Performance Modeling and Evaluation of Distributed Deep Learning Frameworks on GPUs

Shaohuai Shi, Xiaowen Chu
Department of Computer Science, Hong Kong Baptist University
{csshshi, chxw}@comp.hkbu.edu.hk

***Abstract*—Deep learning frameworks have been widely deployed on GPU servers for deep learning applications in both academia and industry. In the training of deep neural networks (DNNs), there are many standard processes or algorithms, such as convolution and stochastic gradient descent (SGD), but the running performance of different frameworks might be different even running the same deep model on the same GPU hardware. In this paper, we evaluate the running performance of four state-of-the-art distributed deep learning frameworks (i.e., Caffe-MPI, CNTK, MXNet and TensorFlow) over single-GPU, multi-GPU and multi-node environments. We first build performance models of standard processes in training DNNs with SGD, and then we benchmark the running performance of these frameworks with three popular convolutional neural networks (i.e., AlexNet, GoogleNet and ResNet-50), after that we analyze what factors that results in the performance gap among these four frameworks. Through both analytical and experimental analysis, we identify bottlenecks and overheads which could be further optimized. The main contribution is two-fold. First, the testing results provide a reference for end users to choose the proper framework for their own scenarios. Second, the proposed performance models and the detailed analysis provide further optimization directions in both algorithmic design and system configuration.**

***Index Terms*—Deep Learning; GPU; Distributed SGD; Convolutional Neural Networks; Deep Learning Frameworks**

I. Introduction

In recent years, deep learning (DL) techniques have achieved great success in many AI applications [1]. With large amount of data, deep neural networks (DNNs) can learn the feature representation very well. Very deep neural networks and large scale of data, however, result in a high requirement of computation resources. Fortunately, on one hand, some hardware accelerators including GPUs, FPGA and Intel Xeon Phi processors can be used to reduce the time of model training [2][3]. On the other hand, it has been recently proven that DNNs with very large size of mini-batch can converge well to a local minimal [4][5][6], which is significant to utilize many processors or clusters efficiently. A single accelerator has limited computational resources (e.g., computation units and memory) to process large-scale neural networks, so parallel training algorithms are proposed to solve this problem such as model parallelization [7] and data parallelization [8][9][10]. This also triggers the technology giants deploy their cloud services with highly scalable deep learning tools. Amazon adopts MXNet [11] as the main deep learning framework for cloud service AWS, Google uses TensorFlow [12] for Google cloud, and Microsoft develops Microsoft Cognitive Toolkit (CNTK) [13] for Microsoft Azure. In addition, Inspur also develops Caffe-MPI[1] to support the distributed deployment of HPCs.

CNTK, MXNet and TensorFlow have achieved not only high throughout in a single GPU with the help of cuDNN [14] which is a high performance DNN library provided by NVIDIA, but they also have good scalability across multiple GPUs and multiple machines. These frameworks provide an easy way to users to develop DNNs, and try to optimize related algorithms to achieve high throughput by using hardware platforms like multi-core CPU, many-core GPU, multiple GPUs and multiple machines. However, because of the different implementation methods by vendors, these tools show different performance even when training the same DNNs on the same hardware platform. Researchers have evaluated different tools on various hardware with diverse DNNs [15][16][17][18], but the upgrade of frameworks and GPU is quite frequent so that these benchmarks could not reflect the latest performance of newer GPUs and softwares. In addition, the scalability in multi-GPU and multi-machine platforms is not well studied, which is one of the most important factors in clusters. In this paper, we extend our previous work [16] to evaluate the performance of four distributed DL frameworks (i.e., Caffe-MPI, CNTK, MXNet and TensorFlow) with convolutional neural networks (CNNs) over the GPU cluster. We use four machines connected by a 56Gb InfiniBand network, each of which is equipped with four NVIDIA Tesla P40 cards, to test the training speed of each framework in CNNs covering single-GPU, multi-GPU and multi-machine environments[2]. We first test the running performance of SGD optimization, and then focus on the performance of synchronous SGD (S-SGD) across multiple GPUs/machines to analyze the performance details. Our major findings are summarized as follows[3]:

1) For relatively shallow CNNs (e.g., AlexNet), loading large amounts of training data could become a potential bottleneck with large mini-batch size and fast GPUs. Efficient data pre-processing can be used to reduce the impact.

[1]https://github.com/Caffe-MPI/Caffe-MPI.github.io

[2]Our source code and experimental data can be downloaded from http://www.comp.hkbu.edu.hk/~chxw/dlbench.html.

[3]The software tools are being upgraded frequently. The findings are based on our own experimental platforms, software configurations and only apply to the software versions specified in the paper.

2) To better utilize cuDNN, autotune and input data layout (e.g., NCWH, NWHC) should be considered. Both CNTK and MXNet expose the autotune configuration of cuDNN, which could achieve better performance during forward and backward propagation.
3) In S-SGD with multiple GPUs, CNTK does not hide the overhead of gradient communication, while MXNet and TensorFlow parallelize the gradient aggregation of the current layer with the gradient computation of the previous layer. By hiding the overhead of gradient communication, the scaling performance could be better.
4) All the frameworks scale not so well across four high throughput dense GPU servers. The inter-node gradient communication via 56Gbps network interface is much slower than the intra-node via PCIe.

The rest of the paper is organized as follows. Section II introduces the related work. Section III presents preliminaries of SGD and S-SGD implemented by different approaches. We derive some performance models for different implementations of S-SGD in Section IV. Our experimental methodology is introduced in Section V, followed by the experimental results and our analysis in Section VI. We conclude the paper and discuss our future work in Section VII.

II. Background and Related Work

Stochastic gradient descent (SGD) methods are the most widely used optimizer in deep learning communities because of its good generalization and easy computation with first order gradient [19][20], and it can scale to multiple GPUs or machines for larger dataset and deeper neural networks. Distributed SGD methods have achieved very good scaling performance [21][10][4][5][6][22], and the existing popular DL frameworks have the built-in components to support scaling by using some configurations or APIs, among which CNTK, MXNet and TensorFlow are examples of the most active and popular ones. However, these frameworks implement the working flow of SGD in different ways, which results in some performance gap even though they all make use of high performance library cuDNN [14] to accelerate the training on GPUs. In addition, the implementation of S-SGD may vary so much for different purposes.

Parameter server (PS) based methods [23][24] for distributed machine learning algorithms have been widely used in many distributed SGD algorithms like asynchronous SGD [25] and S-SGD. Several performance models based on PS methods have been proposed by S. Zou et al. [26], and they develop the procedure to help users better choose the mini-batch size and the number of parameter servers.

A. Awan et al. [27][28] propose the high performance CUDA-Aware MPI to alleviate the overhead of data communication so that they can scale the distributed learning better on GPU clusters. In the recent research, P. Goyal et al. [6] use a dense GPU cluster with 256 GPUs to achieve about 90% efficiency. Except the PS-based method used in [6], the optimized all-reduce implementation and pipelining all-reduce operations with gradient computation make training nearly perfect linear scale up possible in ResNet-50 [29]. Most of these researches focus on the optimization of PS-based methods which have very high requirement on the network bandwidth between the parameter server and workers, while the decentralized methods are less studied since they are initially considered to highly rely on the PCIe topology between GPU and CPU. X. Lian et al. [30] come up with a decentralized S-SGD algorithm which has a theoretical guarantee of convergence to overcome the communication bottleneck across the dense GPU cluster. Even though this work only conducts experiments on small size of dataset, it lets us re-consider the importance of decentralized S-SGD algorithms on GPU clusters. And the hybrid method of PS-based and decentralized is also proposed to speed up training [31]. It is noted that both PS-based methods and the decentralized S-SGD have been integrated into most distributed DL frameworks.

