Received: 15 June 2018 | Revised: 20 August 2018 | Accepted: 21 August 2018
DOI: 10.1002/cpe.4989

WILEY

SPECIAL ISSUE PAPER

TensorFlow at Scale: Performance and productivity analysis of distributed training with Horovod, MLSL, and Cray PE ML

Thorsten Kurth[1] | Mikhail Smorkalov[2] | Peter Mendygral[3] | Srinivas Sridharan[4] | Amrita Mathuriya[5]

[1]National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, Berkeley, California
[2]Software and Services Group, Intel Corporation, Moscow, Russia
[3]Cray Programming Environments Performance Engineering, Cray Inc, Bloomington, Minnesota
[4]Parallel Computing Labs, Intel Corporation, Karnataka, India
[5]Data Center Group, Intel Corporation, Hillsboro, Oregon

Correspondence
Thorsten Kurth, National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, Berkeley, CA 94270.
Email: tkurth@lbl.gov

Funding information
Swiss National Supercomputing Center (CSCS); National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science of the U.S. Department of Energy, Grant/Award Number: DE-AC02-05CH11231; ALCF supported by DOE/SC, Grant/Award Number: DE-AC02-06CH11357; Intel Big Data Center at NERSC

Summary

Deep learning has proven to be a successful tool for solving a large variety of problems in various scientific fields and beyond. In recent years, the models as well as the available datasets have grown bigger and more complicated, and thus, an increasing amount of computing resources is required in order to train these models in a reasonable amount of time. Besides being able to use HPC resources, deep learning model developers want flexible frameworks which allow for rapid prototyping. One of the most important of these frameworks is Google TensorFlow, which provides both features, ie, good performance as well as flexibility. In this paper, we discuss different solutions for scaling the TensorFlow Framework to thousands of nodes on contemporary Cray XC supercomputing systems.

KEYWORDS

deep learning, performance, scalability

1 | INTRODUCTION

Deep learning has proven to be a powerful tool for solving challenging problems not only in a large variety of verticals in industry but also in many scientific fields, ie, scientific datasets grow rapidly in size as well as in complexity, and therefore, some data analytics tasks have become challenging problems, which can only be attacked by using high performance computing resources. For these problems, it is almost imperative to consider a distributed training approach of deep learning models. For many years, one unchallenged mantra of the HPC community has been that, in order to obtain scalable high-performance code, one has to resort to classical HPC languages such as Fortran or C/C++ and use efficient communication layers such as MPI.[1] Most deep learning frameworks however utilize python to provide flexibility to the model developer. Nevertheless, those frameworks usually provide highly optimized back-ends written in C/C++ for good single node performance. When it comes to distributed computing, however, many frameworks fall short in the sense that their distributed computing capabilities are inefficient, if at all available.

2 | DATA PARALLEL TRAINING

Due to its simplicity, the predominant parallelization scheme is data parallelism, ie, the global batch of input data will be split across the nodes and each node computes the forward pass fully independently on his shard of the batch. Using the loss obtained from the forward pass, each node will perform the back propagation process[2] by computing the relevant gradients locally. Before being incorporated into the model by the solver, the gradients have to be averaged over all the nodes. It is important to note that, in this update process, the local gradient computation can proceed without waiting for the reduction to complete. Only before starting the next forward pass for a given layer, the gradient reduction and incorporation has to be finished. In deep and compute intensive models, it is therefore possible to hide a big fraction of the communication time behind computation. This method of data parallel training is called *(fully) synchronous* training, as the model and gradients are always in sync between the nodes. Another method of distributed data parallel training is *asynchronous* training. There, a parameter server keeps track of the model weights and receives gradient updates from individual nodes and incorporates them into the model as well as ships off the updated model to the nodes in the order as they are received. This approach is more scalable than the synchronous training as there is no global synchronization involved. However, the stochastical convergence of the loss can be negatively impacted if the number of nodes participating in the asynchronous update is too large, because the number of *outdated* gradients incorporated into the model grows linearly with the number of participating nodes. We believe that most users will run distributed training processes on medium scale, ie, probably up to $\mathcal{O}(100)$ nodes for which the synchronous update is still feasible. We will therefore focus on synchronous training and discuss scalability of this approach to $\mathcal{O}(1000)$ nodes.

3 | TENSORFLOW

One of the most widely used frameworks today is TensorFlow.[3,4] It was developed by Google in 2015 and is maintained and continuously updated by implementing results of recent deep learning research. Therefore, TensorFlow supports a large variety of state-of-the-art neural network layers, activation functions, optimizers and tools for analyzing, profiling, and debugging deep neural networks. In order to deliver good single node performance, the computationally heavy kernels such as convolutions, dense matrix multiplications, etc, are using optimized libraries as backend. For example, TensorFlow can be compiled to use Intel MKL-DNN,[5] a library, which contains highly optimized deep learning primitives for recent Intel CPU architectures. On recent nvidia GPU architectures, TensorFlow can utilize cuDNN.[6,7] On modern architectures such as Intel Xeon Phi or nvidia P100 or V100 GPUs, those libraries are able to achieve a performance of multiple TFLOP/s on a single node for certain layers and layer parameters. A practitioner can develop and implement deep learning models in a sequential programming approach in Python, but in the execution phase, TensorFlow is highly asynchronous, ie, once a so-called `Session` is started, the computational graph for the entire calculation is constructed and executed in an asynchronous manner, eg, by executing independent edges of the graph in parallel or out-of-order. A distributed computing approach to TensorFlow needs to take this into account by not relying on a fixed order of certain collective operations such as global reductions or broadcasts. The frameworks, which we are going to discuss below, all satisfy this requirement and thus are suitable for distributed execution and training of TensorFlow models.

