An In-Depth Analysis of Distributed Training of Deep Neural Networks

Yunyong Ko
Dept. Computer Software
Hanyang University
Seoul, Korea
koyunyong@hanyang.ac.kr

Kibong Choi
Dept. Computer Software
Hanyang University
Seoul, Korea
rlqhd26@hanyang.ac.kr

Jiwon Seo[*]
Dept. Computer Software
Hanyang University
Seoul, Korea
seojiwon@hanyang.ac.kr

Sang-Wook Kim[*]
Dept. Computer Software
Hanyang University
Seoul, Korea
wook@hanyang.ac.kr

***Abstract*—As the popularity of deep learning in industry rapidly grows, efficient training of deep neural networks (DNNs) becomes important. To train a DNN with a large amount of data, distributed training with data parallelism has been widely adopted. However, the communication overhead limits the scalability of distributed training. To reduce the overhead, a number of distributed training algorithms have been proposed. The model accuracy and training performance of those algorithms can be different depending on various factors such as cluster settings, training models/datasets, and optimization techniques applied. In order for someone to adopt a distributed training algorithm appropriate for her/his situation, it is required for her/him to fully understand the model accuracy and training performance of these algorithms in various settings. Toward this end, this paper reviews and evaluates seven popular distributed training algorithms (BSP, ASP, SSP, EASGD, AR-SGD, GoSGD, and AD-PSGD) in terms of the model accuracy and training performance in various settings. Specifically, we evaluate those algorithms for two CNN models, in different cluster settings, and with three well-known optimization techniques. Through extensive evaluation and analysis, we made several interesting discoveries. For example, we found out that some distributed training algorithms (SSP, EASGD, and GoSGD) have highly negative impact on the model accuracy because they adopt intermittent and asymmetric communication to improve training performance; the communication overhead of some centralized algorithms (ASP and SSP) is much higher than we expected in a cluster setting with limited network bandwidth because of the PS bottleneck problem. These findings, and many more in the paper, can guide the adoption of proper distributed training algorithms in industry; our findings can be useful in academia as well for designing new distributed training algorithms.**

***Index Terms*—deep learning, distributed training algorithm**

I. Introduction

Deep neural networks (DNNs) are widely adopted in industry, and they are used to provide increasingly more complex functionality. Thus, recent DNN models are becoming more complicated, for example, with residual connections and deeper network structures [10], [25]. These models are trained with a large amount of data, which requires massive training time and computing power. To speed up the training, distributed training with data parallelism is commonly applied; each worker machine given with a subset of training data iteratively trains its local parameters. After each training iteration, workers communicate the computed gradients (or parameters) either in a centralized manner (using parameter servers) or in a decentralized manner (via peer-to-peer communication). While distributed training is generally effective, the aggregation overhead can be non-trivial as the number of workers increases; particularly in a cluster with low network bandwidth, the overhead can be much costly. A recent study reported that the overhead can be more than 80% of the entire training [22].

To reduce this overhead, a number of distributed training algorithms have been studied [9], [13], [16], [18]–[20], [26], [27]. These algorithms aim to improve the training performance by reducing the communication overhead required for parameter aggregation, while maintaining the model accuracy, as much as possible. In general, the model accuracy and training performance are in a trade-off relationship; that is, reducing the communication overhead for the aggregation improves the training performance, but it may have an adverse effect on the model accuracy. Thus, in order to choose the best distributed training algorithm, it is necessary to fully understand the parameter aggregation process of each algorithm and its effect on the model accuracy and training performance.

The model accuracy and training performance of distributed training algorithms can be different depending on various aspects such as cluster settings (e.g., the number of workers, computing power of GPUs, or network bandwidth), training models/datasets, and optimization techniques applied. For example, some asynchronous centralized algorithms (e.g., ASP and SSP) show much better training performance than synchronous algorithms (e.g., BSP) on a network with sufficient bandwidth, while achieving high model accuracy comparable to that of BSP. However, they would be slower than synchronous ones if network bandwidth is limited because a specific server (i.e., a parameter server) becomes the communication bottleneck. The PS (parameter server) bottleneck problem can be resolved if an optimization technique to reduce the amount of communication (e.g., gradient compression) is applied. Therefore, for this reason, these distributed training algorithms should be extensively evaluated with respect to various aspects.

However, these algorithms have not been fully understood in those aspects so far; they have been evaluated only in a

*Corresponding authors

specific environment of cluster settings, DNN models, DNN frameworks, and optimization techniques. On the other hand, it is not feasible in practice for a user to evaluate all of the existing distributed training algorithms in various aspects because it requires a lot of time and effort. In this paper, we review existing distributed training algorithms in terms of the parameter aggregation process and its effect on the model accuracy and training performance. We implement and evaluate the distributed training algorithms in a fair way and perform comprehensive analysis of the evaluation results. With the evaluation and analysis, we aim to help users select the suitable algorithm for their own environment and provide the guidance for designing new distributed training algorithms.

For the evaluation, we carefully select seven distributed training algorithms including centralized and decentralized ones that are widely recognized in the literature [6], [9], [11], [13], [15], [16], [19], [20], [24], [26]. Table I shows the algorithms that we evaluate in this paper. We also test them with three well-known optimization techniques to understand how the optimizations affect distributed training algorithms in different settings. Only four of the seven algorithms – BSP, ASP, SSP, and AllReduce – are already implemented in more than one deep learning (DL) framework (Tensorflow [3], MXNet [2], PyTorch [5], or MS DMTK [1]). For fair evaluation, we carefully re-implemented the seven distributed training algorithms and three optimization techniques in the same DL framework (Google TensorFlow). We conduct an in-depth analysis for them in the following four aspects; (1) the model accuracy, (2) hyperparameter sensitivity, (3) scalability, and (4) the effectiveness of existing optimizations on them.

