most of these changes are related to the fact that each worker in charge of a model replica needs to access a different batch, which implies a need to either to shuffle, split and distribute or to randomly sample the training data on each worker. This is a delicate choice well studied in the literature [18] and there are reference implementations available in Keras that can be extended and customized as needed. Thanks to this ease-of-use, Horovod is one of the most widely used synchronous deep learning frameworks.

Internally, Horovod can use multiple communication libraries to perform all-reduce, depending on where the workers are located (same node but different GPUs, different nodes, etc.). We focus our study the case when MPI is used. In this case, each worker is assigned to a MPI rank and holds a model replica. Furthermore, the layers of the model and the corresponding gradients are represented as tensors. To maintain consistency with the terminology used by Horovod, we will refer to them as tensors for the rest of this paper.

Horovod implements an optimized version of synchronized data parallel training (detailed in Section II) as follows: rank 0 is designated as a global coordinator. As soon as all ranks have finished feed-forward, they employ a three-phase protocol for each tensor. In the first phase (called "negotiation"), all ranks compute the local gradients, create a local copy and report completion to the coordinator. This is followed by a "reduction" phase that performs a blocking all-reduce on the local copies using the *sum* operator, which has an optimal MPI implementation. Finally, in the third phase the reduced gradients are averaged (sum divided by number of ranks, which maintains the same tensor size). The result is passed to the model replica to update the weights of the layer corresponding to the tensor (each weight has a corresponding averaged gradient). Since the gradient computation is completely localized, once the negotiation for a given tensor has finished, the next negotiation for the adjacent lower tensor can be immediately started without waiting for the reduction or average phase. MPI does not allow multiple concurrent all-reduce operations, therefore the second phase is serialized: a new reduction is started only when both the negotiation and the reduction of the next adjacent upper tensor has finished. This is also the reason why a coordinator is needed. After the reduction, the third phase is again localized, therefore it can be performed in parallel with any other phases of the same or different tensor.

A key feature that makes Horovod easy to study at fine granularity is its logging capability called *the timeline*. Since rank 0 acts as a global coordinator, it can collect information about when important events happen and log them based on its local timestamp, which provides a coherent ordering. Specifically, information about the following events is collected for each tensor: when the negotiation phase has started, when each rank reported completion of the negotiation, when the reduction phase has started, when the reduction phase has ended. Horovod organizes this information in a specific JSON file format that can be loaded and visualized externally (e.g., using Google Chrome's built-in tools). An example of this is illustrated in Figure 3. Unfortunately the timeline quickly explodes at scale and becomes a complex interleaving difficult to understand visually, negating its ability to explain fine-grain details.

Several efforts have focused on the investigation of the behavior of deep learning applications from different perspectives: performance and power characteristics [19], scalability and fine-tuning [20], GPU optimizations [21], I/O workloads [22], [23]. However, a systematic understanding of fine-grain behavior at tensor level that explains the interplay of layer-wise pipelining and all-reduce in synchronous data parallel training is missing. To the best of our knowledge, we are the first to study this aspect and to discuss the opportunities it opens for further optimizations and/or coupling with pre/post processing.


Record
Save
Load
resnet_timeline_keras_363695.json
Processes
View Options
690 s
700 s
710 s
720 s
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_AddN_17_0 (pid 1303)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_res5b_branch2c_1_BiasAdd_grad_BiasAddGrad_0 (pid 1304)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_AddN_20_0 (pid 1305)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_AddN_18_0 (pid 1306)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_AddN_21_0 (pid 1307)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_res5b_branch2b_1_BiasAdd_grad_BiasAddGrad_0 (pid 1308)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_AddN_22_0 (pid 1309)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_AddN_24_0 (pid 1310)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_AddN_25_0 (pid 1311)
ining/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_res5b_branch2a_1_BiasAdd_grad_BiasAddGrad_0 (pid 1312)
NEGOTIATE_ALLREDUCE
ALLREDUCE
MPI_ALLREDUCE
File Size Stats
Metrics
Frame Data
Input Latency
Alerts
1 item selected.
Slice (1)

```
Title                   ALLREDUCE
User Friendly Category  other
Start                   695,666.074 ms
Wall Duration             6,420.319 ms
Self Time                     1.284 ms
▾Args
   dtype                "float32"
   shape                [2048]
```

