Performance, Power, and Scalability Analysis of the Horovod Implementation of the CANDLE NT3 Benchmark on the Cray XC40 Theta

Xingfu Wu[1], Valerie Taylor[1], Justin M. Wozniak[2], Rick Stevens[3], Thomas Brettin[4] and Fangfang Xia[3]

[1]*Mathematics and Computer Science Division, Argonne National Laboratory and University of Chicago*

[2]*Data Science and Learning Division, Argonne National Laboratory and University of Chicago*

[3]*Computing, Environment, and Life Sciences Directorate, Argonne National Laboratory and University of Chicago*

[4]*Computing, Environment, and Life Sciences Directorate, Argonne National Laboratory*

Email: {`xingfu.wu, vtaylor, woz, stevens, brettin, fangfang`}`@anl.gov`

***Abstract*—Training scientific deep learning models requires the large amount of computing power provided by HPC systems. In this paper, we use the distributed deep learning framework Horovod to parallelize NT3, a Python benchmark from the exploratory research project CANDLE (Cancer Distributed Learning Environment). We analyze NT3's scalability, performance, and power characteristics with different batch sizes and learning rates under two memory modes, cache and flat, on the DOE pre-exascale production system Cray XC40 Theta at Argonne National Laboratory. Our experimental results indicate that the power profiles for the node, CPU, and memory are useful in showing how the Horovod NT3 benchmark behaves on the underlying system. Using the communication timeline of this benchmark, we found that the Horovod communication overhead in NT3 increases significantly with the number of nodes although Horovod has the ability to scale up. The benchmark leads to smaller runtime and lower power consumption for the node and CPU under the cache mode than under the flat mode. Furthermore, increasing the batch size leads to a runtime decrease and slightly impacts the power. Increasing the learning rate results in a slight decrease in runtime and node power and an increase in accuracy. Several issues raised by the Horovod NT3 benchmark results are discussed, and suggestions are proposed for further work.**

1. Introduction

Training modern deep learning models requires the large amount of computing power provided by high-performance computing (HPC) systems. TensorFlow [2] [22] is one of the most widely used open source frameworks for deep learning; it supports a wide variety of deep learning uses, from conducting exploratory research to deploying models in production on cloud servers, mobile apps, and even self-driving vehicles [20]. Horovod [11] [20], developed by Uber, is a distributed training framework for TensorFlow and Keras [12]. In this work, we use Horovod to parallelize NT3, a Python-based benchmark [6] from the exploratory research project CANDLE (Cancer Distributed Learning Environment) [4]. We then analyze the Horovod implementation of NT3 in terms of performance, power, and scalability on the DOE pre-exascale production system Cray XC40 Theta [9] at Argonne National Laboratory.

The CANDLE project [4] [25] focuses on building a single scalable deep neural network code that can address three cancer challenge problems: the RAS pathway problem, understanding the molecular basis of key protein interactions in the RAS/RAF pathway presented in 30% of cancers; the drug response problem, developing predictive models for drug response to optimize preclinical drug screening and drive precision-medicine-based treatments for cancer patients; and the treatment strategy problem, automating the analysis and extraction of information from millions of cancer patient records to determine optimal cancer treatment strategies. CANDLE benchmark codes [5] implement deep learning architectures that are relevant to these three cancer problems. The NT3 benchmark [6] is one of the Pilot1 benchmarks [5] that are formed from problems and data at the cellular level. The goal behind these Pilot1 benchmarks is to predict the drug response based on molecular features of tumor cells and drug descriptors.

The NT3 benchmark, like other CANDLE benchmarks, is implemented in Python by using the Keras framework. Python allows for the rapid development of the application. It also enables code reuse across the CANDLE benchmarks, since each benchmark uses common Python-based CANDLE utilities and each benchmark implements a common interface used by higher-level Python-based driver systems, such as the CANDLE/Supervisor framework for hyperparameter optimization [25]. These benchmarks, which are intended to run on exascale systems as they emerge, are currently being tested on pre-exascale systems such as Theta. These pre-exascale systems feature new hardware at ever greater scale, requiring new analysis of performance and power to determine how best to use them. Deep learning is expected to play a greater role in scientific computing on systems such as Summit [21]. Thus, it is critical for studying the performance and power usage of the whole application stack, including the scripting level, numerics, and communication.

To speed TensorFlow applications by utilizing large-scale supercomputers such as Theta requires a distributed TensorFlow environment. Currently, TensorFlow has a native method for parallelism across nodes using the gRPC layer in TensorFlow based on sockets [1] [10], but this is difficult to use and optimize [15] [20]. The performance and usability issues with the distributed TensorFlow can be addressed, however, by adopting an MPI communication model. Although TensorFlow has an MPI option, it replaces only point-to-point operations in gRPC with MPI and does not use MPI collective operations. Horovod adapts the MPI communication model by adding an allreduce between the gradient computation and model update, replacing the native optimizer with a new one called the Distributed Optimizer. No modification to TensorFlow itself is required; the Python training scripts are modified instead. The Cray programming environment machine learning plugin (CPE ML Plugin) [15], like Horovod, does not require modification to TensorFlow, but it is designed for Cray systems and is not available to the public. Therefore, we chose Horovod for this investigation.

Related work with Horovod and TensorFlow has been reported in the literature. A. Sergeev and M. Del Balso [20] designed and developed Horovod, and they used the TensorFlow benchmarks [23] such as Inception V3 and ResNet-101 to compare the performance (images per second) of the Horovod implementations with standard distributed TensorFlow on different numbers of NVIDIA Pascal GPUs. They observed larger improvements in Horovod's ability to scale, and the training in the Horovod implementation was about twice as fast as standard distributed TensorFlow. P. Mendygral et al. [15] discussed the Horovod-like Cray CPE ML Plugin, and they used TensorFlow benchmarks such as Inception V3 and ResNet50 to compare the performance (samples per second) of the CPE ML Plugin implementations with standard distributed TensorFlow with gRPC and Horovod on a Cray XC40 system. They observed that the CPE ML Plugin outperformed both Horovod and standard distributed TensorFlow. They also discussed convergence considerations at scale in deep learning and presented square and linear learning rate scaling rules. They found that the scaling rules are more attractive (when they work) because they do not require many additional iterations to reach the same accuracy.