The remainder of this paper is organized as follows. Section II describes the preliminary of distributed training for DNN models in general. Sections III and IV review the distributed training algorithms that we evaluate in this paper. Section V describes optimization techniques that are commonly used in DNN training. Section VI presents the evaluation results of the distributed training algorithms and our in-depth analysis on them. Finally, Section VII concludes the paper.

II. Preliminary

A. Deep Learning

Deep learning is a machine learning technique to train deep neural network models (DNNs). DNNs have a series of layers of neurons that transfer signals via weight parameters, which collectively approximate an objective function (also called a target function). To find the optimal values for the weight parameters with a gradient descent technique, backpropagation is applied to efficiently compute the derivatives of the parameters.

One particular aspect of deep learning is its large amount of parameters – a modern DNN model can easily have hundreds of millions of parameters [10], [14], [25]. Such a large number of parameters are updated iteratively for each training input, i.e., $x_t = x_{t-1} - \eta \cdot \nabla\mathcal{L}(x_{t-1}; \xi_t)$, where x_t is a set of parameters at training iteration t, η is the learning rate, $\mathcal{L}$ is the loss function, and ξ_t is a set of data samples at iteration t. Because the number of parameters is large in modern DNN models and so does the size of a training dataset, the amount of computations for training DNN parameters is substantially large. To speed up the training of DNN models, parallel hardware accelerators such as GPUs or TPUs are typically used [8], [13].

B. Distributed Training for DNN models

With the increasing size of a training dataset and DNN model parameters, a single GPU often cannot process DNN training in practice. For scalable training of DNN models, distributed training algorithms have been widely studied [6], [8], [9], [11], [13], [15], [16], [18], [20], [24], [27]. Distributed training has two styles – data parallelism and model parallelism. This work focuses on data parallelism, where training data is split into partitions, each of which is stored and learned at a worker while all workers use the same model.

In distributed training, each iteration consists of two stages; computation and communication. In a computation stage, each worker trains its local parameters based on its local training data. In a communication stage, the parameters (or gradients) of workers are aggregated via communication, which are used in the subsequent training iteration. For this aggregation, either communication via a separate centralized parameter server (i.e., centralized), or peer-to-peer communication (i.e., decentralized) can be used. Such an aggregation can be performed either synchronously or asynchronously.

In general, distributed training aims to find global parameters $\tilde{x}$ that minimize the following equation:

$$\min \sum_{i=1}^{n} \mathbb{E}[\mathcal{L}(x^i, \xi^i)] + \frac{\rho}{2}||x^i - \tilde{x}||^2 \tag{1}$$

where $\mathcal{L}$ is a loss function, x^i is the set of local parameters of worker i, and ξ^i is the dataset of worker i. The goal of Eq. 1 is twofold: the first term minimizes the loss function (thus maximizing accuracy) and the second term minimizes the variance between local and global parameters. The global model parameter $\tilde{x}$ is computed differently depending on distributed training algorithms. We will describe details of distributed training algorithms in Sections III and IV.

III. Centralized Training

We review existing distributed DNN training algorithms in recent literature. We found ten algorithms [6], [9], [11], [13], [15], [16], [19], [20], [24], [26] and selected seven of them in Table I for our evaluation based on their popularity (number of DL frameworks supporting them), effectiveness (convergence rate), and the theoretical guarantee on the convergence. In this section and the following one, we describe those seven algorithms focusing on the following three aspects; (1) parameter aggregation: how does each algorithm aggregate the parameters of workers? (2) accuracy and performance: how effectively does each algorithm train the model in terms of the model accuracy and training performance? and (3) our implementation: how do we implement each algorithm,

TABLE I
SUMMARY OF DISTRIBUTED TRAINING ALGORITHMS

	Centralized				Decentralized		
	Synchronous	Asynchronous			Synchronous		Asynchronous
Name	**BSP**	**ASP**	**SSP**	**EASGD**	**AR-SGD**	**GoSGD**	**AD-PSGD**
Converge. Rate	$O(\frac{1}{\sqrt{NK}})$	$O(\frac{1}{\sqrt{NK}})$	$O(\sqrt{\frac{2(s+1)N}{K}})$	-	$O(\frac{1}{\sqrt{NK}})$	-	$O(\frac{1}{\sqrt{K}})$
Comm. Complexity	$O(2MN \cdot \frac{1}{l})$	$O(2MN)$	$O((1+\frac{1}{s+1}) \cdot MN)$	$O(2MN \cdot \frac{1}{\tau})$	$O(2MN)$	$O(MN \cdot p)$	$O(MN)$

addressing implementation issues? We first describe the centralized algorithms that have separate parameter servers (PS) with global parameters $\tilde{x}$.

A. Bulk Synchronous Parallel (BSP)

Parameter aggregation. In BSP, the model parameters of all workers are synchronized by aggregating the gradients from all workers at once. In each iteration, each worker sends its computed gradients to PS and waits for PS to return the updated parameters. PS aggregates the gradients from all workers, updates the global parameters using them, and broadcasts the updated parameters to all workers.