4 | DISTRIBUTED TENSORFLOW

TensorFlow provides a built-in mechanism for distributed training based on Google's GRPC protocol.[8] This protocol however is not suited for HPC architectures as it requires at least one server and utilizes network protocols such as TCP/IP. Designed for internet traffic, messages from these protocols have a large header and thus require more bandwidth than traditional HPC interconnect messages. Furthermore, they are forced into 64k frames and always routed through the kernel. Therefore, GRPC cannot make use of performance optimized interconnect features such as hardware atomics, RDMA, etc. A study investigating the scaling capabilities of TensorFlow with GRPC on the NERSC Cori System[9] showed rapid decrease of the per-worker efficiency on more than 128 nodes for ResNet-50.[10] In this paper, we will discuss more efficient alternatives to the GRPC approach which integrate seamlessly with TensorFlow and thus have the potential to offer good distributed training performance at minimal implementation overhead for the TensorFlow practitioner.

4.1 | Horovod

This framework[11-13] is developed by Uber and provides functions and function wrappers, which seamlessly integrate into the TensorFlow programming style. It uses MPI as communication backend and thus can benefit from any optimizations made in the underlying MPI library. The amount of code, which needs to be added in order to enable distributed training, is limited to a few lines of code. By default, Horovod uses OpenMPI,[14] but for efficiency, we re-compiled Horovod and linked it against Cray MPICH. In order to distinguish this Horovod build from other flavors, we will refer to it as *Horovod-MPI* in this study.

4.1.1 | Modus operandi

Horovod implements a gradient reduction cycle which run concurrently to the TensorFlow graph execution and is connected through hooks to the latter. Once TensorFlow reaches such a hook, it will push the tensor identifier to a queue managed by a Horovod background thread. Until the tensors belonging to these identifiers are reduced, all operations on those tensors are blocked. The TensorFlow scheduler is free to execute other tasks in the meanwhile, eg, such as continuing the local part of the back propagation step so that communication and computation can be overlapped. The background thread in the meanwhile executes a 5-ms cycle in which it will do the following.

1. It copies the queue and clears the original one so that the graph scheduler can push new items.
2. It sends the identifiers to rank 0 to notify the root that those tensors are ready for reduction.
3. It sends empty message to rank 0 to signal that the rank is ready for reductions.
4. Rank 0 gathers all tensor identifiers and *ready for reductions* messages. Once it has received all of the latter, it will parse through the tensor identifier buffer and look for tensors, which are ready for reduction on all nodes. Depending on the maximal fusion size in bytes as specified by the environment variable `HOROVOD_FUSION_THRESHOLD`, it will fuse multiple tensors into the same `MPI_Allreduce` call. It then broadcasts the identifiers of all tensors involved in the following reduction to all nodes together with information about which tensors will be fused. Rank 0 then calls `MPI_Allreduce` on these tensors.
5. All ranks receive the identifiers of the tensors for the following reduction, allocate buffers, and perform the requested `MPI_Allreduce` operations.
6. The nodes copy the reduced data back into the original buffers and notify the TensorFlow Scheduler that those tensors can be operated on. The cycle is continued.

For neural networks with only one sequential pass, this dynamic scheduling approach is over-complicated, as all reductions occur solely in order and thus can be statically scheduled. However, since TensorFlow can in theory perform out-of-order executions, eg, for neural networks with multiple parallel passes, a dynamic approach like the one implemented by Horovod is necessary. For whole process to work, Horovod relies on a thread-safe MPI implementation, ie, by default it asks for `MPI_THREAD_MULTIPLE` for multi-threading interoperability with `mpi4py`. However, in our case, only one thread is performing the actual communication, and thus, this can be lowered to `MPI_THREAD_SERIAL`.

4.1.2 | Using Horovod

Despite the complicated underlying dynamic scheduling approach, using Horovod in a TensorFlow program is very simple. In the best case, only three lines of code have to be changed to make the program multi-node ready. After importing the Horovod module for TensorFlow, `horovod.tensorflow` (referred to as `hvd` in the following), the MPI communicators needs to be initialized by calling `hvd.init()`. Next, the optimizer needs to be wrapped by `hvd.DistributedOptimizer`. This effectively overloads the `minimize` and `compute_gradients` function and inserts global reduction hooks in the appropriate places. In order to broadcast the model to all the nodes at beginning of the run, the broadcast hook `hvd.BroadcastGlobalVariablesHook(0)` needs to be added to the list of hooks. Finally, the `tf.Session()` statement needs to be replaced by `tf.train.MonitoredTrainingSession()`, which supports hooks and certain initializers, which are relevant to multi-node processing. The broadcast hook needs to be passed to that call. It is up to the user to split his dataset for distributed training but that can be achieved with standard python tools and usually does not need additional MPI calls. It is noteworthy that Horovod supports a global reduction call for numpy arrays so that performance metrics such as losses, accuracies, etc, can be averaged over the ranks before printing. Besides these functions for communication, Horovod implements convenience functions for querying the number of ranks and local rank id as well as global rank id.