In this paper, we analyze the Horovod parallel implementation of the NT3 benchmark, focusing on its scalability, performance, and power characteristics with different batch sizes and learning rates under two memory modes, cache and flat (a high bandwidth on-package memory Multi-Channel DRAM can be configured as a shared L3 cache (cache mode) or as a distinct NUMA node memory (flat mode)), on the Cray XC40 Theta. Our experimental results indicate that power profiling for the node, CPU, and memory is useful for showing how the Horovod NT3 benchmark behaves on the system. Using the communication timeline of this benchmark, we find that the Horovod communication overhead in NT3 increases significantly with the number of nodes. The benchmark leads to smaller runtime and lower power consumption for the node and CPU under cache mode than under flat mode. Furthermore, increasing the batch size leads to a runtime decrease and slightly impacts the power; and increasing the learning rate results in a slight decrease in runtime and node power and an increase in accuracy.

This work makes the following contributions.

- We use Horovod to parallelize the CANDLE NT3 benchmark. This parallelization method can be applied to other CANDLE benchmarks such as the Pilot1 and Pilot3 benchmarks in the similar way.
- We analyze the scalability of the Horovod implementation of the NT3 benchmark with weak scaling, and we discuss the Horovod overhead.
- We investigate the performance and power characteristics of the Horovod implementation of the NT3 benchmark with strong scaling, and we use power profiling to analyze how parameters such as the learning rate and batch size affect the performance and power.

The remainder of this paper is organized as follows. Section 2 briefly describes the CANDLE NT3 benchmark and Horovod and then discusses the Horovod implementation. Section 3 depicts the system platform Cray XC40 Theta. Section 4 analyzes the scalability of the Horovod implementation of the NT3 benchmark with increasing numbers of nodes. Section 5 uses the experimental results to analyze performance and power characteristics of the NT3 benchmark. Section 6 summarizes our conclusions and discusses future work.

2. CANDLE NT3 Benchmark and Its Horovod Implementation

In this section, we briefly describe the CANDLE NT3 benchmark and the distributed deep learning framework Horovod. We then discuss the Horovod implementation of the benchmark in detail.

2.1. CANDLE NT3 Benchmark

The CANDLE NT3 benchmark [6] is written in Python and Keras, which is a high-level neural network API written in Python and capable of running on top of TensorFlow, CNTK [16], or Theano [24]. This benchmark is a 1D convolutional network for classifying RNA-seq gene expression

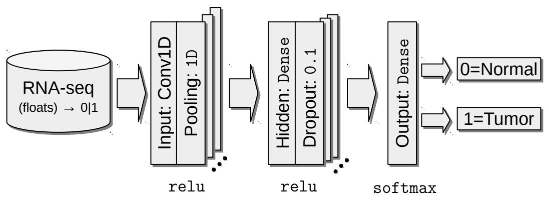


Figure 1. Schematic of NT3 neural network architecture.

profiles into normal or tumor tissue categories. This network follows the classic architecture of convolutional models with multiple 1D convolutional layers interleaved with pooling layers followed by final dense layers. This model is trained on the balanced 700 matched normal-tumor gene expression profile pairs available from the NCI Genomic Data Commons and acts as a quality control check for synthetically generated gene expression profiles. The full dataset of expression features contains 60,483 float columns transformed from RNA-seq FPKM-UQ values [6] that map to a column that contains the integer 0|1. Hence the NT3 benchmark is useful for studying the difference and transformation of latent representation between normal and tumor tissues.

The NT3 architecture is depicted in Figure 1. The input data is first processed by a variable number of pairs of convolution-pooling layers, controlled by parameters `conv`, with layer unit counts specified by the numbers in the `conv` list. `relu` is used as the activation function. Then, the signal flows into a various number of pairs of dense-dropout layers controlled by the `dense` list and `dropout` specifier (`dropout` may be omitted for no dropouts). Finally, a `softmax` function is used to obtain the output, and scores for Normal and Tumor which add to 1. The NT3 benchmark entails four phases: data loading, preprocessing, basic training and cross-validation, and prediction and evaluation on test data. It uses the following main global parameters.

```
data_url = 'ftp://ftp.mcs.anl.gov/pub/candle/public/
benchmarks/Pilot1/normal-tumor/'
train_data = 'nt_train2.csv'
test_data = 'nt_test2.csv'
model_name = 'nt3'
conv=[128, 20, 1, 128, 10, 1]
dense=[200,20]
activation='relu'
out_act='softmax'
loss='categorical_crossentropy'
optimizer='sgd'
metrics='accuracy'
epochs=400
batch_size=20
learning_rate=0.001
drop=0.1
classes=2
pool=[1, 10]
save='.'
```

The size of the training data file `nt_train2.csv` is 597MB, and the size of the test data file `nt_test2.csv` is 149MB. The benchmark uses the importing data function `pandas.read_csv()` [18] with FTP to remotely read the data files; the optimizer is SGD (stochastic gradient descent); the number of epochs is 400, where for each epoch the model training has a full pass through the whole dataset and the key parameters (accuracy, loss, ...) are saved for retraining the model in the next epoch; the batch size is 20; and the learning rate is 0.001.

2.2. Horovod

Horovod [11] [20] is a distributed training framework for TensorFlow and Keras and is a stand-alone Python package developed by Uber. The goal of Horovod is to make distributed deep learning fast and easy to use. The core principles of Horovod are based on MPI concepts such as size, rank, local rank, allreduce, allgather, and broadcast; and it is implemented by using MPI subroutines. A unique feature of Horovod is its ability to interleave communication and computation. Moreover, it is able to batch small allreduce operations by combining all the tensors that are ready to be reduced at a given moment into one reduction operation, an action that results in improved performance. The Horovod source code is based on the Baidu tensorflow-allreduce repository [3]. Horovod provides MPI-based data parallelism for TensorFlow. In its examples [11], it provides the parallelization at the epoch level (keras_mnist.py) and at the batch step level (keras_mnist_advanced.py).

2.3. Using Horovod to Parallelize the NT3 Benchmark

As described in [11], to use Horovod, we made the following additions to the NT3 benchmark to utilize CPUs:

- Add `import horovod.tensorflow as hvd` to import the Horovod package.
- Add `hvd.init()` to initialize Horovod.
- Obtain the size (`hvd.size()`) and rank (`hvd.rank()`), and adjust the number of epochs based on the number of CPU nodes used as follows.

```
nprocs = hvd.size()
myrank = hvd.rank()

def comp_epochs(n, myrank=0, nprocs=1):
        j = int(n // nprocs)
        k = n % nprocs
        if myrank < nprocs-1:
                i = j
        else:
                i = j + k
        return i

epochs =
  comp_epochs(gParameters['epochs'],
              myrank, nprocs)
```

  We use `comp_epochs()` to calculate the number of epochs for each node. For load balancing, we ensure that the number of epochs is the same for each node.
- Scale the learning rate by the number of workers. We scaled the learning rate to `learning_rate` $\times$ `hvd.size()`. We properly increased the batch size as well.