Fig. 3. Horovod timeline: An example of the visual representation of the events generated by the ResNet-50 benchmark

III. Timeline Analytics

We have contributed with an analytics framework that parses the Horovod timeline to produce aggregated statistics that are specifically targeted at explaining how well the parallelization of the various phases works, where potential bottlenecks are, and what opportunities to leverage idle resources exist. Note that our approach is by no means tied to Horovod: given appropriate instrumentation, the key metrics and principles discussed in this section are general enough to be applicable for other frameworks as well.

To this end, we introduce a series of key metrics, which are detailed below. Note that the training process involves multiple epochs, with each epoch consisting of multiple rounds during which each worker processes a batch. To avoid bias due to warm-up, the first round is excluded from the analysis.

Feed-forward: measures the average elapsed time from the start of a new round until the start of the first negotiation. This is averaged for all rounds. This metric is important because it indicates for how long the workers are performing embarrassingly parallel computations and do not need to synchronize with each other, which can be exploited to leverage the idle I/O and network bandwidth for other purposes.

Negotiation delays: measures the sum of all idle time intervals for each rank between the moment when the first negotiation has started until the beginning of the next round. This is averaged for all workers and rounds. An idle time interval happens when a worker is not involved in any negotiation (i.e., has finished reporting to the coordinator for all negotiation phases that have started) or in any reduce phase for any of the tensors. A common cause for this are stragglers: if a rank is slow in finishing one of the negotiations, it blocks not only the corresponding reduce phase, but all reduce phases that follow (because all-reduce operations are serialized), ultimately leading to the situation where the other ranks finish the third phase and sit idle waiting for it. This metric is important because it exposes the degree of imbalance experienced by the workers, which causes inefficiencies in the pipeline. This can be used either to better tune the pipeline or scavenge idle resources for other types of work.

Back-propagation window per tensor: measures for each tensor the interval from the moment when the all-reduce operation has finished until the beginning of the next round. This is averaged for all rounds. In other words, it indicates how much time is available after the end of the all-reduce to average the gradients and apply them to update the weights of the layer corresponding to the tensor. This metric is important because it exposes potential opportunities to delay the third phase and/or schedule additional work that modifies the weights (e.g. pruning of weights close to zero) without delaying the next round (once the weights have been updated, they will not be read again before the beginning of the next round).

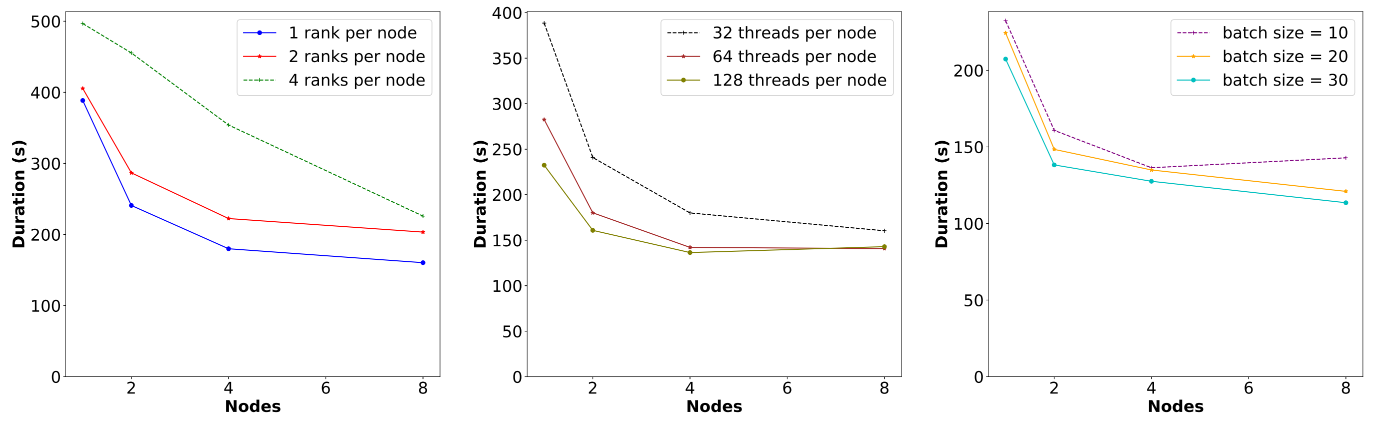
Reaction bandwidth per tensor: the amount of work needed for the third phase and/or to schedule additional work during the back-propagation window depends on the size of the tensor (the number of averaged gradients equals number of weights in the layer). Therefore, it is important to measure the minimum rate at which the tensor needs to be processed (i.e., react on it after all-reduce) such as to avoid delaying the next round. We call this metric reaction bandwidth and obtain it by dividing the size of the tensor by its back-propagation window.