Accuracy and performance. In BSP, each of workers has the same parameter values since the model parameters of all workers are *synchronized* at every iteration. Thus, the training is consistent across the workers, which helps to improve the accuracy of the trained model. The theoretical convergence rate of BSP is known as $O(\frac{1}{\sqrt{NK}})$ [12]; here N is the number of workers and K is the number of training iterations. Note that the convergence rate is the difference between the current parameters and the optimal parameters at iteration K; thus the smaller the rate is, the faster the parameters converge to the optimal values. However, the synchronization overhead might be quite significant when some workers fall behind other workers for gradient computation (i.e., straggler), which may degrade the training performance significantly. Because every worker sends gradients and receives the updated parameters, the communication complexity of BSP is $O(2MN)$, where M and N are the numbers of parameter size and workers.

Our implementation. Note that we used TensorFlow 1.12 and MPICH 3.1.4 to implement all algorithms. We additionally implemented local aggregation that reduces the amount of communication by aggregating the gradients computed by multiple workers (GPUs) on a same machine [9], [18]. Through local aggregation, the communication complexity is reduced from $O(2MN)$ to $O(2MN \cdot \frac{1}{l})$ where l is the number of workers (GPUs) in a same machine. We will discuss the effect of the local aggregation in Section VI.

B. Asynchronous Parallel (ASP)

Parameter aggregation. To improve the training performance of BSP, PS in ASP processes the gradients from workers in an *asynchronous* manner. That is, PS aggregates the gradients of each worker, *immediately* updates the global parameters using them, and then it sends the updated global parameters right back to the worker.

Accuracy and performance. ASP does not suffer from the straggler problem because fast workers do not wait for slow workers. However, PS may become the bottleneck of entire training because all of the workers communicate with PS individually without local aggregation. In each iteration, every worker sends gradients and receives the updated parameters, so the communication complexity of ASP is $O(2MN)$. We observed that the training performance of ASP is worse than even that of BSP on a cluster with limited network bandwidth, which will be discussed in Section 6.

The variance among the parameters of workers could be significant if there are workers with different training speeds, which may adversely affect the model accuracy. Nonetheless, the theoretical convergence rate of ASP is $O(\frac{1}{\sqrt{NK}})$ which is same as that of BSP. We will verify the accuracy of ASP experimentally in Section 6.

Our implementation. In order to efficiently process the network I/O of PS, we made communication threads allocated to PS as many as the number of workers. Since a communication thread is in charge of one worker, PS can communicate with multiple workers in parallel at the same time. To mitigate the communication overhead of PS, we additionally implemented the parameter sharding technique [8], [9], [18]. For fair evaluation, we applied this optimization to all the centralized algorithms (BSP, ASP, SSP, and EASGD).

C. Stale Synchronous Parallel (SSP)

Parameter aggregation. SSP relaxes the parameter synchronization by allowing the workers to train with different parameter versions. In SSP, a threshold s is defined to represent the maximum difference of parameter versions among workers, allowed for the training. Thus, SSP guarantees that the difference between the parameter versions of each worker and the slowest worker is less than the threshold s.

In each iteration, each worker sends the computed gradients to PS and decides whether it requests the aggregated global parameters to PS based on its staleness. If the staleness of a worker is less than the threshold, it updates its parameters locally and proceeds to the next iteration without waiting for PS. Otherwise, it requests the aggregated global parameters to PS. PS aggregates gradients from each worker, immediately updates the global parameters using them in each iteration.

Unlike ASP, PS sends the global parameters to a worker only if the worker requests the global parameters.

Accuracy and performance. SSP controls the model accuracy and training performance by varying the staleness threshold s. The larger the threshold is, the less frequently a worker receives the aggregated global parameters from PS, thereby improving the training performance. For example, if s is 0, SSP is equivalent to BSP, while, if s is ∞, it is equivalent to local training only (i.e., ensemble). The convergence rate of SSP is $O(\sqrt{\frac{2(s+1)N}{K}})$ and the communication complexity is $O((1+\frac{1}{s+1}) \cdot MN)$.

Our implementation. We implemented SSP as described in [16]. If a worker's staleness is within the limit of the threshold, the worker performs two tasks; (i) sending the computed gradients to PS and (ii) updating its local parameters using the gradients. These two tasks are independent from each other. To further improve the training throughput, we implemented the two tasks to be executed in parallel.

D. Elastic Averaging SGD (EASGD)

Parameter aggregation. EASGD [26] aims to reduce the communication overhead required for parameter aggregation to improve the training performance. To reduce the overhead, it adopts *periodic* communication between PS and workers. In each iteration, a worker communicates its parameters with PS every τ iterations, where τ is a hyperparameter that controls the communication period.

Accuracy and performance. In EASGD, communication period τ indirectly controls the training performance and model accuracy. With a large value of τ, the workers less frequently communicate with PS, which improves the training performance. However, it may negatively impact the model accuracy because the local parameters of workers are not sufficiently aggregated to PS. The theoretical convergence rate of EASGD is not known [26] and the communication complexity is $O(2MN \cdot \frac{1}{\tau})$.

In our evaluation, we observed that the training performance (e.g., scalability and training throughput) of EASGD is substantially better than other centralized algorithms. However, we also observed that the accuracy of EASGD is relatively low compared to those by other algorithms.

Our implementation. We implemented EASGD as described in [26]. In EASGD, the global parameters and the local parameters of each worker are updated at every τ iterations. We made both the global parameters and the local parameters of each worker are updated on the PS process at the same time when a worker sends its local parameters to the PS. Thus, the PS sends back not the global parameters but the updated local parameters to the worker.

IV. Decentralized Training

In this section, we review three decentralized training algorithms; AR-SGD, GoSGD, and AD-PSGD. In decentralized training, the parameters (or gradients) are aggregated via peer-to-peer communication. Since there is no PS, decentralized training does not have explicit global parameters. Implicitly though, the average of all workers' parameters is often considered as the global parameters in decentralized training.