4.2 | Horovod-MLSL

This is an experimental framework developed by Intel, which adopts Horovod's interface for portability/usability reasons but employs Intel® *Machine Learning Scaling Library* (MLSL)[15,16] as communication backend instead of MPI. It uses an optimized Horovod background thread logic for communicating tensors and tuning affinity settings. Intel® MLSL supports optimized deep learning communication primitives and further allows to spawning separate communication threads, which can perform message progression in the background. For instance, MLSL delivers 1.5x-2x performance improvement over MPI for Deepbench Allreduce benchmark.[17] Besides highly efficient communication, MLSL supports prioritization of latency-bound communication, ie, gradient exchange for the first layers of neural network topologies,[15] and message quantization. While we do not enable these optimizations for the results presented in this paper, we believe Horovod-MLSL has further potential to improve over Horovod-MPI and is being planned for public release. Please note that we refer to this experiment as Horovod-MLSL as no official name has been assigned yet.

4.2.1 | Using Horovod-MLSL

The modifications to the TensorFlow program are the same as for Horovod-MPI. The main difference is that additional steps have to be performed in the launch phase of the program if background processes steering the communication are used, ie, the number of those can be adjusted by the environment variable `MLSL_NUM_SERVERS`. If at least one background server is used, those have to be started also. At the time of writing of this paper, Cray MPI does not support dynamic spawning of MPI processes, and thus, these servers have to be launched manually using the `ep_server` script shipped with MLSL. Note that this is a Cray-specific limitation as process spawning is part of the MPI standard and supported by other widely used MPI implementations such as OpenMPI and Intel MPI.

4.3 | Cray Programming Environment (CPE) ML Plugin

The CPE ML Plugin is a deep learning framework portable communication plugin based on MPI. It couples itself to a TensorFlow graph with a provided custom TensorFlow operation (cf the TensorFlow Developer Documentation[18] for documentation on defining custom TensorFlow operations). Similar to Horovod, it is able to directly operate on in-graph memory. The API and usage differ from Horovod because it supports frameworks other than TensorFlow as well.

4.3.1 | Modus operandi

CPE ML is fully GPU/CPU address aware and for the case of GPUs manages a tunable number of CUDA streams for message buffering. The plugin manages a pool of helper threads organized into teams to progress communication independent from graph operations. There are no unique processes when using the CPE ML Plugin. Every MPI rank produces local gradients and contributes to the global reduction. This eliminates the need for parameter server processes, simplifies its usage, and results in overall efficient use of resources. Compared to Horovod, CPE ML requires more code modifications in the TensorFlow program but also provides more fine grained control over, eg, communication threads, message pipelining etc.

4.3.2 | Using CPE ML

The necessary modifications to the TensorFlow program are in the same spirit as those Horovod requires but it uses a different syntax. After importing `ml_comm` (referred to as `mc` in the following), the user needs to initialize the framework using `mc.init`. In contrast to Horovod, CPE ML requires pre-allocation of buffers, and thus, the user has to specify the approximate size of the required buffer size in Bytes. Additionally, the user can spawn multiple teams and threads per team in this call. This feature is especially useful when using gradient pipelining. The user can specify the number of total update steps per team and other useful features via `mc.config_team`. In contrast to Horovod, CPE ML does not provide convenience wrappers for variable broadcasting and gradient reduction yet. Therefore, the user has to implement his own session hook using `mc.broadcast` and gradient reduction step using `mc.gradients`. However, these commands are nicely integrated into the TensorFlow graph execution and can thus be used like any other TensorFlow operation. Other convenience routines for checking model consistency across ranks, querying rank number, or communicator size are also available.

5 | MODELS

We chose two neural network models for our performance study. Both are convolutional neural networks as that is what most of our users are training at the moment, but they differ in their communication to computation ratios. We will describe the two models in more detail below.