Immutable bandwidth per tensor: after the start of a new round, a layer will stay immutable and can be safely read until the end of its corresponding reduce phase. This is a worst case scenario, because the layer may also stay immutable during the previous back-propagation window after the gradients have been averaged and the weights have been updated. Therefore, in the ideal case, when considering these operations to be negligible, a layer will stay immutable between two reduce phases. Therefore, we propose to divide the size of the corresponding tensor by the average time interval for the best and worst case. This gives the minimum bandwidth necessary to safely read a layer and use it as an immutable object (unlike reaction bandwidth, which allows changes) without causing delays. Such a metric is important to understand the opportunity to perform asynchronous immutable operations (e.g., checkpointing, compression, etc.).

IV. Evaluation

A. Experimental Setup

Our experiments were performed on ANL *Theta*, a 11.69 petaflops pre-Exascale Cray XC40 system based on the



(a) Variable number of ranks per node. Batch size is 10. Threads per node is 32.

(b) Variable number of threads per node. Batch size is 10. One rank per node (optimal setting).

(c) Variable batch size. One rank per node (optimal setting). Threads per node is 128 (optimal setting).

Fig. 4. CANDLE NT3: Performance and strong scalability for an increasing number of nodes under variable batch size, ranks per node, threads per node. Threads per rank equals threads per node divided by number of ranks.

second-generation KNL Intel Xeon Phi 7230 SKU. The system is equipped with 4392 nodes, each containing a 64 core processor (256 hardware threads) with 16 gigabytes (GB) of high-bandwidth in-package memory (MCDRAM), 192 GB of DDR4 RAM (20 GB/s), and a 128 GB SSD (700 MB/s). The interconnect topology is based on Dragonfly with a total bisection bandwidth of 7.2 TB/sec. In terms of software, we use *Horovod* v.0.16.1, *Keras* v.1.10 and *Tensorflow* v.2.2.2. Note that all these libraries are compiled with optimized support for the KNL architecture by taking advantage of Intel's Math Kernel Library (MKL).

B. Applications

We study two representative deep learning applications.