A. AllReduce SGD (AR-SGD)

Parameter aggregation. AR-SGD [13] is a *synchronous* training algorithm like BSP. In each iteration, the model parameters of all workers are synchronized via *AllReduce* communication which collectively and synchronously aggregates the gradients computed by all workers and distributes the aggregated gradients to all the workers [4]. Then, each worker updates its local parameters by applying the aggregated gradients.

Accuracy and performance. AR-SGD is essentially the same as BSP in terms of the model accuracy because they both use synchronous communication to aggregate gradients from workers. Thus, the theoretical convergence rate of AR-SGD is $O(\frac{1}{\sqrt{NK}})$ same as BSP. AR-SGD also has the same performance problem as BSP (i.e., straggler problem) because of its synchronous communication. While, since the aggregation is performed via peer-to-peer communication not a separate PS, AR-SGD does not have a PS (or specific node) bottleneck problem.

Our implementation. We implemented AR-SGD standard MPICH [4] where AllReduce is implemented in two steps; Reduce-Scatter step and AllGather step. In Reduce-Scatter step, each worker partially aggregates a disjoint subset of gradients from all workers. In AllGather step, the partially aggregated gradients are distributed to all workers. As a result, all workers have the completely aggregated gradients.

B. Gossip SGD (GoSGD)

Parameter aggregation. In GoSGD [6], the model parameters of workers are aggregated by an *asymmetric* gossip algorithm [17] proposed to aggregate information in social networks. In each iteration, each worker determines whether it would communicate its parameters with the probability p, where p is a hyperparameter that controls the frequency of communication. If a worker decides to communicate, it chooses a target worker uniformly at random and sends its model parameters to the target worker. Then, it proceeds to the next training iteration without waiting for the response from the target worker (i.e., asymmetric communication). The model parameters of each worker are updated only when it receives the parameters from other workers.

Accuracy and performance. In GoSGD, the communication probability p controls the model accuracy and training performance. The smaller the p is, the less frequently model parameters of each worker are aggregated via peer-to-peer communication, thereby improving the training performance. However, this makes the local parameters of each worker slowly propagated to other workers, adversely affecting the model accuracy. We observed that GoSGD shows almost linear speedup with respect to the number of workers, but it has a model accuracy issue due to the infrequent parameter communication. We will evaluate how the probability p affects the model accuracy and training performance of GoSGD in

Section VI. The theoretical convergence rate of GoSGD is not known [6] and the communication complexity of GoSGD is $O(MN \cdot p)$ because every worker asymmetrically communicates the updated parameters with the probability p.

Our implementation. We implemented GoSGD as described in [6]. To maximize the performance, we implemented separate communication threads on background; thus, the computation and communication of each worker are executed concurrently in our implementation.

C. Asynchronous Decentralized Parallel SGD (AD-PSGD)

Parameter aggregation. In AD-PSGD [20], the model parameters of workers are aggregated via *symmetric* peer-to-peer communication among workers unlike GoSGD that adopts asymmetric communication. In each iteration, a worker (active worker) sends its updated parameters to another worker (passive worker), and has to wait for the passive worker to send its parameters back. Then, the active worker updates its parameters using the passive worker's parameter (i.e., taking the averages of their local parameters). The passive worker updates its parameters using the active worker's parameters as well.

Accuracy and performance. As explained before, each worker in AD-PSGD has to communicate its parameters with another worker *symmetrically*. That is, the communication for parameter aggregation among workers in AD-PSGD is more frequent than that in GoSGD, which helps the parameter aggregation to be more effective than GoSGD. AD-PSGD has the theoretical convergence rate of $O(1/\sqrt{K})$ [20]. Although AD-PSGD has slightly larger communication complexity, $O(MN)$, than GoSGD, we observed that the model accuracy of AD-PSGD is much higher than that of GoSGD and is comparable to those of synchronous algorithms (i.e., BSP and AR-SGD).

Our implementation. As described in [20], we implemented two separate threads for computation and communication. To maximize the performance, the communication thread runs in the background; thus, the computation and communication are executed concurrently in our implementation. AD-PSGD may cause deadlock: given three fully connected workers A, B, and C, A sends its parameters x^A to B and waits for x^B from B; B has already sent out x^B to C and waits for x^C from C; and C has sent out x^C to A and waits for x^A from A. To prevent the deadlock, [20] designs a communication network for workers to be a bipartite graph. That is, workers are split into active and passive workers, where all edges in the graph connect active workers and passive workers. Active workers are connected only to passive workers not other active workers. As described in [20], we implemented that active and passive workers in a bipartite graph, where only active workers can start communication. That is, in each iteration an active worker sends its parameters to a passive worker, and waits for the passive worker to send its parameters back. A passive worker sends its parameters to an active worker only when it receives the parameters from the active worker.

V. Optimization techniques

In this section, we review three well-known optimization techniques for distributed DNN training: parameter sharding [9], distributed wait-free backpropagation (wait-free BP) [27], and deep gradients compression (DGC) [21]. We will evaluate how these optimizations affect the seven distributed training algorithms in Section VI.

A. Parameter sharding

In a centralized training, PS aggregates the parameters (or gradients) from workers and sends back the updated parameters to the workers. Because a single PS aggregating the whole parameters may be the bottleneck of the entire training, parameter sharding is widely applied in centralized training. Parameter sharding divides the parameters and distributes them into multiple PSs to process them in parallel. Via parameter sharding, the communication and computation are distributed to multiple PSs, thus improving the training performance. We take sharding the parameters in a layer-wise way because a layer is generally represented as a single data structure (e.g., a tensor in Tensorflow) so that it can be processed sequentially. It means that the parameters in the same layer are stored in the same PS, which is the same way as Tensorflow [3]. This optimization is applicable to the centralized training algorithms (BSP, ASP, SSP, and EASGD).