Bahrampour et al. [15] and Shi et al. [16] have evaluated the performance of some state-of-the-art DL frameworks on the single-GPU environment. But both frameworks and hardware are upgraded so frequently, hence new results are needed to be further analyzed. In the distributed environment, Shams et al. [17] have studied the performance of NVIDIA's NVLink and Intel's Knights Landing on different CPU and GPU technologies. However, the evaluated TensorFlow is at version v0.12, while Google has upgraded TensorFlow to v1.0+ for performance improvement, and the other two popular commercial frameworks (CNTK and MXNet) are not compared in [17]. In this paper, we first compare the performance of Caffe-MPI, CNTK, MXNet and TensorFlow via single-GPU, multi-GPU and multi-node environments through analysis and experimental results, and then identify the performance gap among these four frameworks.

III. Preliminaries

In this section, we first introduce the basic workflow of SGD and the naive implementation of the S-SGD algorithm, which may result in suboptimal efficiency due to the bottleneck of data communication. Then two optimized strategies of S-SGD are introduced, followed by a further optimization approach to reducing the impact of data communication. The mathematical notations in our analysis are summarized in Table I. We assume that each node in the cluster has the same hardware configuration.

A. Mini-batch SGD

To train the model with the mini-batch SGD, one should update the model iteratively with feeding data. An iteration is also called a mini-batch, which has a part of instances of the total dataset. It generally contains five steps in an iteration. 1) Read a mini-batch of data from disk to memory. 2) Transfer the data from the CPU memory to the GPU memory. 3) GPU kernels are launched to do feed forward operation layer by layer. 4) To minimize the objective function, first order gradients w.r.t weights and inputs are calculated with chain rule, which is the phase of backward propagation. 5) The

TABLE I
SUMMARY OF NOTATIONS

Name	Description
N	# of nodes or machines
N_g	# of total GPUs
n_g	# of GPUs on each node
B	Network throughput between two nodes
$t_{latency}$	Latency of the network
B_{pcie}	Bandwidth of PCIe on a single node
B_{disk}	Bandwidth of data I/O from the disk
B_{cache}	Bandwidth of reading data from the memory cache
M	The number of training samples per GPU in a mini-batch
D	Input data size fed for each GPU in a mini-batch
t_{iter}	Time of an iteration
t_{io}	Time of I/O in each iteration
t_{h2d}	Time of data transfer from CPU to GPU in each iteration
t_f	Time of forward phase in each iteration
t_b	Time of backward propagation in each iteration
$t_b^{(i)}$	Time of backward propagation of layer i in each iteration
t_u	Time of model update in each iteration
t_{comm}	Time of gradients aggregation in each iteration
$t_{comm}^{(i)}$	Time of gradients aggregation of layer i in each iteration
L	The number of learnable layers of DNN
C	The assumption of the C^{th} learnable layer, $t_{comm}^{(i)} \leq t_b^{(i-1)}$ for $i = 2, 3, ..., C-1$, and $t_{comm}^{(i)} > t_b^{(i-1)}$ for $i = C, C+1, ..., L$

model is updated based on the gradients. So the total time of one iteration can be calculated as:

$$t_{iter} = t_{io} + t_{h2d} + t_f + t_b + t_u. \quad (1)$$

To achieve higher efficiency of training, step 1) is often processed with multiple threads so that the I/O time of a new iteration can be overlapped with the computing time of the previous iteration. In other words, in every new iteration, the data can be accessed from the CPU memory directly without waiting for data from the disk if the data has been ready during the computation of the previous iteration.

B. Naive S-SGD

In general, S-SGD makes each worker perform feed forward and backward propagation on the same mini-batch of M samples independently with duplicate copy of model. Before updating the model, the gradients are aggregated. The pseudo-code of S-SGD algorithm is shown in Algorithm 1 [8]. However, different DL platforms choose different optimization methods and results in different performance.

Algorithm 1 S-SGD

1: **procedure** S-SGD((parameters, data, N))
2: **for** *each worker* $i \in \{1, 2, ..., N\}$ **do**
3: $\nabla g_i \leftarrow SGD(parameters, \frac{data}{N})$
4: *Aggregate from all workers:* $\nabla g \leftarrow \frac{1}{N} \sum_{i=1}^{N} \nabla g_i$
5: **Return** ∇g

There are five steps to implement the naive S-SGD algorithm with a distributed cluster. 1) NMn_g samples are read and/or preprocessed to get ready for training, and each machine reads and/or preprocesses Mn_g samples. 2) In each machine, Mn_g samples are evenly distributed to n_g different GPUs through PCIe or NVLink. 3) each GPU launches kernels to do feed forward and backward propagation operations in parallel. 4) Parameters from all GPUs are aggregated in one node, and it can be done by CPU or GPU. 5) Averaged parameters are sent back to each GPU, and each GPU updates its own parameters. In step 4), the aggregation operation should not be started before the parameters from all GPUs are received, which indicates the meaning of synchronous SGD.

There are two main drawbacks in the above implementation. First, step 4) could easily become the bottleneck of performance since it requires a very high bandwidth of network to receive the parameters from all GPUs. Second, gradients computation and aggregation are in a sequential way, which results in low efficient usage of computational resources.

C. PS Method

PS-based method is one of the state-of-the-art methods to solve the first problem above. It is a centralized topology. There is a parameter server (PS) to store the whole model in a single node, and it can be extended to two or more PSes if needed. PS aggregates parameters at each iteration and updates the model and then pushes the updated model to each GPU. As a centralized node, it also suffers from the high pressure if the number of parameters is huge. In such case, more than one PS nodes can be deployed. Each PS only receives a subset of GPUs to accumulate, and inter-PS aggregates the model.

D. Decentralized Method

Decentralized method is another algorithm to reduce the impact in step 4) of naive S-SGD. The parameters are exchanged between GPUs, and it is generally implemented by the high performance all-reduce or all-gather operations. The all-reduce collective can accumulate the parameters from all GPUs. Gloo[4] and NCCL[5] are two efficient implementations of all-reduce.

E. Pipeline of layer-wise gradient aggregation and backward propagation

Both PS and decentralized methods are efficient to address the gradient communication problem. However, initial proposals are also suboptimal since there does exist overhead of gradient aggregation. It is noted that for each layer of CNNs, there is no dependency between gradient computation and update. This property makes it possible to hide the overhead of gradient communication via pipelining the layer-wise gradient aggregation with backward propagation.

IV. PERFORMANCE MODELING

In this section, we build the performance models of training DNNs with SGD (or S-SGD) in Caffe-MPI, CNTK, MXNet and TensorFlow. From the workflow described in Sections

[4]https://github.com/facebookincubator/gloo
[5]https://developer.nvidia.com/nccl

III-A and III-B, it is straightforward to represent the iteration time t_{iter} with:

$$t_{iter} = t_{io} + t_{h2d} + t_f + t_b + t_u + t_{comm}. \quad (2)$$

Let $t_{gpu} = t_{h2d} + t_f + t_b + t_u$, then we have

$$t_{iter} = t_{io} + t_{gpu} + t_{comm}. \quad (3)$$

In the single-GPU environment, $t_{comm} = 0$. Some terms in Equation 3 can be eliminated by pipeline techniques. Here we will discuss four optimization techniques to achieve better performance improvement.