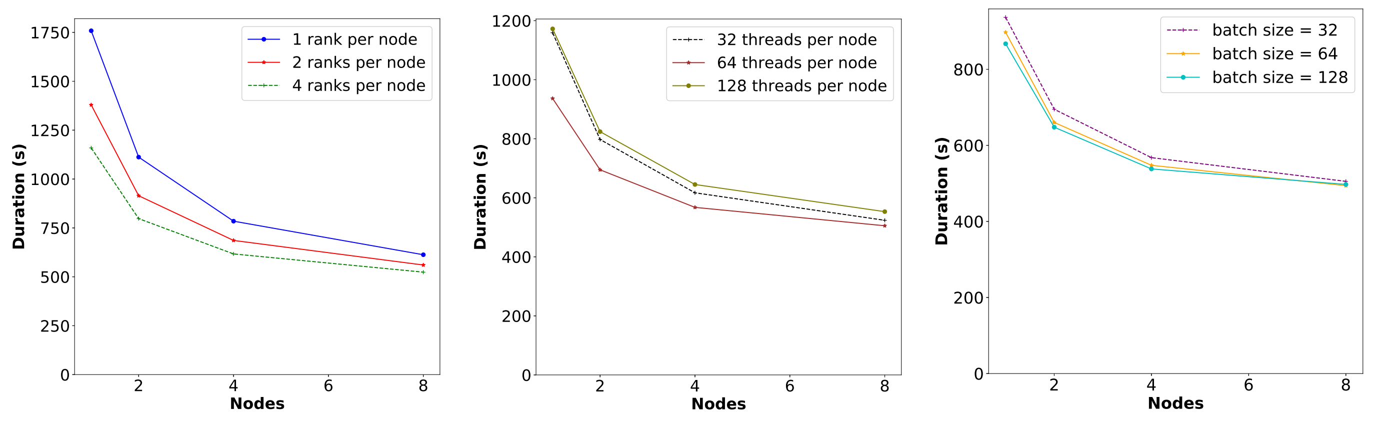
1) CANDLE NT3: CANDLE [24] (Cancer Distributed Learning Environment) is a project that aims to combine the power exascale computing with deep learning to address a series of loosely connected problems in cancer research. Each such problem is driven by a series of benchmarks. One such direction (Pilot 1) aims to predict drug response based on molecular features of tumor cells and drug descriptors. In this context, we study on NT3 [25], which consists of a 1D convolutional networks for classifying tissue expressed as gene sequences as normal or tumorous. This type of network follows the classic architecture of convolutional models with multiple 1D convolutional layers interleaved with pooling layers followed by final dense layers. The optimizer used by NT3 is SGD (stochastic gradient descent). The training data size for this benchmark is $\approx$ 600 MB, which includes 1120 training samples. The default batch size is 20. For the purpose of this work, we modified NT3 to enable data parallel training. Specifically, we introduced a partitioning scheme that evenly distributes the training data to the workers and added a new distributed optimizer based on Horovod.

2) ResNet-50: is a deep neural network where the layers learn residual functions with reference to the input layers, instead of learning unreferenced functions. This allows ResNet to train extremely deep neural networks with 150+ layers, which was difficult prior to its introduction due the problem of vanishing gradients [26]. Due to this breakthrough, ResNet became a highly popular image classification benchmark especially in a simpler form that uses 50 layers. We study this form, called ResNet-50. A data parallel implementation is shipped together with Horovod as an example [4]. The optimizer used by this implementation is also SGD. As training data, we use the ImageNet dataset [27], which is $\approx$ 200 MB large and includes 100,000 samples. The default batch size is 128. The training set of each worker is randomly sampled from the training data.

C. Scalability

First, we evaluate the performance and scalability under variable number of nodes, processes per node, hardware threads per process, and batch size. The goal of this step is to understand how well data parallel training can scale using Horovod, as well as to identify the optimal granularity of parallelism at node level and the impact of the batch size.

To this end, we vary the number of nodes in range: 1, 2, 4, 8. Since Tensorflow can take advantage of multi-core architectures, we vary the number of workers per node in a low spectrum: 1, 2, 4. There are 64 cores per node, therefore we explore three scenarios: under-utilization (half of the cores run 32 hardware threads), full utilization (all cores run a hardware thread), hyper-threading (2 hardware threads per core). For each scenario, we allocate the hardware threads evenly among all workers co-located on the same node. The batch size is a variation around the default setting. We run all experiments for all possible combinations of settings and record the completion time. Since exploring all combinations is a time-consuming process, each run is limited to one epoch, during which each worker will process several batches. This is enough to reveal a repetitive behavior.



(a) Variable number of ranks per node. Batch size is 32. Threads per node is 32.

(b) Variable number of threads per node. Batch size is 10. Four ranks per node (optimal setting).

(c) Variable batch size. Four ranks per node (optimal setting). Threads per node is 64 (optimal setting).