B. Wait-free Backpropagation (Wait-free BP)

Wait-free BP is an optimization technique to improve the training performance by overlapping the computation and the communication. In the backpropagation phase of DNN training, the computation of gradients in layer $L-1$ can be done independently of the communication of computed gradients in layer L. Once the gradients of layer L are computed, the communication of the gradients can be done in parallel with the computation of layer $L-1$'s gradients. The more the computation and communication are overlapped, the shorter the training time is. This technique is applicable to the four distributed training algorithms that send gradients not parameters (BSP, ASP, SSP, and AR-SGD).

C. Deep Gradients Compression (DGC)

Gradient sparsification is a well-known technique to reduce the communication overhead of distributed training by compressing the size of gradients to be communicated [7], [21]. Deep gradient compression (DGC) is a variant of the gradient sparsification technique that communicates only *important gradients* in each iteration. That is, DGC sorts the computed gradients by their absolute sizes and communicates only top 0.1% gradients; thus, 99.9% of the computed gradients are not communicated. To prevent the accuracy loss of the trained model, DGC applies several techniques such as local gradient accumulation, momentum correction, local gradient clipping, momentum factor masking, and warm-up training. This technique is applicable to the distributed training algorithms that communicate gradients (BSP, ASP, SSP, and AR-SGD).

TABLE II
TOP-1 ACCURACY FOR RESNET-50 ON IMAGENET-1K.

BSP	**ASP**	**SSP**	**EASGD**	**AR-SGD**	**GoSGD**	**AD-PSGD**
0.7511 (loss)	0.7459 (-0.0052)	0.6448 (-0.1063)	0.4528 (-0.2983)	**0.7513** (loss)	0.3938 (-0.3575)	0.7411 (-0.0102)

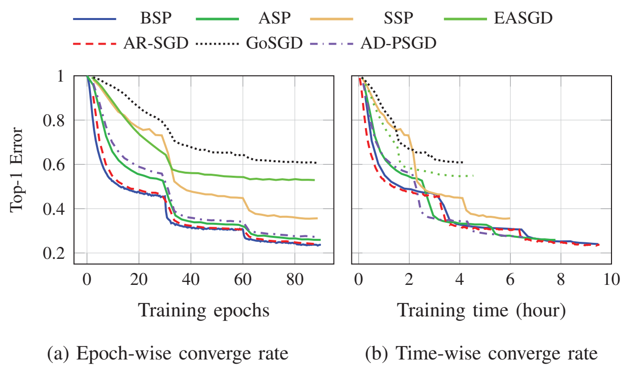


Fig. 1. Top-1 error w.r.t epochs and time for ResNet-50 on ImagetNet-1K.

VI. EVALUATION

In this section, we evaluate seven distributed training algorithms and conduct an in-depth analysis of them in the four aspects; (1) the model accuracy, (2) hyperparameter sensitivity, (3) scalability, and (4) the effectiveness of existing optimization techniques on them.

Models and datasets. We evaluate the seven algorithms with widely used CNN models, ResNet-50 [14] and VGG-16 [25]. ResNet-50 with 23M parameters is a *computation-intensive* model while VGG-16 with 138M parameters is a *communication-intensive* model. For the training datasets, we use the ImageNet-1K dataset [23] composed of about 1.2M training images and 5K test images with 1K labels.

System setting. We use TensorFlow 1.12 and MPICH 3.1.4 to implement the seven algorithms on Ubuntu 16.04. We evaluate those algorithms on the cluster with three machines. Each machine has 8 NVIDIA TITAN V GPUs and Intel Xeon CPU E5-2698 v4 with 256 GB memory, where TITAN V is capable of executing 14.90 TFLOPS and has 12GB memory. All machines are inter-connected by 10Gbps Ethernet and 56Gbps Infiniband network. We run two light-weight virtual machines (Dockers) to each of the host machines. Thus, each virtual machine is assigned with four GPUs and shares other computing resources such as main memory and network bandwidth. The cluster that we use consists of 6 (virtual) machines, where each machine has four workers (GPUs).

A. Model Accuracy

Experimental goal and setup. In this experiment, we evaluate the accuracies of the seven algorithms and their convergence rate with respect to training epochs and time. We train ResNet-50 on ImageNet-1K for 90 epochs, which is large enough for the model to converge [14], using each of seven algorithms; then, we measure the model accuracy and the errors with respect to training epochs and time.

We conduct this experiment on the cluster with 6 virtual machines (24 GPUs in total) connected by a 56Gbps network. We set the batch size for each worker as 128 to fully utilize the GPU memory. We use momentum SGD for optimization and set momentum as 0.9, weight decay factor as 0.0001, and learning rate η as $0.05 \cdot n$ based on the learning rate scaling rule [13]. We apply the learning rate gradual warm-up for the first five epochs [13], which is known to alleviate the problem of training loss fluctuation in early stage of the training, and decay the learning rate by $1/10$ at epoch 30, 60, and 80 as described in [13], [14]. Three of the evaluated algorithms – SSP, EASGD, and GoSGD – have hyperparameters. We set those parameters to be the values that are used and recommended by the authors of the algorithms [6], [16], [26]. Specifically, for SSP, we set the staleness threshold s to be 10, for EASGD we set the communication period τ to be 8, and for GoSGD, we set the gossip probability p to be 0.01.