A. I/O hidden

First, I/O could be very slow because of its low bandwidth, for example, $B_{disk} = 750$MB/s and $B_{cache} = 3.5$GB/s in our tested servers, and $t_{io} = \frac{Dn_g}{B_{cache}}$. So t_{io} should be hidden to achieve better utilization of GPUs. Pipelining the I/O with the GPU computation is a useful way to hide the overhead of I/O. In other words, there are multiple CPU threads preparing the data for GPU computation. So we can calculate the average iteration time of pipelined SGD as:

$$\bar{t}_{iter} = max\{t_{gpu} + t_{comm}, t_{io}\}. \quad (4)$$

B. Efficient communication via NCCL

Regarding S-SGD, gradients communication across multiple GPUs or multiple servers could also easily become the bottleneck of performance due to the transferring overhead via low-bandwidth or high-latency devices (i.e., PCI-e and network interface). NCCL is a high-performance library to exchange data across multiple GPUs both in the single node and the distributed environment. Therefore, it is somehow effective to reduce the value of t_{comm} by using NCCL.

C. Communication hidden

A main property of mini-batch SGD training of CNNs is that the gradient computation has no dependency with the updating of their next layers, so the gradient computation in layer l_{i-1} can be parallelized with the gradient aggregation, which helps reduce the overhead of data communication. Let τ_{comm_s} and τ_{comm_e} denote the start and the end time points of communication during one iteration respectively, so the iteration time is:

$$t_{iter} = t_{io} + t_{h2d} + t_f + t_b^{(L)} + \tau_{comm\_e} - \tau_{comm\_s} + t_u, \quad (5)$$

where $t_b^{(L)}$ is the gradient computation time of the last learnable layer. In the PS method, the updating of gradients can be calculated in the parameter server, and its cost is much smaller than gradient computation, so that it can be hidden by pipelining. We discuss two cases:

- **Case 1.** The gradient communication is totally hidden by the backward propagation. I.e., $t_{comm}^{(i)} \leq t_b^{(i-1)}, 2 \leq i \leq L$.
- **Case 2.** There exist some layers whose communication time are longer than the time of backward computation of the previous layers. We formulate this case with: $t_{comm}^{(i)} \leq t_b^{(i-1)}$ for $i = 2, ..., C-1$, and $t_{comm}^{(i)} > t_b^{(i-1)}$ for $i = C, C+1, ..., L$.

For **Case 1**, the communication time is hidden by computation, we can update the representation of t_{iter} by:

$$t_{iter} = t_{io} + t_{h2d} + t_f + t_b + t_{comm}^{(1)} + t_u^{(1)}, \quad (6)$$

For **Case 2**, there exist some layers whose communication overheads are not hidden, so t_{iter} is estimated by:

$$t_{iter} = t_{io} + t_{h2d} + t_f + \sum_{i=C}^{L} t_{comm}^{(i)} + \sum_{i=1}^{C-1} t_b^{(i)} + t_{comm}^{(1)} + t_u^{(1)}, \quad (7)$$

where L is the number of learnable layers of DNN. It is noted the larger C, the more communication can be hidden. By using efficient communication library like NCCL, t_{comm} could be somehow lower values, which could make the value of C larger.

D. Merged-layer communication hidden

Considering the overhead of network communication, there are two overheads, i.e., latency and transmission time. If exchanging large number of small data packets, the latency could take over a not neglectable overhead of total communication time. Therefore, gradients from different c layers can be merged into a big data packet, and it just needs to transmit once, which reduces the impact of latency. The performance rely on the network environment, neural network models and the the layer merge strategy. We leave this for future work.

Let $t_{iter\_N_g}$ and $t_{io\_n_g}$ denote the iteration time and the I/O time of a mini-batch with N_g GPUs across N machines (each machine has n_g GPUs) respectively. For example, t_{iter_1} and t_{io_1} are the iteration time and the I/O time respectively on the single-GPU environment. The speedup can be formulated by:

$$\begin{aligned} S &= \frac{MN_g/t_{iter\_N_g}}{M/t_{iter\_1}} = N_g \frac{t_{iter\_1}}{t_{iter\_N_g}} \\ &= N_g \frac{t_{io\_1} + t_{gpu}}{t_{io\_n_g} + t_{gpu} + t_{comm}}. \end{aligned} \quad (8)$$

So the overheads of I/O and communication are two key factors to influence the scalability of the system. On one hand, in terms of I/O, the large mini-batch size may result in $t_{io\_n_g} \approx n_g t_{io\_1}$, so if I/O becomes the bottleneck, the system can only achieve limited speedup. Regarding data communication, on the other hand, the higher overhead of t_{comm}, the lower S. In summary, the more I/O and data communication hidden, the higher speedup can be achieved.

Caffe-MPI. In Caffe-MPI, there are some optimized techniques to improvement the training performance. First, multiple threads are used to read data in parallel. Second, the efficient all-reduce library NCCL is used for the communication of multi-GPU and multi-node. Furthermore, the backward propagation is pipelined with the gradient aggregation in some time.

CNTK. In CNTK, multiple threads are generated by OpenMP for parallelized data reading. In terms of t_{comm},

CNTK exploits the optimized all-reduce library NCCL to cut down the overhead of data communication when aggregating the gradients in S-SGD. And the S-SGD algorithm implemented in CNTK is a decentralized architecture. However, the backward propagation is not pipelined with the gradient aggregation, so the iteration time of CNTK can be represented by Equation 4, and the speedup can be estimated by Equation 8.

MXNet. Instead of using decentralized communication mechanism, MXNet exploits PS methods and allows GPU to be used as the PS. In addition, the technique of pipeline between gradients aggregation and backward propagation is also integrated in MXNet, which means the overhead of gradient aggregation could be hidden by the backward propagation. The performance model of MXNet can be expressed by Equation 6 for the case that the overhead of gradient communication can be hidden or Equation 7 when $t_{comm}^{(i)} > t_b^{(i-1)}$, and the speedup can be estimated by:

$$S = \begin{cases} \frac{N_g(t_{io\_1}+t_{gpu})}{t_{io\_n_g}+t_{h2d}+t_f+t_b+t_{comm}^{(1)}+t_u^{(1)}} & \textbf{Case 1} \\ \frac{N_g(t_{io\_1}+t_{gpu})}{t_{io\_n_g}+t_{h2d}+t_f+\sum_{i=C}^{L} t_{comm}^{(i)}+\sum_{i=1}^{C-1} t_b^{(i)}+t_{comm}^{(1)}+t_u^{(1)}} & \textbf{Case 2} \end{cases} \quad (9)$$

TensorFlow. It supports multiple modes of parallelism for distributed training including the PS method, decentralized all-reduce and the mixture of PS and all-reduce. To show the performance gap among different implementations, we use the hybrid method of TensorFlow to test its scalability. The hybrid method has two differences with CNTK and MXNet. First, in the single node environment, the gradient aggregation is finished by collectives, which is similar with CNTK, while it further exploits the pipeline to overlap the gradient communication and backward propagation, but TensorFlow updates the model at the end of all gradient aggregation, which is different from MXNet. Therefore, in the single-node environment, we have:

$$t_{iter} = t_{io} + t_{h2d} + t_f + t_b + t_{comm\_1} + t_u \quad (10)$$

and

$$S = N_g \frac{t_{io\_1} + t_{gpu}}{t_{io\_n_g} + t_{h2d} + t_f + t_b + t_{comm\_1} + t_u}. \quad (11)$$

Second, in a multi-node cluster, TensorFlow also uses PS to store global parameters, which is similar to MXNet. So t_{iter} can be expressed by Equation 6 or Equation 7 and the speedup can be represented by Equation 9.