Fig. 5. ResNet-50: Performance and strong scalability for an increasing number of nodes under variable batch size, ranks per node, threads per node. Threads per rank equals threads per node divided by number of ranks.

The results for CANDLE NT3 are highlighted in Figure 4. In all cases, there is a clear trend visible: an increasing number of nodes decreases the training duration for a fixed number of samples, which means strong scalability is possible. First, we focus on the impact of the number of ranks per node. We fix the batch size to 10, which produces many small all-reduce operations and thus finer-grain pipelining. Also, we fix the number of threads per node to 32, which emphasizes an under-subscribed scenario that favors multiple ranks per node. As can be observed in Figure 4(a), the best configuration is one rank per node, which means increasing the degree of fine-grain pipelining leads to better performance than creating more model replicas for the same amount of nodes. Next, we focus on increasing the number of threads to further study the scalability of pipelining. As can be observed in Figure 4(b), when we fix one rank per node, over-subscribing the number of threads per node using hyper-threading (2 per core) produces better results (compared with one thread per core). Thus, we conclude the NT3 model pipeline is highly parallelizable. Next, we study the impact of batch size, using one rank per node and 128 threads per node (Figure 4(c). As expected, an increasing batch size leads to shorter completion time. However, it is interesting to observe that the difference is small, which means the number of steps in an epoch has little impact as long as the total size of the training data per worker remains the same.

We perform a similar study for ResNet-50 (Figure 5). Again, we observe strong scalability in general. However, unlike the case of CANDLE NT3, pipelining does not leverage the full compute capability of a node efficiently, as the optimal number of ranks per node is four (as per Figure 5(a)). Furthermore, the optimal number of threads per node is 64 (one hardware thread per core), which amounts of 16 hardware threads per rank, as shown in Figure 5(b). Interestingly, 128 threads per node (two hardware threads per core) delivers the worst performance, which is the opposite of the NT3 case. Furthermore, under-subscription (32 threads per node) does not seem to degrade performance significantly, which confirms why more ranks per node increase performance. Finally, as depicted in Figure 5(c), the impact of the batch size is negligible in the optimal configuration. This finding is consistent with the behavior of NT3 as well.

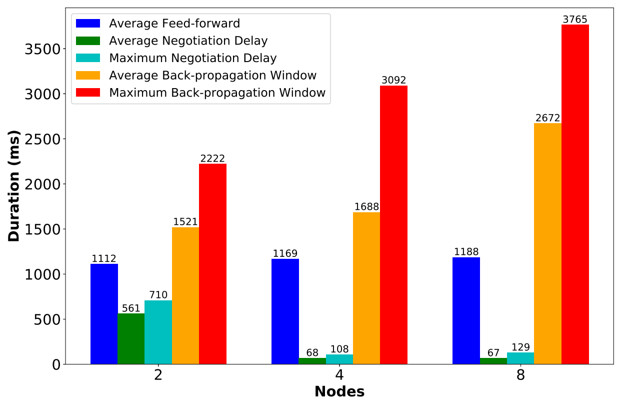
Note that we have experimental results for all possible combinations of settings for both applications. We verified that the optimal number of ranks per node and threads per node remains constant for all combinations, not just the ones highlighted above.

D. Fine-grain study of key metrics

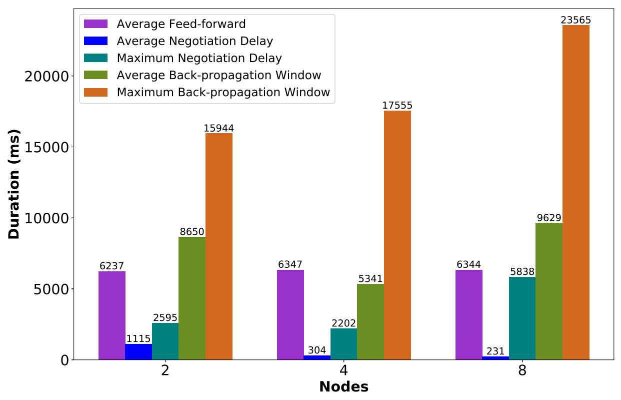
Next, we zoom on the configurations for which we obtained the best performance for a given number of nodes. For these configurations, we study the timeline produced by Horovod using the key metrics produced by the analytics framework introduced in Section III.