Results and discussions. Table II shows the final accuracies of ResNet-50 models trained by the seven algorithms and Figure 1 shows top-1 test errors with respect to training epochs and time (i.e., epoch/time-wise convergence rate). We first observed that the two synchronous algorithms, BSP and AR-SGD, achieve the highest model accuracy and show the best epoch-wise convergence rate, as shown in Figure 1(a). This is because all workers have the same parameter values via their synchronous communication for the gradient aggregation, which allows the training is consistent across all workers.

Next, we noticed that three asynchronous algorithms – SSP, EASGD, and GoSGD – show much more loss in model accuracy and much lower epoch-wise convergence rate than others asynchronous algorithms (ASP and AD-PSGD), as shown in Table II and Figure 1(a). The distinct difference between the two groups is in the way how those algorithms aggregate the parameters (or gradients). In the former three algorithms (SSP, EASGD, and GoSGD), the parameters (or gradients) of workers are *intermittently* aggregated. In SSP and GoSGD, the communication for the aggregation is *asymmetric*; a worker sends its local gradients to PS or its parameters to another worker in each iteration, but receives the aggregated parameters intermittently. In EASGD, a worker communicates its parameters with PS(s) at every few iterations, and thus the local parameters of each worker are updated with the global parameters intermittently.

While, in ASP and AD-PSGD, the gradients/parameters are aggregated at every iteration as explained in Sections III and IV. This makes the variance among model parameters of workers relatively small, thereby improving the model accuracy. As shown in Table II and Figure 1(a), ASP and AD-PSGD outperform the former three algorithms and are comparable to synchronous algorithms in terms of the model accuracy and epoch-wise convergence rate.

Here, ASP shows slightly better model accuracy and epoch-wise convergence rate than AD-PSGD. The reason is that ASP

TABLE III
TEST ACCURACY OF ASYNCHRONOUS ALGORITHMS FOR RESNET-50 ON IMAGENET-1K.

	BSP	ASP	SSP		EASGD		GoSGD			AD-PSGD
# of workers	-	-	$s = 3$	$s = 10$	$\tau = 4$	$\tau = 8$	$p = 1$	$p = 0.1$	$p = 0.01$	-
4	0.7514	0.7508	0.7480	0.7462	0.7028	0.7027	0.7160	0.6892	0.6775	0.7483
8	0.7509	0.7482	0.7450	0.7412	0.6357	0.6269	0.6529	0.6173	0.5845	0.7447
16	0.7496	0.7447	0.7393	0.7147	0.5416	0.5237	0.5492	0.5135	0.4922	0.7439
24	0.7511	0.7459	0.7282	0.6448	0.4709	0.4528	0.4641	0.4475	0.3938	0.7411

aggregates the gradients from workers through PS, whereas AD-PSGD does through peer-to-peer communication. In ASP, a worker receives the latest global parameters from PS in *every* iteration, where the global parameters reflect the latest gradients computed by other workers. On the other hand, in AD-PSGD, at most $O(n)$ iterations are required to receive the newly updated parameters from other workers.

ASP and AD-PSGD show the better time-wise convergence rate than synchronous algorithms (i.e., BSP and AR-SGD) as shown in Figure 1(b). This indicates that the aggregation overhead in ASP and AD-PSGD is lower than that in BSP and AR-SGD. In BSP and AR-SGD, the overhead is not trivial because of their synchronous communication for gradient aggregation, particularly when there are some stragglers for gradient computation who fall behind other workers. While, the aggregation overhead in ASP and AD-PSGD is relatively small because they adopt asynchronous communication. Thus, ASP and AD-PSGD conduct more training iterations than BSP and AR-SGD in the same time period.

B. Hyperparameter Sensitivity

Experimental goal and setup. For the five asynchronous distributed training algorithms (ASP, SSP, EASGD, GoSGD, and AD-PSGD), the model accuracy may vary with the number of workers. Three of the algorithms (SSP, EASGD, and GoSGD) also have hyperparameters, which may also affect the model accuracy. Thus, we evaluate the accuracies of the models trained by the five algorithms by varying the number of workers and hyperparameters of the algorithms.

We train ResNet-50 with ImageNet-1K for 90 epochs, varying the number of workers (up to 24 workers) and measure the accuracies of the models trained by the algorithms. We set the batch size as 128 and the learning rate η as $0.05 \cdot n$ the same as in the previous experiment. We also apply the learning rate warm-up technique for all the experiments here. For SSP, we set the staleness as 3 and 10; for EASGD, we run the experiments with the communication period τ of 4 and 8; for GoSGD, we set the exchange probability p as 1, 0.1, and 0.01.
Results and discussions. Table III shows the results. We first confirmed that the synchronous algorithm, or BSP, maintains the model accuracy without loss as the number of workers increases. In a synchronous algorithm, we note that increasing the number of workers is equivalent to simply increasing the total batch size in each training iteration.