V. Experimental Methodology

We first specify the hardware environment conducted in the experiments. We use a 4-node GPU cluster, in which each node has four NVIDIA Tesla P40 cards, and the network connection between nodes is a 56 Gbps InfiniBand combined with 1 Gbps Ethernet. Table II shows the hardware setting. The topology of cluster is shown in Fig. 1, and the intra-node topology with bandwidth of data transfer between different components is displayed in Fig. 2. Each Tesla P40 GPU runs at the base core frequency of 1.3 GHz and the auto boost function is disabled to ensure reproducibility of our experimental results.

TABLE II
THE EXPERIMENTAL HARDWARE SETTING FOR DATA PARALLELIZATION.

Hardware	Model
GPU	NVIDIA Tesla P40
CPU	Intel Xeon E5-2650v4 Dual
Network	56 Gbps InfiniBand + 1 Gbps Ethernet
Memory	128 GB DDR4
Hard disk	SSD 6T (x2 with RAID 0) in Node 0, and others are HDD 1T (x4 with RAID 5). Each node has one copy of the dataset

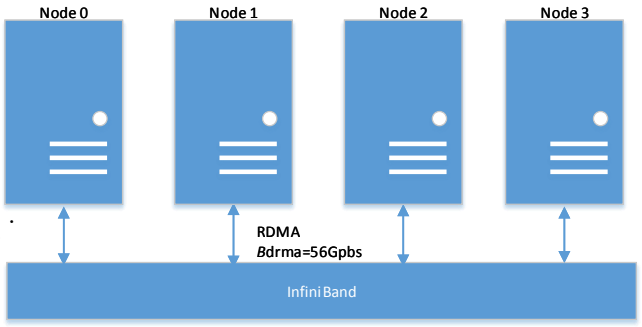


Fig. 1. The topology of the GPU cluster.

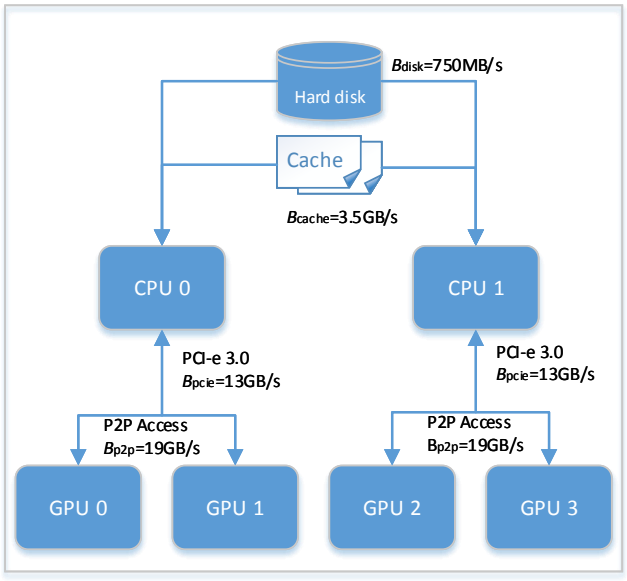


Fig. 2. The topology of a single node. *Cache* refers to the system cache that contains parked file data. The values of B_{disk} and B_{cache} are benchmarked by *dd* command. The bandwidth of PCIe and P2P access are tested via NVIDIA CUDA SDK samples.

Versions of tested frameworks installed in each node are shown in Table III. The operation system of the server is CentOS 7.2, and the software are installed with CUDA-8.0 and cuDNNv6.

TABLE III
THE SOFTWARES USED FOR EXPERIMENTS.

Software	Marjor Version	GitHub Commit ID
Caffe-MPI	2.0	-
CNTK	2.1	4a8db9c
MXNet	0.10.0	34b2798
TensorFlow	1.2.1	-

One popular and effective way to evaluate the running performance is to measure the time duration of an iteration that processes a mini-batch of input data or the number of samples can be processed in one second. Therefore, we benchmark the CNNs by using a proper mini-batch size (try to fully utilize the GPU resource) for each network on these tools.

We choose three popular CNNs (i.e., AlexNet [32], GoogleNet [33] and ResNet-50 [29]) running on the ILSVRC-2012 ImageNet dataset [34]. Each machine in the cluster has one copy of the dataset. The data formats for different frameworks are not the same, and we list the data formats below for the tested frameworks.

- **Caffe-MPI**. LMDB database is used by Caffe-MPI. The original JPEG images are converted LMDB records, and the script can be found in the GitHub repository of Caffe[6].
- **CNTK**. There is no pre-converted data format for CNTK. It needs to read the original JPEG images during training with a provided file list.
- **MXNet**. A binary file that contains all the images is used for MXNet. The converting script refers to the official document of MXNet[7].
- **TensorFlow**. It also uses a pre-converted file format called *TFRecord* in TensorFlow. The converting script refers to the script from the GitHub repository of TensorFlow[8].

These three deep models have their own characteristics to test the performance of frameworks. They have different configurations and the details are shown in Table IV.

TABLE IV
THE EXPERIMENTAL SETUP OF NEURAL NETWORKS.

Network	# of Layers	# of FCs	Parameters	Batch size
AlexNet	8	3	~60 millions	1024
GoogleNet	22	1	~53 millions	128
ResNet-50	50	1	~24 millions	32

Note: The architecture of AlexNet is the same with [32] except that the local response normalization (LRN) operation is excluded because it is not supported by CNTK by default. We choose the proper batch size for each device, which can be run properly for all frameworks, and it tries to fully utilize the GPU resource.

In order to avoid the heavy impact of I/O overheads from hard disks, we run two epochs and the first epoch is excluded to calculate the average time of one iteration. Since the total number of images is up to 1.2 million, it is very time consuming to run all the samples in one epoch. So we limit the epoch size to make each experiment run about 50-100 batches in one epoch. The time of each iteration is recorded and all iterations in the second epoch are averaged to calculate the mean and standard deviation to measure the running performance.

Beside the running speed measured in this paper, we also break down the timing for each phase by using *nvprof*, which is a tool to profile the GPU activities, to help us identify the performance problem.

VI. EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we demonstrate the running performance followed with analysis based on the modeling of CNTK, MXNet and TensorFlow in training AlexNet, GoogleNet and ResNet-50 on a single P40 card, multiple P40 cards, and across the 4-node GPU cluster.

A. Single GPU

We first present the performance results on a single GPU. The average time of one iteration during training is used to metric the performance of frameworks. So we compare the time cost in each step of SGD. The overall performance comparison of three CNNs are shown in Fig. 3. We break down the timing of each phase in Table V. Results in each phase will be analyzed in the following.

TABLE V
THE TIME BREAKDOWN OF DIFFERENT PHASES OF SGD IN SECOND ($mean \pm std$).