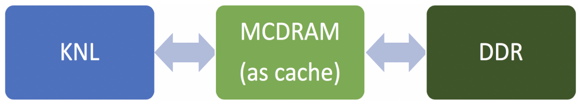


Figure 2. Cache mode on Cray XC40 Theta

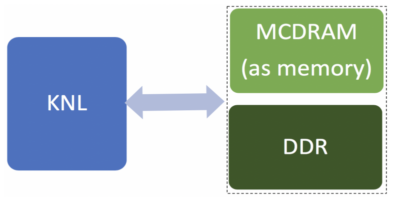


Figure 3. Flat mode on Cray XC40 Theta

- Wrap the original optimizer in the Horovod distributed optimizer using `optimizer = hvd.DistributedOptimizer(optimizer)`. The distributed optimizer delegates the gradient computation to the original optimizer, averages gradients using `MPI_Allreduce()`, and then applies those averaged gradients.
- Add `hvd.BroadcastGlobalVariablesHook(0)` to the callbacks to broadcast initial variable states from rank 0 to all other processes. This step ensures consistent initialization of all workers when training is started with random weights.

3. System Platform: Cray XC40 Theta

We conducted our experiments on the Cray XC40 Theta [9], which is a pre-exascale production system at Argonne National Laboratory. Each Cray XC40 node has 64 compute cores (one Intel Phi Knights Landing (KNL) 7230 with the thermal design power (TDP) of 215 W), shared L2 cache of 32 MB (1 MB L2 cache shared by two cores), 16 GB of high-bandwidth in-package memory, 192 GB of DDR4 RAM, and a 128 GB SSD. The Cray XC40 system uses the Cray Aries dragonfly network with user access to a Lustre parallel file system with 10 PB of capacity and 210 GB/s bandwidth. Cray XC40 [8] [14] provides power management to operate more efficiently by monitoring, profiling, and limiting the power usage. In this work, we use a simplified PoLiMEr library [13], which utilizes Cray's CapMC [14] to measure power consumption for the node, CPU, and memory at the node level on Theta. The power sampling rate used is approximately two samples per second (default). In a Python code, we import ctypes to export the CDLL for loading the shared PoLiMEr library in order to measure the power for the code.

Each XC40 node has one Intel KNL, which brings in new memory technology, an on-package memory called Multi-Channel DRAM (MCDRAM) in addition to the traditional DDR4 RAM. MCDRAM has a high-bandwidth (around 4 times more than DDR4 RAM) and low-capacity (16 GB) memory. MCDRAM can be configured as a shared

TABLE 1. RUNTIME (S), POWER (W), AND ENERGY (J) OF THE SINGLE-NODE NT3 BENCHMARK WITH DIFFERENT BATCH SIZES UNDER CACHE MODE.

Batch Size	Runtime	Node Power	CPU Power	Memory Power	Energy
20	1567	159.21	105.26	12.08	249,560.42
50	1497	159.35	99.42	12.12	238,546.95
100	1476	158.88	101.04	11.60	234,506.88
150	1469	160.54	100.22	12.12	235,833.26
200	1462	160.38	101.86	12.14	234,475.56

L3 cache (cache mode) shown in Figure 2 or as a distinct NUMA node memory (flat mode) shown in Figure 3. For the flat mode, the default memory allocation preference is DDR4 first, then MCDRAM. With the different memory modes by which the system can be booted, understanding the best mode for an application becomes challenging for the user.

4. Scalability Analysis

In this section, we investigate the performance and power characteristics of the original NT3 benchmark under different memory modes. We then analyze the scalability of the Horovod NT3 benchmark for our weak-scaling study.

4.1. Original NT3 Benchmark under Different Memory Modes

The number of epochs is 400 for the NT3 benchmark. For simplicity, in this section we use just one epoch to conduct the experiments under a learning rate of 0.001 and a batch size of 20.

Figure 4 shows power over time on Theta, with cache mode on one node and epochs = 1. The runtime is 1567s, and the average power is 159.21 W for the node, 105.26 W for the CPU, and 12.08 W for memory. We observe that NT3 takes around 800s to do the data loading and preprocessing because of the FTP remote data access used by `pandas.read_csv()` in the benchmark.

Figure 5 shows power over time on Theta (with flat mode on one node and epochs = 1). The runtime is 1608s, and the average power is 164.37 W for the node, 110.37 W for the CPU, and 17.0 W for memory. Comparing the cache mode with the flat mode, we observe that the NT3 benchmark under cache mode results in better performance and lower power consumption in the node and CPU. The memory power consumption is lower because the MCDRAM configured as the L3 cache is enough to hold both data files in the cache.

In the experiments, we used an initial batch size of 20 as default. We then changed the batch size to determine its effect on the performance and power of NT3. Table 1 shows the runtime, power, and energy of the benchmark with different numbers of batch sizes under cache mode. We ran the same experiment several times to ensure the

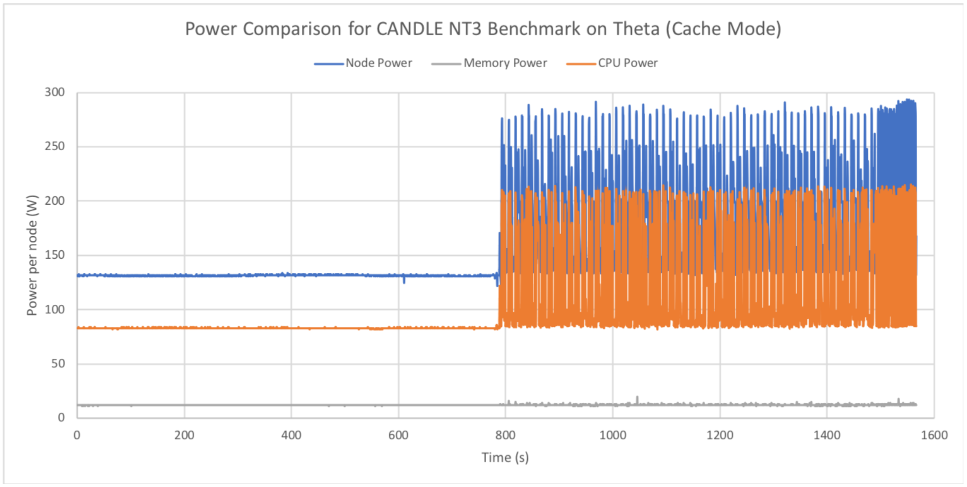


Figure 4. Power over time on Theta (with cache mode on one node and epochs = 1)