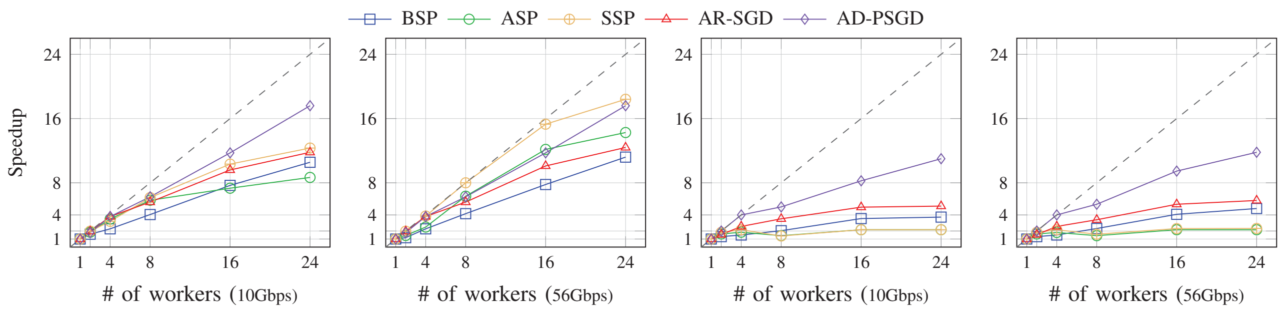
Next, we observed that the accuracy of the trained model decreases for all the asynchronous algorithms as the number of workers increases. As the number of workers increases in a cluster, the amount of parameters aggregated via asynchronous communication also increases. As a result, the disparity of the model parameters among workers becomes substantial, resulting in the accuracy loss. The accuracy decrease is particularly notable for SSP (especially, when the staleness s is 10), EASGD, and GoSGD. As we discussed in Section VI-A, workers in these algorithms aggregate their parameters (or gradients) intermittently, which makes the difference of the parameters among workers bigger than ASP and AD-PSGD.

The hyperparameters of the three algorithms (SSP, EASGD, and GoSGD) affect the model accuracy in a similar way. When the hyperparameters set for workers to aggregate more infrequently, the loss of the model trained by each of them becomes bigger; for SSP, if the staleness s gets larger ($s = 10$), workers receive the aggregated parameters from PS more infrequently, resulting in higher accuracy loss. We observed that the communication period τ in EASGD and the probability p in GoSGD affect the model accuracy in a similar way as shown in Table III.

C. Scalability

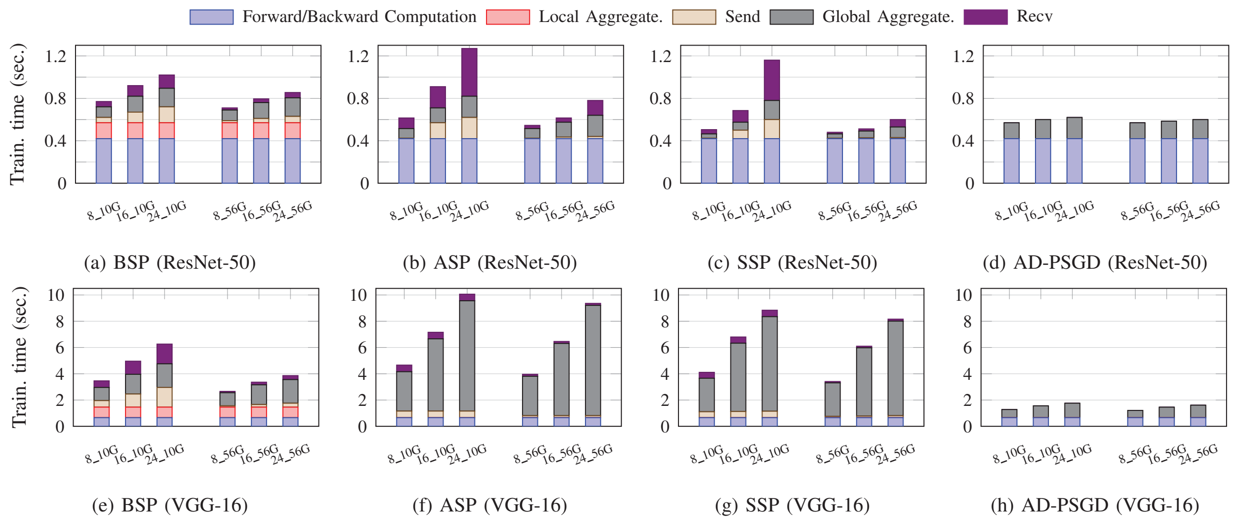
Experimental goal and setup. As the number of workers in a cluster increases, the communication overhead required for parameter aggregation inevitably increases. With the increase of the overhead, the throughput per unit time (e.g., images per sec for the ImageNet-1K dataset) of each worker decreases, which may adversely affect the scalability of algorithms. Thus, in this experiment, we evaluate the scalability of each algorithm with the number of workers. Because EASGD and GoSGD incur a substantial model accuracy loss, we exclude the two and evaluate only the rest algorithms – BSP, ASP, SSP, AR-SGD, and AD-PSGD.

To evaluate the scalability of the algorithms, we train ResNet-50, computation-intensive, and VGG-16, communication-intensive, with 1, 2, 4, 8, 16, and 24 workers (GPUs) on 10Gbps and 56Gbps networks, and measure the throughput per unit time of workers in each algorithm. The training with 1 to 4 workers is done on a virtual machine. We set the batch size for ResNet-50 and VGG-16 as 128 and 96, respectively. For the optimization techniques, we test with the two techniques that do not affect the model accuracy – parameter sharding and wait-free BP – to each algorithm. Note that the baseline for all the evaluation is the throughput of a single worker.



(a) ResNet-50 (b) VGG-16

Fig. 2. Scalability for ResNet-50 and VGG-16 on ImageNet-1K with the increasing number of workers.



(a) BSP (ResNet-50) (b) ASP (ResNet-50) (c) SSP (ResNet-50) (d) AD-PSGD (ResNet-50)

(e) BSP (VGG-16) (f) ASP (VGG-16) (g) SSP (VGG-16) (h) AD-PSGD (VGG-16)

Fig. 3. Breakdown of training time for ResNet-50 and VGG-16 on 10Gbps and 56Gbps networks.