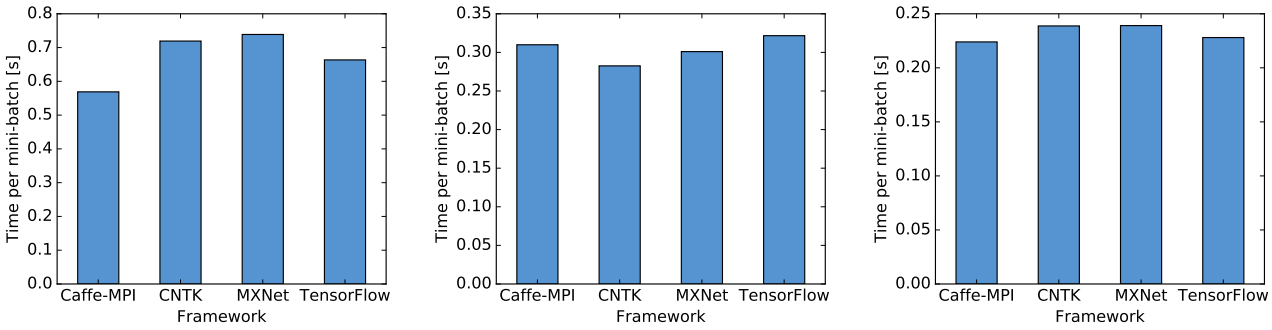
AlexNet	Caffe-MPI	CNTK	MXNet	TensorFlow
t_{io}	.0002±6.9e-05	.2233±5.1e-02	.0001±1.8e-05	.0008±9.3e-04
t_{h2d}	.0526±3.4e-04	.0528±4.1e-04	.1109±1.6e-02	.1140±1.5e-02
t_f	.1718±8.4e-03	.1684±1.9e-03	.2147±3.5e-04	.1804±1.2e-02
t_b	.3560±5.6e-03	.2919±9.3e-04	.4086±1.5e-03	.3417±3.1e-02
t_u	.0062±1.0e-05	.0086±2.3e-06	.0041±4.5e-06	.0031±1.7e-06
t_{iter}	.5772±6.0e-02	.7433±1.0e-02	.7235±1.0e-01	.6593±1.6e-02
GoogleNet	Caffe-MPI	CNTK	MXNet	TensorFlow
t_{io}	.0002±3.8e-04	.0001±8.2e-06	.0001±2.1e-05	.0010±4.6e-04
t_{h2d}	.0021±2.6e-03	.0033±3.3e-03	.0143±1.8e-03	.0161±1.5e-03
t_f	.0920±1.8e-03	.0814±7.5e-04	.0892±1.2e-04	.1192±2.3e-02
t_b	.2073±4.0e-04	.1780±1.4e-03	.1819±2.1e-04	.1856±2.4e-04
t_u	.0015±1.2e-05	.0095±1.7e-03	.0095±1.5e-05	.0005±5.6e-06
t_{iter}	.3050±1.3e-02	.2767±2.0e-02	.2943±8.4e-03	.3100±1.5e-03
ResNet	Caffe-MPI	CNTK	MXNet	TensorFlow
t_{io}	.0002±5.4e-05	.0001±1.1e-05	.0001±9.9e-06	.0003±1.2e-04
t_{h2d}	.0009±7.2e-04	.0016±2.7e-05	.0039±5.7e-04	.0045±1.0e-03
t_f	.0807±4.3e-05	.0724±1.0e-03	.0732±5.4e-05	.0767±1.2e-02
t_b	.1291±5.4e-03	.1482±9.4e-04	.1307±1.4e-04	.1355±1.4e-04
t_u	.0033±4.5e-05	.0073±4.7e-03	.0164±2.5e-05	.0015±1.8e-06
t_{iter}	.2078±5.8e-03	.2261±1.4e-02	.2242±1.2e-02	.2148±3.4e-02

I/O. Among the evaluated tools, they all support data prefetch, which means during training, there are extra threads reading data to the CPU memory for preparing to feed into GPU. However, some implementation details are not the same. Regarding Caffe-MPI, there exist GPU buffers to prefetch the data, which means that each iteration, except the first one, can load data from the GPU memory without waiting

[6]https://github.com/BVLC/caffe/blob/master/examples/imagenet/create_imagenet.sh

[7]https://github.com/apache/incubator-mxnet/tree/master/example/image-classification#prepare-datasets

[8]https://github.com/tensorflow/models/blob/master/research/inception/inception/data/build_imagenet_data.py



(a) AlexNet (b) GoogleNet (c) ResNet-50

Fig. 3. Performance comparison of three networks on a single GPU. (The lower the better)

I/O and PCIe transfer. For CNTK, there could be a limited buffer for data caching, which may result in dropdown of performance if the size of data in a mini-batch is too large. On the contrary, MXNet and TensorFlow are more flexible and have little opportunity to fall into I/O problem. In Table V, CNTK has a big overhead in reading data when running AlexNet with a mini-batch size of 1024. The reason is that it needs $S_d = 1024 \times 224 \times 224 \times 3 \times 4 = 588$MB to store data of one batch, while it is not fast enough to store data of next batch in CNTK. CNTK needs to read and decode the original JPEG files to prepare the input data while other frameworks just need to read from pre-converted files. From Fig. 2, the bandwidth of system cache is $B_{cache} = 3.5$GB/s, so the overhead of reading data is $t_{io} = \frac{S_d}{B_{cache}} = \frac{588}{(3.5 \times 1024)} s = 0.164$s, which is the optimal time, adding the time of decoding JPEG files, the actual value of t_{io} is 0.223s in CNTK. By contrast, MXNet and TensorFlow only need negligible time in reading data.

Memory copy: from host to device (h2d). After reading data from disk to memory, data should be transfered to GPU for training. In our tested environment, CPU and GPU are connected by PCIe with bandwidth B_{pcie} =13 GB/s. It is noticed that t_{h2d} in both Caffe-MPI and CNTK are about half smaller than MXNet and TensorFlow even the size of data is same because of the difference of memory allocation. There are non-pageable and pageable memories, and their performances of memory copy from CPU to GPU are different [35]. The bandwidth of non-pageable and pageable memory copy in the tested hardware are 11.4 GB/s and 8.7 GB/s respectively. Since Caffe-MPI and CNTK allocate the non-pageable memory for input data, while MXNet and TensorFlow allocate pageable memories, so CNTK achieves better memory copy performance than that of MXNet and TensorFlow.

Forward, backward and update. The high performance library cuDNN [14] on GPUs, provided by NVIDIA, has been widely used in DL frameworks. During the training of DNNs, most of time consuming layers (e.g., convolutional, fully connected) are invoked by cuDNN. However, parameters in APIs of cuDNN may result in different performances, which is the main reason why t_f and t_b are different in Table V. For example, there are many types of implementations of convolution like GEMM, FFT and Winograd. Users can specify which algorithm to use or autotune to choose the best one. When invoking the APIs of cuDNN, another performance related factor is the data layout (e.g., NCWH, NWHC). In both forward and backward phases, CNTK achieves the best performance in all networks. Actually, Caffe-MPI, CNTK and MXNet could autotune to find the best convolution algorithms for convolutional layers, but TensorFlow prefers to use Winograd algorithm which could be suboptimal in some cases. Regarding AlexNet, CNTK invokes the FFT algorithm for the second convolutional layer, while MXNet uses the GEMM-based convolution so that there is 0.04s larger in the forward phase and up to 0.1s higher in the backward phase. The FFT-based convolution is faster than the GEMM-based in general [36]. The suboptimal invoking of cuDNN APIs makes TensorFlow slightly worse compared to CNTK in both forward and backward phases. The update operation is much simpler since it only updates the parameters with computed gradients, and the complexity is O(1), so there is no obvious difference in this phase.