5.1 | HEP-CNN

The motivation behind this network is to solve the challenging task of distinguishing interesting events from uninteresting events measured by the ATLAS detector at the Large Hadron Collider (LHC) experiment at CERN. In the LHC, protons collide violently at almost light speed and will produce new particles in the scattering process. Those are detected by experiments such as ATLAS by measuring energy deposits in two different calorimeters (so-called *hits*) as well as reconstructed tracks through the tracking detector. To first approximation, the detector is cylindrical, and thus, the data can be unrolled into a 224×224 pixel, three channel image with one channel per calorimeter, and one for the tracking detector. One such snapshot is called event and it contains a pile-up of results from multiple collisions in a short readout time-interval. Interesting events are those which cannot be explained by the Standard Model of particle physics and those need to be distinguished from the events which can. In order to solve this task, the HEP-CNN model[19-21] implements a 6-layer convolutional binary classifier (5 convolution and pool units and one fully connected layer) and can be trained on labeled data generated by the fast Monte-Carlo event generator Delphes.[22] It is developed for scalability and thus does not use a big multi layer perceptron (MLP) for global feature correlation but instead it employs a lightweight convolutional layer and average pooling combination combined with a small fully connected layer. The interesting fact here is that the whole model is only about 2.3 MB in size and does not provide many opportunities for hiding communication behind computation. It can thus be used to test the latency sensitivity of the various distributed training frameworks. For more information on the network architecture, see the work of Kurth et al.[20]

5.2 | CosmoGAN

This network implements a deep convolutional generative adversarial network (DC-GAN) for producing cosmology mass maps.[23] These are usually obtained in expensive simulations and thus having a network, which can generate cheap surrogate examples, is mandatory for training inference workflows on cosmological parameter estimations. The GAN is comprised of two independent networks, ie, the generator and the discriminator. The former utilizes one linear layer for correlating the features of the 64-element random input vector with each other and then a stacked de-convolutional architecture comprised of four layers to scale that vector up to a full 256×256 pixel single channel image. The discriminator fundamentally implements a binary classifier by essentially reversing this architecture (except that the fully connected layer projects to a single value).

After each (de-)convolutional layer of generator and discriminator, a batch norm layer is inserted to improve precision. Since the network consists of sparse layers and only very lightweight dense layers, it is inherently scalable in a performant way. Compared to the previous example, this model has about $17 \cdot 10^6$ parameters and is thus much more compute intensive. It thus provides more opportunities for hiding communication and tests this aspect of the frameworks discussed above. For more information on CosmoGAN, cf the work of Mustafa et al.[23]

6 | EXPERIMENT SETUP

We will run a variety of experiments on two contemporary HPC systems, which will be described below.

6.1 | Cori (Phase II)

The Cori (Phase II) HPC system at NERSC[9] is a Cray XC40 supercomputer comprised of 9,688 self-hosted Intel Xeon Phi 7250 (Knight's Landing, KNL) compute nodes. Each KNL processor includes 68 cores running at 1.4 GHz and capable of hosting 4 HyperThreads for a total of 272 threads per node. Each core has two 512bit-wide vector units supporting the `MIC-AVX512` instruction set. Each node is equipped with 96 GB DDR4 and 16 GB high-bandwidth on-package memory. The processor can be configured in different numa and memory modes and we are using quadrant, cache mode in this paper. The nodes are connected with the diameter-5 dragonfly topology high speed low latency Aries interconnect. The single precision AVX peak performance for the whole system is approximately 50.6 PetaFLOP/s.

On the software side, we are using TensorFlow 1.5 with MKL-DNN 2017 backend and compile it using gcc 6.3.0. Note that we only activate `O3` and `AVX2` optimization flags in the Google Bazel[24] build system as most FLOPS come from the precompiled optimized MKL-DNN backend.

We build Horovod-MPI v0.11.2 against `cray-mpich` v7.6.2 with disabled hugepages support. Note that this MPI version has support for RDMA accelerated collectives and makes use of those under the hood. Horovod-MLSL is based on Horovod v0.11.2 and uses MLSL 2018 Preview as communication layer. The Cray ML plugin `craype-ml-plugin` used is v1.1.0. Both Horovod-MPI and CPE ML Plugin do not need special binding, and thus, we perform runs with those on 66 threads in spread binding by setting `OMP_NUM_THREADS` to 66, `OMP_PLACES` to `threads`, and `OMP_PROC_BIND` to `spread`. For CPE ML, we are employing one team with a single thread for HEP-CNN and two teams with a single thread each for CosmoGAN. The latter model runs independent optimizers for each of the two sub-networks and we investigate whether utilizing two independent communication teams will improve the performance. In case of Horovod-MLSL, we use `numactl` to bind threads in a way to avoid the cores occupied by the background processes from Horovod and MLSL. In that case, we have only 64 threads left for the TensorFlow runtime as we avoid using Hyper-Threading. The environment variable `MLSL_NUM_SERVERS` is either set to 0 or 2 for our tests and defines how many background processes are used in order to drive message progression for MLSL. Since on Cray systems `MPI_Comm_Spawn` is not supported at the time of our experiments, we need to start these background services manually. We do this by specifying `–gres=craynetwork:2` as batch environment variable in order to obtain two network resources, and then launch the servers on resource 1 and TensorFlow on resource 0.

6.2 | Piz Daint

Piz Daint at CSCS[25] is a hybrid Cray XC40/XC50 system, and in this section, we are referring to the XC50 portion of the machine. This part is comprised of 5320 hybrid CPU+GPU nodes. The CPU are single-socket Intel Xeon E5-2695v3 with 12 hardware cores, which can host 2 HyperThreads each at 2.6 Ghz. Each node has 64 GB of DDR memory and is further equipped with one nvidia Pascal GPU (P100) with 16 GB HBM2 memory and 32 GB/s PCIe bandwidth. The inter-node network is the same as the one deployed in Cori. The single precision peak performance is about 49.5 PetaFLOP/s.