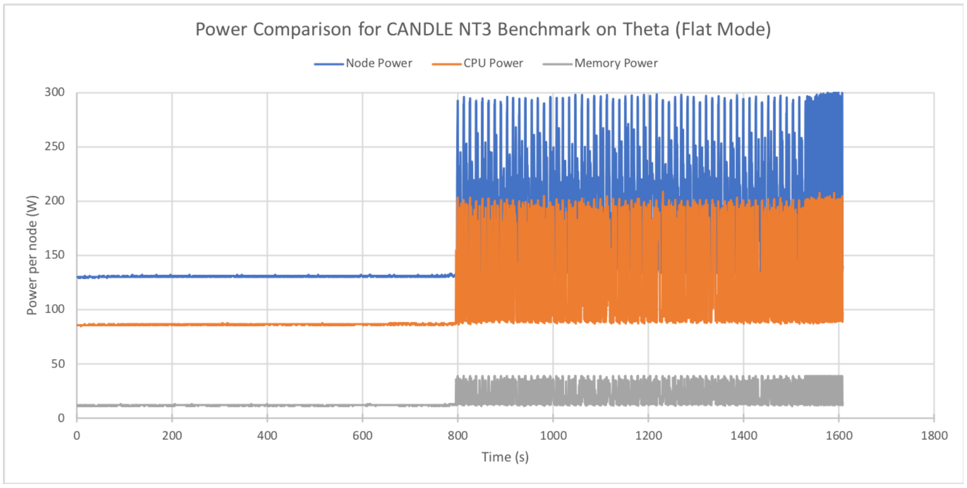


Figure 5. Power over time on Theta (with flat mode on one node and epochs = 1)

results consistent, finding that the difference in runtime is very small (less than 0.1%). Thus, we simply used the run with the smallest runtime to report the time and power. Increasing the batch size results in a decrease in the runtime and impacts the power slightly because it requires fewer iterations to converge to the same validation accuracy for models trained at larger batch sizes. The benchmark with a batch size of 200 achieves the lowest energy. We find that the benchmark fails, however, if the batch size is 300 or larger. Thus, the batch size can be adjusted only to some limited extent.

4.2. Horovod NT3 Benchmark

In this section, we still use one epoch and a batch size of 20 to conduct the experiments under the cache mode on Theta. However, we scale up the number of nodes with one epoch per node for our weak-scaling study. We measure the performance of the Horovod version of NT3 and use Python's cProfile [19] to profile the performance and analyze NT3's scalability on Theta.

Table 2 shows the time for different parts of the benchmark with increasing learning rate (0.001 * `hvd.size()`). The table headers are as follows: **TensorFlow**, the time for the method `_pywrap_tensorflow_internal.TF_Run()`; **Method read**, the time for the operations in `pandas._libs.parsers.TextReader`; **Keras callback**: the time for the Keras callbacks in `callbacks.py`; **Horovod callbacks**, the time for Horovod `callbacks.py`; **Distributed Optimizer**, the time for the Horovod Distributed Optimizer; **Broadcast**, the time for the Horovod broadcast itself; **Allreduce**, the time for Horovod allreduce; and **model.fit()**, the time spent in the model training and validation.

The Horovod distributed optimizer has small overhead, around 1.4s even with increasing numbers of nodes, because this optimizer delegates gradient computation to the

TABLE 2. Performance (in seconds) with the increased learning rate (weak scaling) on Theta.

#Nodes	2	4	8	16	32	64	128	256	512
Total Runtime	1561	1592	1604	1621	1683	1735	1769	1823	1916
TensorFlow	773	803	816	841	900	954	969	1020	1085
Method read	737	740	739	731	734	730	743	741	748
Keras callbacks	5.67	13.89	16.01	27.85	27.49	28.13	20.08	30.64	25.67
Horovod callbacks	5.66	13.88	16.00	27.84	27.48	28.13	20.08	30.62	25.60
Distributed Optimizer	1.40	1.42	1.47	1.39	1.37	1.47	1.44	1.36	1.43
Broadcast	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22
Allreduce	0.24	0.24	0.24	0.24	0.24	0.24	0.24	0.24	0.25
Model.fit	735	756	767	780	840	893	916	976	1028

TABLE 3. Performance (in seconds) with the same learning rate (weak scaling) on Theta.

#Nodes	2	4	8	16	32	64	128	256	512
Total Runtime	1560	1600	1610	1619	1691	1732	1785	1839	1922
TensorFlow	774	813	817	837	905	947	986	1041	1082
Method read	735	734	743	733	735	734	744	735	737
Keras callbacks	10.27	19.92	15.17	24.01	22.49	26.17	24.92	34.85	30.14
Horovod callbacks	10.26	19.91	15.16	24.01	22.47	26.16	24.91	34.83	30.08
Distributed Optimizer	1.37	1.36	1.47	1.37	1.41	1.39	1.38	1.44	1.38
Broadcast	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22
Allreduce	0.24	0.24	0.24	0.24	0.24	0.24	0.24	0.24	0.24
Model.fit	731	761	769	780	850	888	929	974	1020

original optimizer, averages gradients using allreduce, and then applies those averaged gradients. Horovod callbacks introduce some overhead, but it is relatively small. We note that the Horovod broadcast overhead is around 0.22s and the Horovod allreduce overhead is around 0.24s. The dominant parts are data loading (Method read) and TensorFlow. Model.fit is the main part of TensorFlow and takes most of the execution time of TensorFlow. Because the Python cProfile provides the time for TensorFlow only as a whole, we have to add the inline timing for the function model.fit to measure its runtime.