In summary, CNTK has faster data copy, forward and backward propagation operations, which results in better performance in GoogleNet compared to MXNet and TensorFlow. Caffe-MPI outperforms CNTK in AlexNet since Caffe-MPI can hide the overhead of data I/O. However, the test of GoogleNet has two advantage for CNTK. First, there are many large size of filters (e.g., 5×5) in convolutional layers, which could reflect the importance of convolution algorithm selection. Second, the mini-batch size used is only 128, such that it has a very small overhead in data loading. MXNet and TensorFlow have better data prefetch mechanism than CNTK. It is obvious in the case of AlexNet, though TensorFlow has a suboptimal performance in data copy, forward and backward, it hides the overhead of data reading. As a result, TensorFlow achieves 10% better performance than CNTK in AlexNet. Regarding ResNet-50, convolutional layers are with smaller kernels (e.g., 3×3) which requires less computation and the Winograd algorithm could be the better implementation [37].

B. Multiple GPUs

When scaling to multiple GPUs, it is important to shorten the overhead of data aggregation in each iteration, which heavily relies on the bandwidth of data communication between GPUs. Please be noted that in multi-GPU/node testing we use weak scaling, which means the valid mini-batch size is scaling with the number of GPUs, and each GPU keeps the same number of samples like the work in [4][5][6]. To reflect the scalability of deep learning frameworks, we use the metric of samples per second to compare the performance. Ideally, the number of samples per second should be doubled with the number of GPUs doubled. Fig. 4 shows the scaling performance S-SGD running on a machine with four GPUs. Let t_{comm} denote the overhead of data communication when aggregating the gradients, and the numbers are shown in Table VI.

TABLE VI
THE OVERHEAD OF DATA COMMUNICATION WHEN AGGREGATING THE GRADIENTS ACROSS INTRA-NODE MULTIPLE GPUS.

Network	Tool	t_{comm}		Hidden
		2 GPUs	4 GPUs	
AlexNet	Caffe-MPI	.0457±3.1e-03	.0861±1.2e-02	Yes
	CNTK	.0359±2.4e-02	.0420±3.0e-03	No
	MXNet	.0222±4.5e-03	.0505±7.7e-03	Yes
	TensorFlow	.0406±8.9e-03	.0984±1.9e-02	Yes
GoogleNet	Caffe-MPI	.0135±1.4e-03	.0229±1.2e-03	Yes
	CNTK	.0343±1.2e-02	.0592±6.2e-03	No
	MXNex	.0102±8.9e-04	.0318±1.1e-02	Yes
	TensorFlow	.0053±3.8e-04	.0168±1.6e-03	Yes
ResNet-50	Caffe-MPI	.0336±2.4e-03	.0646±2.9e-03	Yes
	CNTK	.0173±1.2e-02	.0295±2.1e-02	No
	MXNex	.0491±4.1e-02	.1626±1.1e-01	Yes
	TensorFlow	.0072±2.1e-03	.0210±5.0e-03	Yes

From Fig. 4 (a), we can see that Caffe-MPI, MXNet and TensorFlow have achieved almost linear scaling from one to two GPUs, while CNTK has only a slight speedup with multiple GPUs. Caffe-MPI, MXNet and TensorFlow parallelize the gradient aggregation with backward propagation. In other word, the previous layer (l_{i-1}) of backward propagation can happen without any delay after gradients of current layer (l_i) computed, and at this time, gradient computation of l_{i-1} is parallelized with gradient aggregation of l_i. In this way, much of the synchronization overhead of early gradients can be hidden by later computation layers. From Table VI, it is noted that Caffe-MPI, MXNet and TensorFlow can hide t_{comm} while CNTK does not. CNTK processes gradient computation and aggregation in a sequential way. Fortunately, the overhead of gradient aggregation can be highly reduced by high performance all-reduce library NCCL which is used by CNTK.

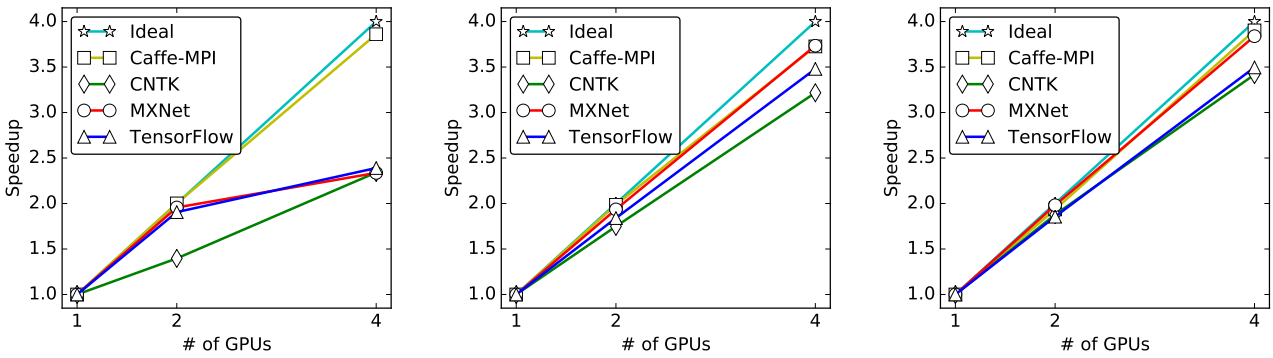
Regarding AlexNet, the low scaling efficiency of CNTK is caused by the data reading from disk to memory. Since the data buffer is not fast enough to prefetch the next-batch data, the GPU computation needs to wait for the data loading from disk to memory in every iteration. Suppose that the data of one epoch has been loaded into the system cache, and the data size is up to $S_d = 588N_g$ MB, we have $t_{io} = \frac{S_d}{B_{cache}} = \frac{588\times N_g}{3.5\times 1024} = 0.164N_g$. In the tested cases of CNTK on AlexNet, t_{io} is up to 0.45s and 0.72s with 2 and 4 GPUs respectively so that CNTK has a poor scaling performance with 2 GPUs and 4 GPUs. From Table V, we have $t_{gpu} = 0.0527 + 0.1684 + 0.2918 + 0.0086 = 0.5215$. According to Equation 8, $S = \frac{N_g(0.223+0.5215)}{0.5215+t_{io\_Ng}+t_{comm}}$. For 2 GPUs, $S = \frac{2\times 0.7445}{0.5215+0.45+0.0359} = 1.478$, and for 4 GPUs, we have $S = \frac{4\times 0.7445}{0.5215+0.72+0.042} = 2.32$. The estimated speedup can match the evaluated results in Fig. 4. There is a similar scenario on 4-GPU training of AlexNet (0.55s for data reading) using MXNet and TensorFlow. In S-SGD with multiple GPUs in a single machine, every data reading thread fetches 4096 samples and distributes them to 4 GPUs. Since t_{io} is longer than t_{gpu}, the I/O time cannot be hidden totally, which results in poor scaling across 4 GPUs.

For GoogleNet and ResNet-50, CNTK achieves worse scaling performance compared to Caffe-MPI, MXNet and TensorFlow since CNTK does not parallelize the gradient computation and aggregation. Since the overhead of I/O can be hidden this time, according to Equation 8, gradients aggregation becomes the main factor that influences the scaling performance in CNTK who does not parallelize the gradient communication with backward propagation. MXNet achieves better scaling efficiency than TensorFlow. The intra-node S-SGD implementations of MXNet and TensorFlow are different. In MXNet, there is a parameter server (on GPU) to keep a copy of parameters. When gradients of layer l_i have been calculated, the gradients from multiple GPUs are transfered to the PS, and then PS aggregates gradients and updates the model parameters directly, and then it copies parameters back to all GPUs. In this way, MXNet can hide both overhead of gradient synchronization and model updating. By contrast, TensorFlow implements S-SGD in a different way. It has no PS, and it uses peer-to-peer memory access if the hardware topology supports it. Beside the decentralized method, another main difference is that each GPU needs to average the gradients from other GPUs and updates the model after backward propagation finished. Therefore, that the model updating is not overlapped with backward propagation leads to suboptimal scaling performance of TensorFlow. To analyze the impact of the update operation, let $r_{u2g} = \frac{t_u}{t_{h2d}+t_f+t_b}$. According to Equation 11, the higher value r_{u2g} is, the lower speedup of TensorFlow. From Table V, $r_{u2g} = \frac{0.0005}{0.016+0.1129+0.1856} = 0.16\%$ in GoogleNet, and $r_{u2g} = \frac{0.0015}{0.045+0.0767+0.1345} = 0.59\%$ in ResNet-50, so TensorFlow scales slightly worse in ResNet-50 compared to GoogleNet.