We compile TensorFlow 1.6 with cuDNN 7.1.1 backend for CUDA 8.0 using gcc 5.3.0. Horovod-MPI v0.12.0 is compiled with `cray-mpich` v7.6.0. with disabled hugepages support. We enable GPU-aware collectives by specifying `HOROVOD_GPU_{ALLGATHER,ALLREDUCE,BROADCAST}` at compile time and enable the corresponding Cray MPICH flags at runtime.

For testing purposes, we also compiled Horovod against the nvidia collective communications library NCCL[26] v2.1.4 and refer to this variant as Horovod-NCCL. Due to the unavailability of a Horovod-MLSL build, we do not include this framework in our measurements on the Piz Daint system. In case of the CPE ML plugin, we used v1.1.0.

Note that some of the software versions differ between Cori and Piz Daint, but the main intention of this paper is to compare the various distributed training frameworks on a given architecture and not cross architectural performance comparisons.

6.3 | Training Process

If not stated otherwise, we train both models using the ADAM optimizer[27,28] without large-batch gradient booster improvements such as Layer-Wise Adaptive Rate Scaling (LARS[29]). The latter would improve the accuracy of the trained model but does not impact scaling performance significantly. Therefore, we would not consider using LARS in this paper.

7 | RESULTS

We perform weak and strong scaling experiments as well as a time to solution study. Strong scaling a model is the easiest way of training a model, which was tweaked for optimal single node convergence in a distributed fashion. However, deep learning models usually do not scale out efficiently when the local batch is distributed across more and more nodes as local parallelism drops rapidly. Nevertheless, we think it is important to test strong scaling capabilities of distributed deep learning frameworks. Therefore, the best way to scale a deep learning process is weak scaling, in which the local batch size is kept constant when adding more nodes to the training process. We will investigate weak scaling capabilities of the frameworks discussed above. Lastly, we will do a small time to solution study to illustrate that scaling a deep learning training process weakly to a small number of nodes can speed up the training process even if no sophisticated large-batch-optimized training algorithms are employed.

7.1 | Weak scaling

We are performing weak scaling runs for the two deep learning models on the two systems described above. In this experiment, we perform calculations with node-local batch sizes of 128 and 64 for the HEP-CNN and CosmoGAN model, respectively. Starting from single node, we scale in powers of two as far as possible. For each run, we measure the timing per training step over a number of steps in order to obtain a good estimate of the duration per step. From that timing, we can compute the duration per sample as well as total number of samples per second. We compute the median and the central 68% confidence interval and take those as our estimate and error bar respectively.

Since this is a weak scaling experiment, the ideal/perfect scaling case would correspond to constant duration and linear increase in total number of samples per second. Note that the total size of the dataset is kept constant in this experiment. This seems to be contradictory as this implies strong scaling with respect to the global dataset. However, this is the typical situation a deep learning practitioner is facing and thus more representative than scaling the size of the dataset with the number of nodes. Furthermore, ignoring file caching effects on small local shards of the total dataset, the total performance is mainly influenced by the local batch size and the computation to communication ratio.

Our weak scaling results are shown in Figure 1. The performance of the various frameworks is shown in a color consistent manner across architectures and models. The black *ideal scaling* line was computed based on the best achieved per-step performance across the frameworks on the given architecture. This number does not necessarily correspond to the single node performance, ie, the reason for that is that there is a tradeoff between file caching effects and communication overhead. The plot exhibits that the HEP-CNN model is harder to scale because of lower compute