We further tested the communication overhead introduced by Horovod with the same learning rate (0.001) shown in Table 3 on Theta. Overall, the total runtime results are slightly larger than those shown in Table 2. The Horovod overheads for the callbacks, optimizer, broadcast, and allreduce are similar for both cases. Compared with the total runtime, the Horovod overhead is relatively small. We note that, from Tables 2 and 3, the time spent in the TensorFlow and model.fit increases significantly—by more than 39%—as the number of nodes increases from 2 to 512. Ideally, if the batch size and learning rate and other parameters are fixed, the model training time is expected to be the same. Although Horovod has the ability to scale up, it causes the large communication overhead within TensorFlow, however, this overhead is not reflected from the Horovod functions provided by the cProfile. In the following section, we further explain this overhead using the Horovod timeline [11].

5. Performance and Power Analysis of the Horovod NT3 Benchmark

In this section, we use the problem size from the original NT3 benchmark for our strong-scaling study, where we fix the number of epochs at 400, the learning rate at 0.001, and the batch size at 20. We conduct our experiments with different numbers of nodes under different memory modes to analyze the performance and power behavior of the Horovod NT3 benchmark. We focus on three metrics: the time spent in the `model.fit()`, loss, and accuracy.

5.1. Cache Mode

Table 4 shows the performance for the original dataset with epochs = 400 and the increased learning rate on Theta. In this table, **loss** is the training loss; **acc** is the training accuracy; **val_loss** is the validation loss; and **val_acc** is the validation accuracy. **Time per epoch** is the total time for `model.fit()` divided by the number of epochs executed. On each node `model.fit()` executes a number of epochs, which is 400 divided by the number of nodes used. Because we did not shard the training data, each epoch makes a complete pass through the data. Thus, epoch time apparently increases as node count increases, but the increase is totally due to communication overhead, allowing us to capture this cleanly.

On 400 nodes, `model.fit()` executes one epoch per node, the total runtime is 1,042s, and the time per epoch is 1,042s. On 100 nodes, `model.fit()` executes 4 epochs per node, it takes 3,593s, and the time per epoch is around

TABLE 4. TIME (IN SECONDS), LOSS, AND ACCURACY FOR THE ORIGINAL DATASET WITH THE INCREASED LEARNING RATE (STRONG SCALING) ON THETA UNDER THE CACHE MODE

#Nodes	400	200	100	50	25
Time (Model.fit)	1,042	1,927	3,593	6,976	13,001
Time per epoch	1,042	964	898	872	813
loss	0.6912	0.6843	0.6498	0.0097	0.0037
acc	0.5420	0.5804	0.6482	0.9991	1.0000
val_loss	0.6941	0.7124	0.6331	0.1206	0.1166
val_acc	0.5143	0.5143	0.7857	0.9786	0.9786

TABLE 5. TIME (IN SECONDS), LOSS, AND ACCURACY FOR THE ORIGINAL DATASET WITH THE SAME LEARNING RATE (STRONG SCALING) ON THETA.

#Nodes	400	200	100	50	25
Time (Model.fit)	1,053	2,229	3,709	7,015	13,308
Time per epoch	1,053	1,115	927	877	832
loss	0.6874	0.6806	0.6345	0.5948	0.3335
acc	0.6750	0.8054	0.8491	0.8696	0.9304
val_loss	0.6815	0.6789	0.6324	0.5960	0.3276
val_acc	0.6964	0.8036	0.8571	0.8786	0.9536

898s. On 25 nodes, `model.fit()` executes 16 epochs per node, it takes 13,001s, and the time per epoch is around 813s. With increasing the number of nodes from 25 to 400, the accuracy decreases and the loss increases because `model.fit()` executes fewer number of epochs. With the increased learning rate, using 25 nodes results in a training accuracy of 1.0 and validation accuracy of 0.9786. These results indicate that properly increasing the learning rate leads to better accuracy, and the model training requires the proper number of epochs (16) to achieve the high accuracy. We note that the time per epoch increases with increasing numbers of nodes because the Horovod communication overhead increases with the number of nodes.

Figure 6 shows power over time for the Horovod NT3 benchmark on 400 nodes. We found that loading the clib for the power measurement using ctypes takes around 11s based on the profile data from the Python cProfile. Therefore, the data loading takes around 781s. Compared with Figure 4, what happened after the data-loading phase? To explain the power behavior in Figure 5, we use the Horovod timeline feature [11] to record the communication activities viewed in the Chrome browser through chrome://tracing [7].

Figures 7 and 8 show the timeline for the communication of the benchmark on 400 nodes with the highlights of broadcast and allreduce from Figure 6. This timeline starts the broadcast communication, not the beginning of the benchmark. It consists of six communication types: negotiate_broadcast, broadcast, mpi_broadcast, allreduce, mpi_allreduce, and negotiate_allreduce, where broadcast is implemented based on mpi_broadcast shown in Figure 7; allreduce is based on the baidu ring-allreduce algorithm [3] and `MPI_Allreduce()` shown in Figure 8. MEMCPY_IN_FUSION_BUFFER and MEMCPY_OUT_FUSION_BUFFER are to copy data into and out of the fusion buffer. Each tensor broadcast/reduction in the Horovod NT3 benchmark involves two major phases.

- The **negotiation phase** (negotiate_broadcast, negotiate_allreduce): all workers send a signal to rank 0 that they are ready to broadcast/reduce the given tensor. Each worker is represented by a tick under the negotiate_broadcast/negotiate_allreduce bar. Immediately after negotiation, rank 0 sends a signal to the other workers to start broadcasting/reducing the tensor.
- The **processing phase**: here the communication operation actually happens. These communications in Figures 7 and 8 indicate the time taken to do the actual operation on the CPU and highlight that the operation was performed by using pure MPI collectives such as `MPI_Broadcast()` and `MPI_Allreduce()`.

Based on the communication activities in Figures 7 and 8, we are able to explain the power behavior in Figure 6. After data loading and preprocessing, the negotiate_broadcast takes place as shown in Figure 7. During the broadcast, the node power and CPU power decrease because of the dynamic power management on the Cray XC40. Then the gradients are computed, so the node power and CPU power increase. Both allreduce and `MPI_Allreduce()` are used to average the gradients, and the averaged gradients are applied. This process takes around 131s as the first allreduce and `MPI_Allreduce()` shown in Figure 8. The process takes place between 800s and 1000s in Figure 6. The model training batch steps then are started. During the training, negotiate_allreduce, allreduce, and `MPI_Allreduce()` take place between two consecutive training batch steps periodically in Figure 8. Compared with Figure 4, this Horovod overhead enlarges the gaps between two consecutive training batch steps, and the power consumption is smaller than that in Figure 4 because the internode communication is used for gradient averaging. This explains the communication overhead caused by Horovod within the TensorFlow run. The learning rate is 0.001 in Figure 4 and the increased learning rate is $0.001 \times 400 = 0.4$ in Figure 6, while keeping other input parameter values the same.