Specifically, we fix the number of ranks per node, threads per node and batch size and study the case of 2, 4, 8 nodes (which feature non-trivial communication over the network during all-reduce). Since it is difficult to plot the metrics obtained for each tensor individually, we aggregate them as follows: (1) for the back-propagation window we calculate the average and maximum among all tensors; (2) for the reaction and immutable bandwidth, we calculate the maximum across all tensors. Note that the lower the sustained bandwidth for a tensor needs to be, the more flexibility there is to perform complex post-processing. Therefore, the maximum among all tensors indicates a conservative bandwidth that provides a good overview of the minimum sustained processing rate that avoids any delay regardless of the size of the tensors and their back-propagation window.

The obtained timing metrics are depicted in Figure 6. Overall, both CANDLE NT3 and ResNet-50 behave similarly. The average feed-forward remains relatively constant with

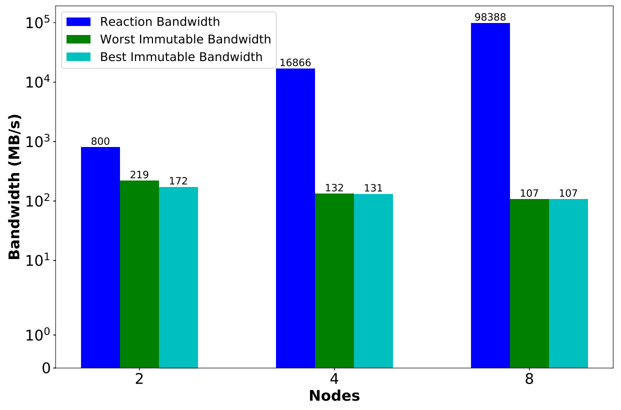


(a) CANDLE NT3: One rank per node, 128 threads per node, batch size 20

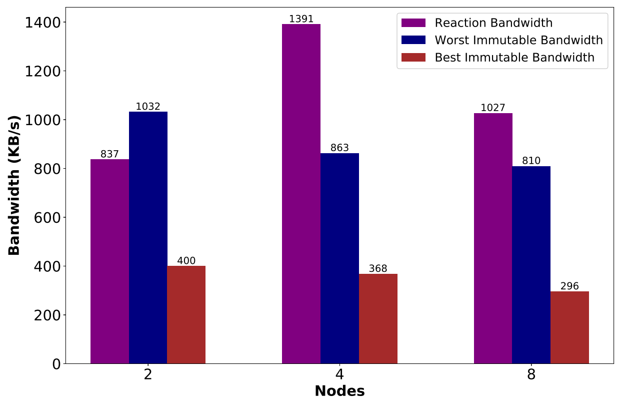


(b) ResNet-50: Four ranks per node, 64 threads per node, batch size 128

Fig. 6. Key metrics related to timing: feed-forward duration, avg. and max. negotiation delay, avg. and max. back-propagation window (higher means more opportunity to leverage idle resources)



(a) CANDLE NT3: One rank per node, 128 threads per node, batch size 20



(b) ResNet-50: Four ranks per node, 64 threads per node, batch size 128

Fig. 7. Key metrics related to bandwidth: reaction bandwidth, worst-case immutable bandwidth, best-case immutable bandwidth (lower means more scheduling flexibility and/or opportunity for additional operations)

increasing number of nodes, with a slight increase observable for more than two nodes (larger number of workers creates stragglers). This hints at consistent opportunities to leverage idle I/O and network resources in the background. The average negotiation delay is significant for two nodes but sharply decreases for more than two nodes due to workers spending more time in all-reduce operations. This means there is little opportunity to consistently leverage idle resources due to imbalance at scale. However, the maximum negotiation delay behaves differently for CANDLE NT3 and ResNet-50: for the former it is close to the average, for the latter it is up to orders of magnitude larger, which implies occasional opportunities to leverage idle resources. Both the average and maximum back-propagation window is larger than the feed-forward duration and increases with scale (which is expected, due to increasing all-reduce overhead).