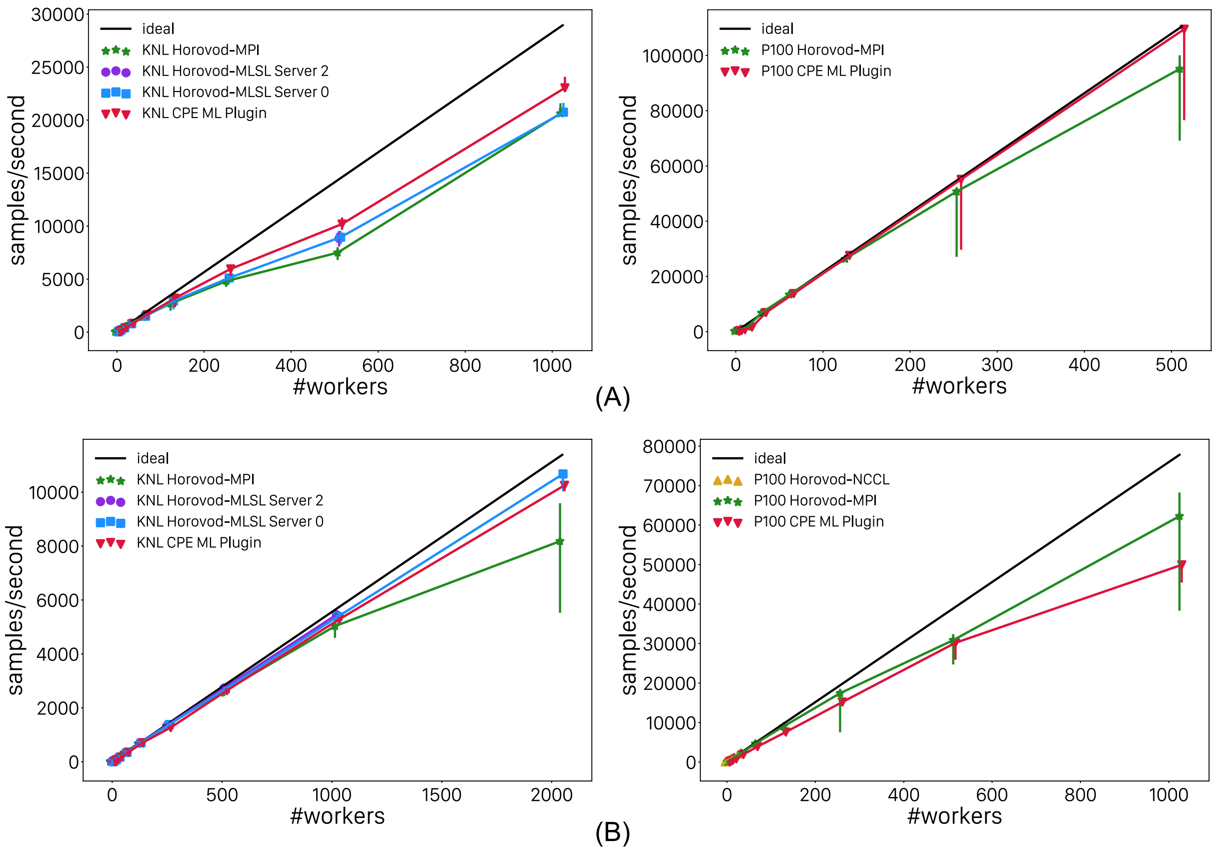

FIGURE 1 Weak scaling of HEP-CNN (top row) CosmoGAN (bottom row) and on the Cori (left column) and Piz Daint (right column) HPC systems. The different colors correspond to the various frameworks. The datapoints are shifted in x-direction to improve readability. The black line depicts the ideal scaling, based on the best achieved performance across frameworks and concurrencies for the given HPC system

to communication ratio. In this regime, CPE ML outperforms all other solutions by a small margin. This is probably due to the reduced overhead as this frameworks makes heavy use of RDMA and other asynchronous optimizations. Horovod-MLSL is better than Horovod-MPI, but it is not beneficial to launch additional background servers here. We are currently investigating this and believe that the communication of the main TensorFlow runtime with the background servers generates some overhead, which is visible in this very latency and jitter sensitive environment. For the more compute intensive CosmoGAN, Horovod-MLSL performs very well even at massive scale and even better when background servers are enabled: it achieves almost ideal scaling up to 1024 workers on Cori. On 2048 workers, the sustained performance of Horovod-MLSL is about 5% higher than CPE ML. However, this difference is less important when performance variability is taken into account. Both solutions outperform Horovod-MPI at that scale by being about 20% more efficient.

On Piz Daint, CPE ML outperforms Horovod-MPI at very large scale for HEP-CNN and shows almost ideal scaling. We observe that, however, Horovod-MPI performs well at medium to large scale. Our experiments with NCCL delivered promising results at small scales but deadlocked for concurrencies greater than 64 nodes so that we were unable to assess its true potential. Nvidia is aware of these issues and working on solving this problem. For CosmoGAN, Horovod-MPI surprisingly showed better scalability on Piz Daint. It might be beneficial to tweak the number of teams and number of threads per team setting slightly to improve performance of CPE ML. Since the computing workload is offloaded to the GPU, most of the CPU cores would be available for communication. We will leave this experiment for a future study.

In terms of performance variation, we see that Horovod-MLSL shows consistent performance even at 2048 Cori nodes, whereas Horovod-MPI shows stronger fluctuations. The overall performance variability on Piz Daint is larger, potentially because a CPU-GPU system is more sensitive to jitter than a CPU only system. In a future study, we also aim at collecting Horovod-MLSL data on Piz Daint as soon as this framework is available on the system.

7.2 | Strong scaling

We perform a strong scaling study for CosmoGAN on Cori, starting with a node-local batch size of 256 samples on single node. This is the largest power-of-two batch we could fit on a single Xeon Phi node, and for this size, the network still converges smoothly. We then double the number of nodes and half the node-local batch and scale out to 128 nodes. On that concurrency, a single node is working on a batch of size 2. We measure the achieved throughput dependent on the number of nodes using various scaling frameworks. Since we expect that communication will be the limiting factor for small local batch sizes, we employ 2 servers in Horovod-MLSL. Figure 2 shows the achieved throughputs.