Table 5 shows the performance for the original dataset with epochs = 400 with the same learning rate on Theta. Compared with Table 4, the execution time increases slightly. For 100 nodes or more, the accuracy using the same learning rate is much higher than that using the increased learning rate. We note, however, that for 50 and 25 nodes, the accuracy using the same learning rate is lower than that using the increased learning rate. With increasing the number of nodes from 25 to 400, the accuracy decreases and the loss increases because `model.fit()` executes fewer number of epochs. This indicates that properly increasing the learning rate results in better accuracy, and the model training requires the proper number of epochs to achieve the high accuracy. Notice that the time per epoch increases with increasing the number of nodes because of the increased

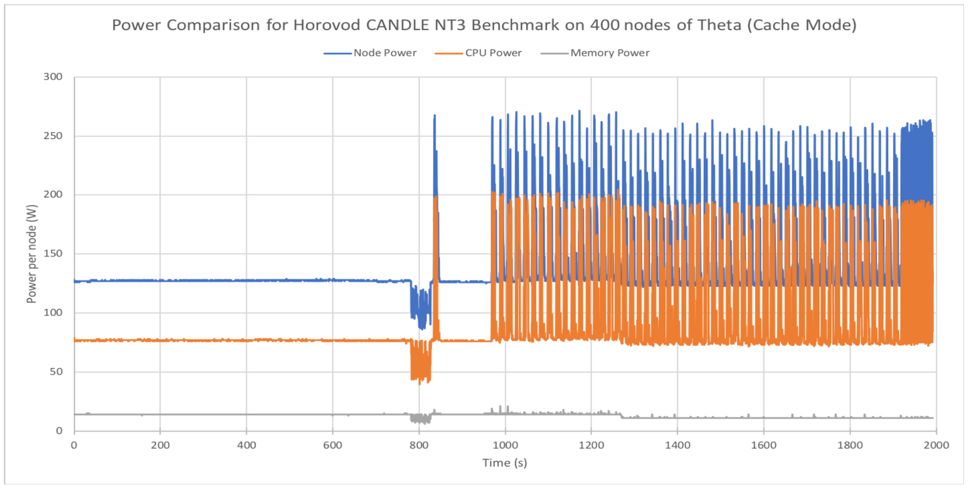
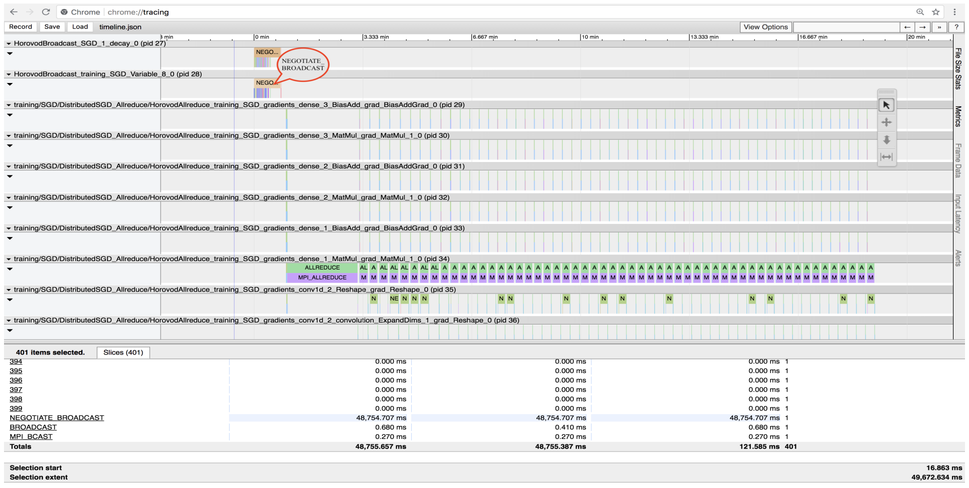


Figure 6. Power over time of the Horovod NT3 with the increased learning rate under the cache mode on 400 nodes.



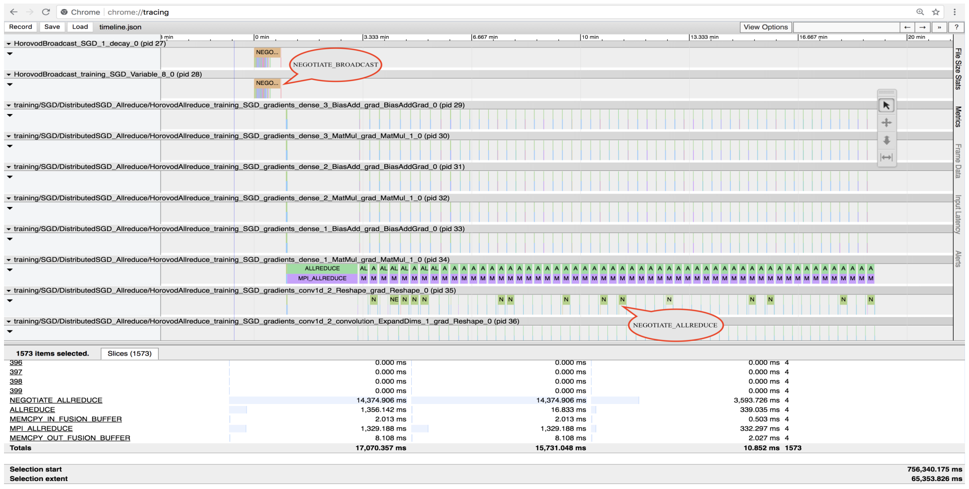
HorovodBroadcast_SGD_1_decay_0 (pid 27)
HorovodBroadcast_training_SGD_Variable_8_0 (pid 28)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_dense_3_BiasAdd_grad_BiasAddGrad_0 (pid 29)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_dense_3_MatMul_grad_MatMul_1_0 (pid 30)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_dense_2_BiasAdd_grad_BiasAddGrad_0 (pid 31)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_dense_2_MatMul_grad_MatMul_1_0 (pid 32)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_dense_1_BiasAdd_grad_BiasAddGrad_0 (pid 33)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_dense_1_MatMul_grad_MatMul_1_0 (pid 34)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_conv1d_2_Reshape_grad_Reshape_0 (pid 35)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_conv1d_2_convolution_ExpandDims_1_grad_Reshape_0 (pid 36)