The bandwidth-related metrics are depicted in Figure 7. As can be observed, for CANDLE NT3 the reaction bandwidth grows fast with increasing scale (note the log scale on the Y axis). This is due to the fact that it features a large lower tensor that makes up the bulk of the model, which is preceded by a few trivial tensors that incur negligible overhead. Therefore, its back-propagation window is tiny and leads to a huge reaction bandwidth, leaving little room to do other operations that modify the tensor without causing delays. However, the situation is different for the immutable bandwidth: the large tensor incurs a large all-reduce overhead, therefore it takes a long time until it is updated again in the next round and can be safely read at a relatively slow rate (less than 200 MB/s, which is enough to copy it to local storage). Also note the small difference between the best and worst immutable bandwidth, which is caused by the tiny back-propagation window of the large tensor. ResNet-50 is in a much different situation. Featuring a large number of tensors that are of comparable size, its pipelining is much deeper and more evenly balanced, which results in very small reaction and immutable bandwidth (note the unit on the Y axis: KB/s). This leaves plenty of room to leverage idle I/O resources and couple the training with additional pre/post processing, regardless whether this implies modifications to the tensor or not. Furthermore, thanks to the small reaction bandwidth, the difference between the worst and best case immutable bandwidth is more pronounced.

V. Conclusions

This paper focuses on the problem of understanding the performance and scalability of synchronous data parallel training at fine granularity. To this end, it introduces series of key metrics and an approach to extract them from low-level operations and events performed by data-parallel frameworks. We illustrate this approach on Horovod, which has a powerful logging mechanism for such low-level operations and events. Using this approach, we study the performance, scalability and behavior of two application benchmarks (CANDLE NT3 and ResNet-50) on a supercomputing infrastructure (ANL Theta).

Our findings show: (1) significant opportunity to leverage idle I/O and network resources during feed-forward; (2) significant flexibility to schedule the back-propagation and/or couple it with additional operations that modify tensors when for deep models with balanced tensor sizes; (3) significant opportunity to perform additional immutable asynchronous operations in the background on the tensors during the training with minimal interference.

Encouraged by these results, we plan to explore in future work how to design advanced asynchronous checkpointing techniques to preserve the state of models at high frequency by taking advantage of the observation that checkpointing is an immutable operation. To this end, we plan to leverage VeloC [28], a large-scale checkpointing system that features asynchronous management of deep storage hierarchies.

Acknowledgments

This research was funded by Argonne National Laboratory, under Contract LDRD-1007397. It used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

[1] E. Yang, S. Kim, T. Kim *et al.*, "An adaptive batch-orchestration algorithm for the heterogeneous gpu cluster environment in distributed deep learning system," in *BigComp'18: 2018 IEEE International Conference on Big Data and Smart Computing*, Shanghai, China, 2018, pp. 725–728.

[2] K. K. Pal and K. S. Sudeep, "Preprocessing for image classification by convolutional neural networks," in *RTEICT'16: 2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology*, Bengaluru, India, 2016, pp. 1778–1781.

[3] I. Bruha, *Pre- and Post-processing in Machine Learning and Data Mining*. Springer LNCS, 2001, pp. 258–266.

[4] "Horovod repository," https://github.com/horovod.

[5] S.-M. Tseng, B. Nicolae, G. Bosilca, E. Jeannot, and F. Cappello, "Towards portable online prediction of network utilization using mpi-level monitoring," in *EuroPar'19 : 25th International European Conference on Parallel and Distributed Systems*, Goettingen, Germany, 2019, pp. 1–14.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR'16: 2016 IEEE Conference on Computer Vision and Pattern Recognition*, June 2016, pp. 770–778.

[7] M. Li, D. G. Andersen, A. Smola *et al.*, "Communication efficient distributed machine learning with the parameter server," in *NIPS'14:Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, Cambridge, MA, USA, 2014, pp. 19–27.