The plot illustrates that training a model with small local batches yields low computational performance. We could not achieve significant speedup beyond 8 and 16 nodes for CPE ML and the Horovod-based frameworks, respectively. However, at a low degree of parallelism, ie, up to 8 nodes, super-linear speedup could be achieved. One possible explanation for this could be lower memory pressure and improved cache reuse. Another reason could be that the convolutional kernels benefit from higher vectorization efficiency, ie, in MKL-DNN convolutions, the samples are vectorized over batch and filter dimensions and thus batch sizes/strides of 32 or 16 floating points numbers can efficiently fill the vector processing units on Xeon Phi. However, this also suggests that the originally chosen single node batch size of 64 is not far from delivering optimal single node performance.

7.3 | Time to Solution

Another important experiment is to print the time to solution, eg, the time to reach a certain loss in the deep learning context. The fact whether a network converges or not heavily depends on the network architecture and the hyper parameters as well as the solver chosen for the optimization task. In that sense, it is more of a design and algorithm question and not dependent on a specific distributed computing framework. However, the practitioner who needs to decide whether to employ distributed training or not wants to know if it is beneficial at all. GAN is notoriously fragile and

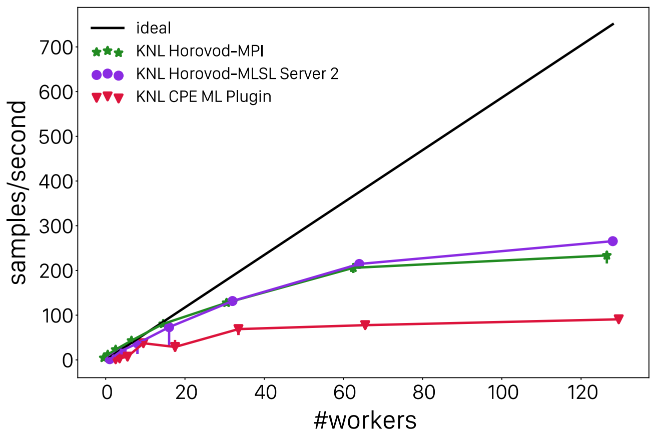

FIGURE 2 CosmoGAN strong scaling with global batch of size 128 for our three main frameworks

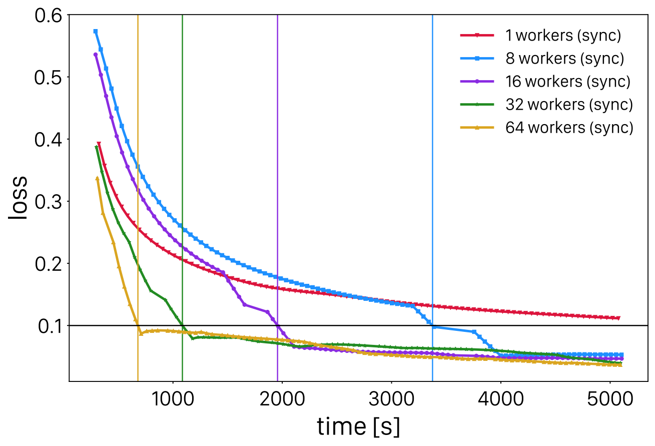

FIGURE 3 Loss curves for the HEP-CNN model on the Cori HPC system obtained using the CPE ML plugin. The vertical colored lines denote the time at which the corresponding loss curve reaches the target loss of 0.1 (black horizontal line). The loss curves do not look smooth because they were computed by applying moving averages over an epoch and resetting the loss at epoch boundaries. To weaken this effect, we applied another moving average with a small window of 10 points

hard to train even on a single node and require a lot of parameter tuning. Therefore, we will answer this question for the more robust HEP-CNN model instead, ie, we weak-scaled the network from a single node to 8, 16, 32, and 64 Cori nodes and computed the loss in dependence on wall clock time. We are only using the CPE ML framework for this study, but due to the small concurrencies, we expect the results to be similar for all the frameworks considered in this paper. The results shown in Figure 3 exhibit that the training could be accelerated by involving more nodes in the calculation. Picking a target value of 0.1 for the loss (black horizontal line), we can compute how long it takes for the individual configurations to reach this value in their minimization process (vertical colored lines). Since the single node training does not reach the target value in the observed time, we normalize the efficiency by the time obtained on 8 nodes. For that case, we achieve efficiencies of 86%, 78%, and 62%, respectively. Note that conventional ADAM was used as the optimizer in this study and we expect these efficiencies to improve when large-batch-optimized solvers such as LARS would be used.

8 | CONCLUSION

We have tested several python modules, which transform a single node TensorFlow program into a model, which can be trained in data parallel fashion. Even the original Horovod-MPI shows good weak scaling up to 1024 workers on the Cori and Piz Daint systems. We can thus recommend this framework without any doubts to users who want to perform small and medium scale distributed training runs. The CPE Machine Learning Plugin is very promising and shows mostly excellent performance even large massive scale. It is rapidly evolving and supports other advanced features, which we have not tested, such as gradient pipelining and solver cool down, for faster training and better convergence. We can recommend this software to users who predominantly run on their training on Cray systems. However, since the module interface is different from Horovod-based solutions and it is only available on Cray architectures; codes using CPE ML are not portable to systems from other vendors. Users looking for a solution, which shows excellent performance at large scale, which is additionally portable to other systems, should have a look at Intel's Horovod-MLSL solution. Since the interface is compatible to Horovod, Tensorflow code written for Horovod-MLSL is interoperable with any other framework, which adopted the Horovod API. A special case is Horovod-MPI with NCCL integration. On Piz Daint, we could not see a benefit over Horovod-MPI, but that is expected as Piz Daint nodes only contain a single GPU. However, we expect that, on systems with fat nodes, hosting multiple GPUs such as the OLCF Summit system[30] could benefit highly from this solution. Users who plan to run on these architectures are encouraged to try this solution as well.