401 items selected. Slices (401)

394	0.000 ms	0.000 ms	0.000 ms	1
395	0.000 ms	0.000 ms	0.000 ms	1
396	0.000 ms	0.000 ms	0.000 ms	1
397	0.000 ms	0.000 ms	0.000 ms	1
398	0.000 ms	0.000 ms	0.000 ms	1
399	0.000 ms	0.000 ms	0.000 ms	1
NEGOTIATE_BROADCAST	48,754.707 ms	48,754.707 ms	48,754.707 ms	1
BROADCAST	0.680 ms	0.410 ms	0.680 ms	1
MPI_BCAST	0.270 ms	0.270 ms	0.270 ms	1
Totals	**48,755.657 ms**	**48,755.387 ms**	**121.585 ms**	**401**

Selection start	**16.863 ms**
Selection extent	**49,672.634 ms**

Figure 7. Timeline for the communication with the highlight of broadcast of the Horovod NT3 on 400 nodes (cache mode).



HorovodBroadcast_SGD_1_decay_0 (pid 27)
HorovodBroadcast_training_SGD_Variable_8_0 (pid 28)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_dense_3_BiasAdd_grad_BiasAddGrad_0 (pid 29)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_dense_3_MatMul_grad_MatMul_1_0 (pid 30)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_dense_2_BiasAdd_grad_BiasAddGrad_0 (pid 31)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_dense_2_MatMul_grad_MatMul_1_0 (pid 32)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_dense_1_BiasAdd_grad_BiasAddGrad_0 (pid 33)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_dense_1_MatMul_grad_MatMul_1_0 (pid 34)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_conv1d_2_Reshape_grad_Reshape_0 (pid 35)
training/SGD/DistributedSGD_Allreduce/HorovodAllreduce_training_SGD_gradients_conv1d_2_convolution_ExpandDims_1_grad_Reshape_0 (pid 36)

1573 items selected. Slices (1573)

396	0.000 ms	0.000 ms	0.000 ms	4
397	0.000 ms	0.000 ms	0.000 ms	4
398	0.000 ms	0.000 ms	0.000 ms	4
399	0.000 ms	0.000 ms	0.000 ms	4
NEGOTIATE_ALLREDUCE	14,374.906 ms	14,374.906 ms	3,593.726 ms	4
ALLREDUCE	1,356.142 ms	16.833 ms	339.035 ms	4
MEMCPY_IN_FUSION_BUFFER	2.013 ms	2.013 ms	0.503 ms	4
MPI_ALLREDUCE	1,329.188 ms	1,329.188 ms	332.297 ms	4
MEMCPY_OUT_FUSION_BUFFER	8.108 ms	8.108 ms	2.027 ms	4
Totals	**17,070.357 ms**	**15,731.048 ms**	**10.852 ms**	**1573**

Selection start	**756,340.175 ms**
Selection extent	**65,353.826 ms**

Figure 8. Timeline for the communication with the highlight of allreduce of the Horovod NT3 on 400 nodes (cache mode).

TABLE 6. TIME (IN SECONDS), LOSS, AND ACCURACY FOR THE ORIGINAL DATASET WITH THE INCREASED LEARNING RATE (STRONG SCALING) ON THETA.

#Nodes	400	200	100	50	25
Time (Model.fit)	1,188	2,089	4,062	7,860	15,436
Time per epoch	1,188	1,045	1,016	983	965
loss	0.7071	0.6783	0.6755	0.0111	0.0057
acc	0.5464	0.5991	0.5920	0.9991	0.9991
val_loss	0.6910	0.6841	0.6718	0.1241	0.1210
val_acc	0.6714	0.8821	0.8393	0.9714	0.9786

Horovod communication overhead.

5.2. Flat Mode

Table 6 shows the performance for the original dataset with epochs = 400 with the increased learning rate using the flat mode on Theta. The results also indicate that, compared with Table 4, properly increasing the learning rate results in better accuracy, and the model training requires a sufficient number of epochs (8) to achieve the high accuracy. When we compare this with Table 4 using the cache mode, we see that the benchmark benefits more from using the cache mode than the flat mode. Notice that the time per epoch also increases with increasing the number of nodes because of the increased Horovod overhead.

Figure 9 shows power behavior similar to that in Figure 6. For the benchmark, using the cache mode results in smaller runtime and lower power consumption for the node and CPU. Compared with Figure 5, the Horovod overhead enlarges the gaps between two consecutive training batch steps because of the allreduce operations as discussed in the preceding section, and similarly the power consumption is smaller than that in Figure 5.

6. Conclusions

In this paper, we used Horovod to parallelize the Python NT3 benchmark from the exploratory research project CANDLE. We then analyzed the performance, power, and scalability of the Horovod implementation with weak scaling and strong scaling on the Cray XC40 Theta with different batch sizes and learning rates under two memory modes: cache and flat on Theta. Our experimental results indicate that power profiling for the node, CPU, and memory is useful for showing how the Horovod NT3 benchmark behaves on the underlying system. The communication timeline of this benchmark showed that the Horovod communication overhead for the benchmark increased significantly with the number of nodes although Horovod has the ability to scale up. The benchmark under the cache mode resulted in smaller runtime and lower power consumption for the node and CPU as compared with results under the flat mode. Increasing the batch size led to a runtime decrease and slightly impacted the power; and increasing the learning rate resulted in a slight decrease in runtime and node power and an increase in accuracy, and the model training in NT3 requires the proper number of epochs to achieve the high accuracy.

We plan to address several issues raised by the NT3 results. First, the data-loading time becomes the bottleneck after speeding the model-training process. We have to consider how to speed the input dataset operations, perhaps by partitioning them. Second, we plan to perform additional measurements with sharded and shuffled data in Horovod; these will improve the loss and accuracy resulting from our training runs, however, they do not affect the systems-related results presented here. Third, the performance profiling using the Python cProfile includes only the total time for the whole TensorFlow run, as shown in Tables 2 and 3. It does not give any details about how TensorFlow behaves. To further improve the performance of the TensorFlow run, we may need a fine-grained performance profiling tool such as NVProf [17] to profile the TensorFlow run. Further, we developed the Horovod version of the NT3 benchmark to support both CPUs and GPUs. We plan to test the benchmark on heterogeneous systems with CPUs and GPUs, such as Summit [21]. We also plan to add checkpoint/restart features to the Horovod benchmark for fault tolerance. Lastly, we plan to use our performance and power modeling work [26] to model and optimize the CANDLE benchmarks. Python codes, like other scripting languages, do not have compiler optimization support and instead rely on the library, resource, and environment settings for improve performance. We can utilize our previous work to identify better resource and environment settings for performance improvement.