[8] X. Lian, W. Zhang, C. Zhang, and J. Liu, "Asynchronous decentralized parallel stochastic gradient descent," in *ICML'18: The 35th International Conference on Machine Learning*, Stockholm, Sweden, 2018, pp. 3049–3058.

[9] M. Abadi, A. Agarwal, P. Barham *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[10] Y. Jia, E. Shelhamer, J. Donahue *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *ICM'14: The 22Nd ACM International Conference on Multimedia*, Orlando, USA, 2014, pp. 675–678.

[11] "Torch: A scientific computing framework for luajit," http://torch.ch/.

[12] M. Ott, S. Edunov, D. Grangier, and M. Auli, "Scaling neural machine translation," in *WMT'18: Proceedings of the Third Conference on Machine Translation: Research Papers*, Belgium, Brussels, 2018, pp. 1–9.

[13] D. Yu, A. Eversole, M. Seltzer *et al.*, "An introduction to computational networks and the computational network toolkit," Microsoft Corporation, Tech. Rep. MSR-TR-2014-112, 2014.

[14] "Machine learning toolkit for extreme scale (matex)," https://github.com/matex-org/matex.

[15] "Distributed deep learning on hadoop and spark clusters." https://github.com/yahoo/CaffeOnSpark.

[16] F. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer, "Firecaffe: Near-linear acceleration of deep neural network training on compute clusters," in *CVPR'16: 2016 Conference on Computer Vision and Pattern Recognition*, Las Vegas, NV, 06 2016, pp. 2592–2600.

[17] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[18] Q. Meng, W. J. Chen, Y. Wang *et al.*, "Convergence analysis of distributed stochastic gradient descent with shuffling," *Neurocomputing*, vol. 337, pp. 46–57, 2017.

[19] X. Wu, V. Taylor, J. M. Wozniak *et al.*, "Performance, energy, and scalability analysis and improvement of parallel cancer deep learning candle benchmarks," in *ICPP'19: Proceedings of the 48th International Conference on Parallel Processing*, Kyoto, Japan, 2019, pp. 78:1–78:11.

[20] Y. You, Z. Zhang, C.-J. Hsieh *et al.*, "Imagenet training in minutes," in *ICPP'18: Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1:1–1:10.

[21] A. A. Awan, J. Bdorf, C. Chu, H. Subramoni, and D. K. Panda, "Scalable distributed dnn training using tensorflow and cuda-aware mpi: Characterization, designs, and performance evaluation," in *CCGRID'19: 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Larnaca, Cyprus, May 2019, pp. 498–507.

[22] S. W. D. Chien, S. Markidis, C. P. Sishtla *et al.*, "Characterizing deep-learning I/O workloads in tensorflow," in *PDSW-DISCS'18: IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, vol. abs/1810.03035, Dallas, TX, 2018.

[23] F. Chowdhury, Y. Zhu, T. Heer *et al.*, "I/o characterization and performance evaluation of beegfs for deep learning," in *ICPP'19: Proceedings of the 48th International Conference on Parallel Processing*, Kyoto, Japan, 2019, pp. 80:1–80:10.

[24] J. Wozniak, R. Jain, P. Balaprakash *et al.*, "Candle/supervisor: A workflow framework for machine learning applied to cancer research," *BMC Bioinformatics*, no. 19, 2018.

[25] "Candle benchmarks," https://github.com/ECP-CANDLE/Benchmarks.

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR'16: 2016 IEEE Conference on Computer Vision and Pattern Recognition*, Las Vegas, NV, 2016, pp. 770–778.

[27] J. Deng, W. Dong, R. Socher *et al.*, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR'09: Conference on Computer Vision and Pattern Recognition*, Miami Beach, FL, 2009, pp. 248–255.

[28] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "Veloc: Towards high performance adaptive asynchronous checkpointing at large scale," in *IPDPS'19: The 2019 IEEE International Parallel and Distributed Processing Symposium*, Rio de Janeiro, Brazil, 2019, pp. 911–920.