ACKNOWLEDGMENTS

This work was supported by the Swiss National Supercomputing Center (CSCS). This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported by the Office of Science of the US Department of Energy under contract DE-AC02-05CH11231, and of the ALCF, which is supported by DOE/SC under contract DE-AC02-06CH11357. Part of this research was supported by the Intel Big Data Center at NERSC.

ORCID

Thorsten Kurth http://orcid.org/0000-0003-0832-6198

REFERENCES

1. The message passing interface (MPI) standard. http://www.mcs.anl.gov/research/projects/mpi/index.htm
2. Amari S-I. Backpropagation and stochastic gradient descent method. *Neurocomputing*. 1993;5(4-5):185-196.
3. Abadi M, Agarwal A, Barham P, et al. TensorFlow: large-scale machine learning on heterogeneous systems software. 2015.
4. TensorFlow. https://tensorflow.org
5. Pirogov V, Gennady F. Introducing DNN primitives in Intel® Math Kernel Library. 2016. https://software.intel.com/en-us/articles/introducing-dnn-primitives-in-intelr-mkl
6. Chetlur S, Woolley C, Vandermersch P, et al. cuDNN: efficient primitives for deep learning. 2014; arXiv preprint arXiv:1410.0759.
7. NVIDIA cuDNN GPU accelerated deep learning. https://developer.nvidia.com/cudnn
8. GRPC. https://grpc.io
9. NERSC Cori. https://www.nersc.gov/users/computational-systems/cori/
10. Mathuriya A, Kurth T, Rane V, et al. Scaling GRPC TensorFlow on 512 nodes of cori supercomputer. 2017; arXiv e-prints.
11. Sergeev A, Del Balso M. Horovod: fast and easy distributed deep learning in TensorFlow. 2018; arXiv preprint arXiv:1802.05799.
12. Sergeev A. Horovod - distributed TensorFlow made easy. 2017. https://www.slideshare.net/AlexanderSergeev4/horovod-distributed-tensorflow-made-easy
13. Sergeev A, Del Balso M. Meet Horovod: Uber's open source distributed deep learning framework for TensorFlow. 2017; https://eng.uber.com/horovod/
14. Open MPI: open source high performance computing. https://www.open-mpi.org
15. Sridharan S, Vaidyanathan K, Kalamkar D, et al. On scale-out deep learning training for cloud and HPC. Paper presented at: SysML 2018 Conference; 2018; Stanford, CA.
16. Intel® Machine Learning Scaling Library for Linux* OS. 2017. https://github.com/01org/MLSL
17. DeepBench. https://svail.github.io/DeepBench/
18. Adding a new OP. https://www.tensorflow.org/extend/adding_an_op
19. Bhimji W, Farrell SA, Kurth T, Paganini M, Prabhat, Racah E. Deep neural networks for physics analysis on low-level whole-detector data at the LHC. Paper presented at: ACAT 2017 Conference; 2017; Seattle, WA.
20. Kurth T, Zhang J, Satish N, et al. Deep learning at 15PF: supervised and semi-supervised classification for scientific data. 2017; arXiv:1708.05256 [cs.PF].
21. HEP-CNN benchmark repository. https://github.com/NERSC/hep_cnn_benchmark
22. de Favereau J, Delaere C, Demin P. DELPHES 3: a modular framework for fast simulation of a generic collider experiment. *J High Energy Phys*. 2014;2:57.
23. Mustafa M, Bard D, Bhimji W, Al-Rfou R, Lukić Z. Creating virtual universes using generative adversarial networks. 2017; arXiv e-prints.
24. Bazel. https://bazel.build
25. Piz Daint. https://www.cscs.ch/computers/piz-daint/
26. NCCL. https://developer.nvidia.com/nccl
27. Kingma DP, Ba J. Adam: a method for stochastic optimization. 2014; arXiv:1412.6980.
28. Chilimbi T, Suzue Y, Apacible J, Kalyanaraman K. Project Adam: building an efficient and scalable deep learning training system. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation; 2014; Broomfield, CO.
29. You Y, Gitman I, Ginsburg B. Large batch training of convolutional networks. 2017; arXiv e-prints.
30. Summit. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

How to cite this article: Kurth T, Smorkalov M, Mendygral P, Sridharan S, Mathuriya A. TensorFlow at Scale: Performance and productivity analysis of distributed training with Horovod, MLSL, and Cray PE ML. *Concurrency Computat Pract Exper*. 2018;e4989. https://doi.org/10.1002/cpe.4989