In conclusion, two things are important to reduce the impact of gradient aggregation in S-SGD. On one hand, high speed of data communication between GPUs is important to ease the overhead of gradients synchronization. On the other hand, parallelism between communication and computation is necessary to hide the overhead of communication.

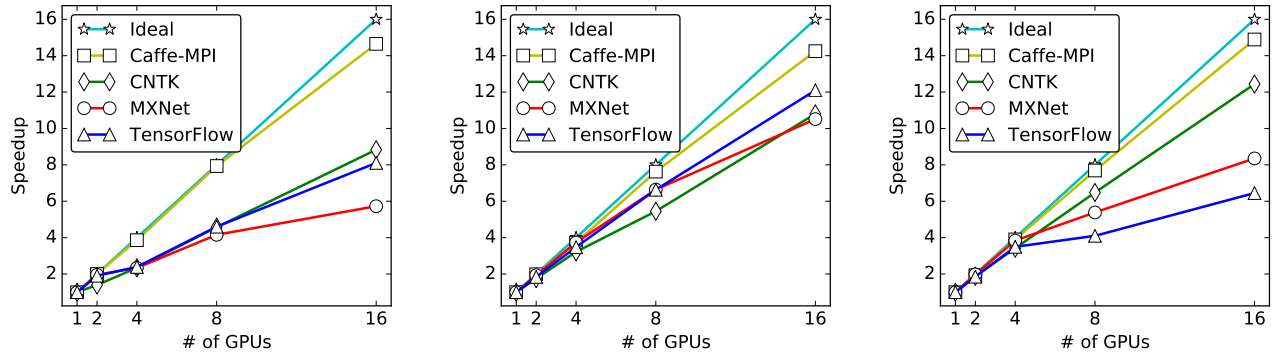
C. Multiple machines

It is more chanlleging to hide the overhead of data communication across multiple servers since the bandwidth (or



(a) AlexNet (b) GoogleNet (c) ResNet-50

Fig. 4. Scaling performance of three networks with multiple GPUs in a single node.



(a) AlexNet (b) GoogleNet (c) ResNet-50

Fig. 5. Scaling performance of three networks with multiple machines. Please be noted that each machine has 4 GPUs, and the cases of 8 GPUs and 16 GPUs are across 2 machines and 4 machines respectively.

TABLE VII
THE OVERHEAD OF DATA COMMUNICATION WHEN AGGREGATING THE GRADIENTS ACROSS MULTIPLE MACHINES.

Network	Tool	t_{comm}		Hidden
		2 nodes	4 nodes	
AlexNet	Caffe-MPI	.1344±6.6e-03	.1650±1.9e-02	Yes
	CNTK	.0906±9.7e-03	.2364±7.5e-02	No
	MXNet	.5004±2.5e-02	.7513±5.6e-01	No
	TensorFlow	-	-	No
GoogleNet	Caffe-MPI	.1161±4.7e-02	.0867±3.0e-02	Yes
	CNTK	.1032±3.6e-02	.1105±1.5e-02	No
	MXNet	.2973±1.9e-01	.4505±2.2e-01	No
	TensorFlow	-	-	No
ResNet-50	Caffe-MPI	.1019±2.4e-03	.1325±2.7e-03	Yes
	CNTK	.0485±9.9e-03	.0595±1.9e-02	No
	MXNet	.4994±1.1e-01	.5561±3.7e-01	No
	TensorFlow	-	-	No

Notes: Due to *grpc* hidden in TensorFlow, we could not get the accurate overhead of gredient communication accross multiple machines.

latency) of network interface is much smaller (or longer) than PCIe or NVLink. In our experiments, we set up the InfiniBand with a theoretical bandwidth of 56 Gbps, which is about a half of PCIe. Scaling performances across multiple machines are shown in Fig. 5.

From Table VI, it is noted that the communication time is hidden in Caffe-MPI, MXNet and TensorFlow. However, when scaling to multiple machines, the overhead of gradient aggregation across multiple machines could not be reduced easily, which is shown in Table VII. It is noted that the overhead of communication is not hidden in the inter-node environment except Caffe-MPI. Even though in the intra-node parallelism between gradient aggregation and backward propagation, the inter-node communication could cause the scaling performance drop down seriously. Both MXNet and TensorFlow use the PS method to synchronize the gradients across machines. The PS should collective the gradients from different machines via 56 Gbps InfiniBand, which has only 7 GB/s bandwidth and a high latency if the data transfer is not well optimized. Among the tested frameworks, they use different methodologies in communication across machines. Caffe-MPI implements the gradient aggregation with a de-centralized method via NCCL2.0, and it parallels with the backward propagation. Since the NCCL2.0 could collective the gradients in an efficient way, the overhead of communication can be hidden by the computation. CNTK also uses

NCCL2.0 to do the all-reduce, MXNet exploits TCP socket communication, and TensorFlow makes use of *grpc*[9] which is a high performance remote process call (RPC) framework.

NCCL has high efficiency and low latency in doing collective communications via GPUDirect in the GPU cluster. For example, t_{comm} of 2 GPUs and 4 GPUs on CNTK with AlexNet are 0.0906 and 0.236 respectively, and the size of gradients for communication is up to $S_g = 63 \times N_g$ MB. The all-reduce efficiency of CNTK (with NCCL2.0) is:

$$E_{allreduce} \approx \frac{\frac{S_g}{t_{comm}}}{B_{rdma}} = \frac{S_g}{t_{comm} B_{rdma}}. \quad (12)$$

It is known that B_{rdma} =56Gbps = 7GB/s, so $E_{allreduce}$ in 2 nodes (8 GPUs) and 4 nodes (16 GPUs) are $\frac{63\times8}{0.0906\times7\times1024} = 77.61\%$ and $\frac{63\times16}{0.236\times7\times1024} = 59.59\%$ respectively. However, the overhead of communication is also heavy compared to the time of computation, for example, $t_{comm} = 0.1105$s and $t_{gpu} = 0.0033 + 0.081 + 0.1780 + 0.0095 = 0.2722$s in GoogleNet. At last, the overall scaling efficiencies of CNTK are about 55%, 67.5% and 77.7% in AlexNet, GoogleNet and ResNet-50 respectively when training on 4 machines.