Because the NT3 benchmark, like other CANDLE benchmarks, is implemented in Python by using the Keras framework, the parallelization method using Horovod in this paper can be applied to other TensorFlow-based CANDLE benchmarks such as the Pilot1 and Pilot3 benchmarks in a similar way. We plan to utilize OpenMP parallelism to further develop hybrid Horovod/OpenMP/GPU versions of CANDLE benchmarks.

Acknowledgments

This work was supported in part by Laboratory Directed Research and Development (LDRD) funding from Argonne National Laboratory, provided by the Director, Office of Science, of the U.S. Department of Energy under contract DE-AC02-06CH11357, and in part by NSF grant CCF-1801856. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. We would like to acknowledge the Argonne Leadership Computing Facility for use of the Cray XC40 Theta under the DOE INCITE project PEACES and ALCF project EE-ECP, and Gail W. Pieper and Emily Dietrich for their edits of this paper.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A.

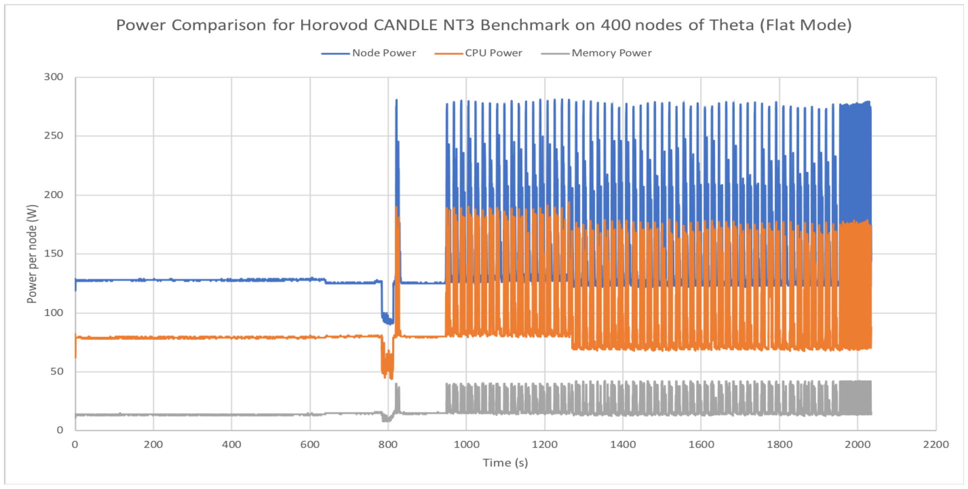


Figure 9. Power over time of the Horovod NT3 with the increased learning rate under the flat mode on 400 nodes.

Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous distributed systems, arXiv:1603.04467, 2016.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, TensorFlow: A system for large-scale machine learning, arXiv:1605.08695, 2016.

[3] Baidu-allreduce, https://github.com/baidu-research/baidu-allreduce, https://github.com/baidu-research/tensorflow-allreduce.

[4] CANDLE: Cancer Distributed Learning Environment, http://candle.cels.anl.gov.

[5] CANDLE Benchmarks: https://github.com/ECP-CANDLE/Benchmarks.

[6] CANDLE NT3 Benchmark, https://github.com/ECP-CANDLE/Benchmarks/blob/frameworks/Pilot1/NT3, https://github.com/ECP-CANDLE/Benchmarks/tree/master/Pilot1/NT3.

[7] Chrome trace event profiling tool, https://www.chromium.org/developers/how-tos/trace-event-profiling-tool.

[8] Cray, Monitoring and Managing Power Consumption on the Cray XC System, Tech Report, S-0043-7204.

[9] Cray XC40 Theta, Argonne National Laboratory, https://www.alcf.anl.gov/theta.

[10] Distributed TensorFlow, https://www.tensorflow.org/deploy/distributed, https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/distributed_runtime/README.md.

[11] Horovod: A Distributed Training Framework for TensorFlow, https://github.com/uber/horovod.

[12] Keras: The Python Deep Learning Library, https://keras.io/#keras-the-python-deep-learning-library.

[13] I. Marincic, V. Vishwanath, and H. Hoffmann, PoLiMEr: An Energy Monitoring and Power Limiting Interface for HPC Applications, SC2017 Workshop on Energy Efficient Supercomputing, Nov. 13, 2017.

[14] S. Martin, D. Rush, M. Kappel, M. Sandstedt, and J. Williams. 2016. Cray XC40 Power Monitoring and Control for Knights Landing. Proceedings of the Cray User Group (CUG), 2016.

[15] P. Mendygral, Scaling Deep Learning, ALCF SDL(Simulation, Data and Learning) Workshop, March 2018.

[16] Microsoft Cognitive Toolkit, https://github.com/Microsoft/cntk.

[17] NVProf, https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[18] Pandas, https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html.

[19] Python Profilers, https://docs.python.org/2/library/profile.html.

[20] A. Sergeev and M. Del Balso, Horovod: Fast and Easy Distributed Deep Learning in TensorFlow, arXiv:1802.05799v3, Feb. 21, 2018.

[21] Summit, https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

[22] TensorFlow, https://www.tensorflow.org.

[23] TensorFlow Benchmarks, https://www.tensorflow.org/performance/benchmarks.

[24] Theano, https://github.com/Theano/Theano.

[25] J. M. Wozniak, R. Jain, P. Balaprakash, J. Ozik, N. Collier, J. Bauer, F. Xia, T. Brettin, R. Stevens, J. Mohd-Yusof, C. G. Cardona, B. Van Essen, and M. Baughman, CANDLE/Supervisor: A Workflow Framework for Machine Learning Applied to Cancer Research, SC17 Workshop on Computational Approaches for Cancer Workshop, November 2017.

[26] X. Wu, V. Taylor, J. Cook, and P. Mucci, Using Performance-Power Modeling to Improve Energy Efficiency of HPC Applications, IEEE Computer, Vol. 49, No. 10, pp. 20-29, Oct. 2016.