MXNet expoits the customized KVStore [11][23] technique. Even though it makes the framework be equipped with ability to scale to distributed environments easily, it also requires a very high quality of network to achieve better performance improvement or scalability. In the tested cases, when scaling to four machines, the communication overhead could become larger and leads to the low scaling efficiency due to the high latency and low actual bandwidth during the gradient communication. For example, the communication overhead is up to 0.4505s, while the backward propagation only needs 0.1819s in GoogleNet with four machines. The overhead of gradient aggregation can not be hidden by backward propagation. Therefore, compared to its scalability of intra-node with multiple GPUs, MXNet performs lower scaling efficiency across multiple machines. As a result, MXNet achieves efficiencies of 35.625%, 65.625% and 35.6% in AlexNet, GoogleNet and ResNet-50 respectively in our multi-node evaluation.

gprc is the remote communication framework for TensorFlow, and RDMA used for TensorFlow may not be optimal, which results in relatively high latency compared to NCCL. When pipelining layer-wise gradients computation and aggregation, it is important that the time of aggregation should be smaller than computation such that the time of aggregation can be eliminated. Looking at the architecture of AlexNet and GoogleNet, the number of layers is small, and the computation of convolutional layers (big kernel size) is heavy. So it is easy to hide the latency of data copy in such scenario. By contrast, ResNet-50 has deeper layers and smaller kernel size (most are 3×3 and 1×1 kernels) of convolutional layers, which requires more frequent communication of gradients but less computation of gradients, so the overhead is hard to hide since gradients of the previous layer are calculated too fast. Scaling to four machines, TensorFlow achieves scaling efficiencies of 50.625%, 75.56% and 52.187% in AlexNet, GoogleNet and ResNet-50 respectively.

To summarize, not only the high-speed network is required to provide fast transfer of gradients, but it also gives a big challenge to the frameworks in optimizing the data communication across multiple machines to better utilize the hardware. Due to the high GFLOPS in a multi-GPU server (e.g., a server with 4 P40 GPUs), it makes the network and the implementation of S-SGD more challenging to achieve high efficiency.

To make a direct view of all our tested results, we put all the numbers in Table VIII.

TABLE VIII
SPEED OF ALL TESTED CASES.

Network	Framework	Speed (# of Samples per second)				
		1 GPU	2 GPUs	4 GPUs	8 GPUs	16 GPUs
AlexNet	Caffe-MPI	1800	3602	6948	14283	26371
	CNTK	1423	1988	3332	6517	12574
	MXNet	1386	2711	3238	5759	7939
	TensorFlow	1543	2941	3689	7102	12511
GoogleNet	Caffe-MPI	413	820	1539	3151	5886
	CNTK	453	792	1457	2469	4894
	MXNet	425	822	1588	2824	4470
	TensorFlow	397	732	1384	2639	4814
ResNet-50	Caffe-MPI	142	276	557	1098	2127
	CNTK	134	251	457	868	1666
	MXNet	133	265	513	720	1118
	TensorFlow	134	260	490	575	905

VII. CONCLUSION AND FUTURE WORK

In this work, we evaluate the performance of four popular distributed deep learning frameworks (Caffe-MPI, CNTK, MXNet and TensorFlow) over a 4-node dense GPU cluster (four Tesla P40 GPUs each node) connected with 56 Gbps InfiniBand via training three CNNs (AlexNet, GoogleNet and ResNet-50). We first build performance models to measure the speedup of synchronous SGD including different implementations from Caffe-MPI, CNTK, MXNet and TensorFlow. Then we benchmark the performances of these four frameworks covering single-GPU, multi-GPU and multi-machine environments. According to the experimental results and analysis, it shows some performance gaps among four different implementations, and there exist suboptimal methods that could be further optimized to improve the performance in evaluated frameworks in terms of I/O, cuDNN invoking, data communication across intra-node GPUs and inter-node GPUs.

For future work, we plan to evaluate the scalability of DL frameworks across low-bandwidth or high-latency networks (e.g., 1 Gbps Ethernet). And the distributed SGD may include asynchronous SGD and model parallelism.

VIII. ACKNOWLEDGEMENTS

We would like to thank Inspur (Beijing) Co., Ltd for providing the GPU cluster for experiments.

[9]grpc: https://grpc.io/

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[3] A. Viebke and S. Pllana, "The potential of the Intel (R) Xeon Phi for supervised deep learning," in *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*. IEEE, 2015, pp. 758–765.

[4] W. Wang and N. Srebro, "Stochastic nonconvex optimization with large minibatches," *arXiv preprint arXiv:1709.08728*, 2017.

[5] Y. You, I. Gitman, and B. Ginsburg, "Scaling SGD batch size to 32k for ImageNet training," *arXiv preprint arXiv:1708.03888*, 2017.

[6] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: Training ImageNet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[7] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing, "On model parallelization and scheduling strategies for distributed machine learning," in *Advances in neural information processing systems*, 2014, pp. 2834–2842.

[8] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, 2010, pp. 2595–2603.

[9] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th annual international conference on machine learning*. ACM, 2009, pp. 873–880.

[10] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD," *arXiv preprint arXiv:1604.00981*, 2016.

[11] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous systems, 2015," *Software available from tensorflow. org*, vol. 1, 2015.

[13] D. Yu, A. Eversole, M. Seltzer, K. Yao, Z. Huang, B. Guenter, O. Kuchaiev, Y. Zhang, F. Seide, H. Wang *et al.*, "An introduction to computational networks and the computational network toolkit," Technical report, Tech. Rep. MSR, Microsoft Research, 2014, 2014. research. microsoft. com/apps/pubs, Tech. Rep., 2014.

[14] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[15] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of deep learning software frameworks," *arXiv preprint arXiv:1511.06435*, 2015.

[16] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," in *Proceedings of the 7th International Conference on Cloud Computing and Big Data, IEEE, Macau, China*, 2016.

[17] S. Shams, R. Platania, K. Lee, and S.-J. Park, "Evaluation of deep learning frameworks over different HPC architectures," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 1389–1396.

[18] H. Kim, H. Nam, W. Jung, and J. Lee, "Performance analysis of CNN frameworks for GPUs," in *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 55–64.

[19] D. E. Rumelhart, G. E. Hinton, R. J. Williams *et al.*, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.

[20] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," *arXiv preprint arXiv:1611.03530*, 2016.

[21] D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey, "Distributed deep learning using synchronous stochastic gradient descent," *arXiv preprint arXiv:1602.06709*, 2016.

[22] Y. You, Z. Zhang, C.-J. Hsieh, and J. Demmel, "100-epoch ImageNet training with alexnet in 24 minutes," *arXiv preprint arXiv:1709.05011*, 2017.

[23] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server." in *OSDI*, vol. 1, no. 10.4, 2014, p. 3.

[24] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 4.

[25] S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu, "Asynchronous stochastic gradient descent for DNN training," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 6660–6663.

[26] S.-X. Zou, C.-Y. Chen, J.-L. Wu, C.-N. Chou, C.-C. Tsao, K.-C. Tung, T.-W. Lin, C.-L. Sung, and E. Y. Chang, "Distributed training large-scale deep architectures," *arXiv preprint arXiv:1709.06622*, 2017.

[27] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-caffe: Co-designing MPI runtimes and Caffe for scalable deep learning on modern GPU clusters," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2017, pp. 193–205.

[28] A. A. Awan, C.-H. Chu, H. Subramoni, and D. K. Panda, "Optimized broadcast for deep learning workloads on dense-GPU infiniband clusters: MPI or NCCL?" *arXiv preprint arXiv:1707.09414*, 2017.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[30] X. Lian, C. Zhang, H. Zhang, C.-J. Hsieh, W. Zhang, and J. Liu, "Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent," *arXiv preprint arXiv:1705.09056*, 2017.

[31] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," *arXiv preprint arXiv:1706.03292*, 2017.

[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[33] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[34] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 248–255.

[35] C. Nvidia, "NVIDIA CUDA C programming guide," *Nvidia Corporation*, vol. 120, no. 18, p. 8, 2011.

[36] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through FFTs," *arXiv preprint arXiv:1312.5851*, 2013.

[37] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4013–4021.