Results and discussions. Figure 2 shows the scalability of the five algorithms. First, we examine the evaluation result for ResNet-50. We observed that the training performance of BSP and AR-SGD increases steadily as the number of workers increases, but does not improve much even when we increase the network bandwidth from 10Gbps to 56Gbps. To understand the reason why the network bandwidth has little impact on their performance, let us examine the breakdown of a worker's execution times for BSP shown in Figure 3. The breakdown for AR-SGD is not shown, but it is very similar to that of BSP. We can first observe that more than half of the execution time is spent on the gradient aggregation in the training of BSP with 24 workers. Because the local aggregation and global aggregation are not affected by the network bandwidth, the performance improvement was insignificant with the higher network bandwidth.

For BSP, the majority (up to 80%) of the local and global aggregations are spent on waiting for other workers. For the local aggregation (red bar in Figure 3), a worker needs to wait for three other workers in a same machine to finish their computations. Although the computation time is believed to have low variance in homogeneous clusters, we found out that the variance in the computation is nontrivial – the difference between fastest and slowest workers is as much as 5% of the computation time. For the global aggregation (gray bar), the PSs need to wait for all the gradients from the workers; this waiting time takes up to 70% of the global aggregation time. The actual aggregation time is only around 30%.

Now, we examine the results for ASP and SSP. As shown in Figure 2(a), the scalability of these two algorithms is shown much better with a 56Gbps network than with a 10Gbps network. For these two algorithms, the communication time takes up more than half of the total execution time (See Figure 3(b) and (c)); thus, increasing the network bandwidth substantially improves the total training performance. ASP and SSP have high global aggregation time because PS aggregates workers' gradients in the order with which they arrive; thus, for 8 workers, for example, a worker may need to wait for the remaining seven workers' gradients to be aggregated in the worst case. We also observed that the scalability of ASP (asynchronous one) is worse than even synchronous algorithms (BSP and AR-SGD) on a 10Gbps network. This means that asynchronous communication causes to make PS the training bottleneck if the network bandwidth is limited.

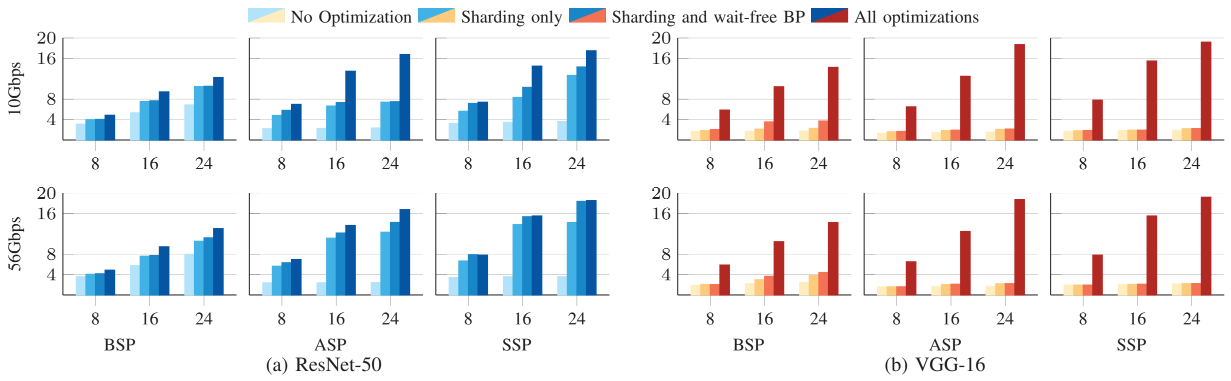


Fig. 4. Training throughput of centralized algorithms with three optimization techniques.

AD-PSGD demonstrates good scalability for ResNet-50 even though its total communication volume is as large as ASP and larger than all the other algorithms. This is because the computation and communication in AD-PSGD are executed independently with each other. That is, while the parameter communication for the current iteration runs in the background, the forward and backward computations of the next iteration are executed at the same time. Besides, the communication is less bursty in AD-PSGD because the communication is distributed into multiple workers not a specific worker (e.g., PS) unlike centralized algorithms, which helps utilize the network bandwidth better.

Next, let us examine the result for VGG-16, which has different characteristics from ResNet-50. The scalabilities of all five algorithms for VGG-16 are not as good as those for ResNet-50 as shown in Figure 2(b). This is primarily because the parameter size of VGG-16 is much larger than that of ResNet-50, thus, incurring more communication overhead. The size of the parameters in each layer is significantly skewed in VGG-16, where the size of the first fully connected layer is particularly large (about 75% of total parameters). Because we apply layer-wise sharding where the parameters in a layer are assigned to a same PS; thus, the communication and the aggregation of the parameters in that layer are the bottleneck of the training, as shown in Figure 3(e-h).

We observe that the decentralized algorithms show better scalability than the centralized ones – for instance, compare ASP and SSP with AR-SGD and AD-PSGD in Figure 2(b). In decentralized algorithms, the communication for parameter aggregation is performed in a distributed manner, which makes them more scalable. Parameter aggregation overhead (including the waiting overhead) is more significant than the communication overhead for VGG-16 as shown in Figure 3. This evaluation result shows that fine-grained sharding for parallel parameter aggregation is necessary for large DNN models such as VGG-16.

D. *Effects of Optimizations*

Experimental goal and setup. In this experiment, we evaluate the effects of the three optimization techniques on the distributed training algorithms. We train ResNet-50 and VGG-16 on ImageNet-1K with cumulatively applying three optimizations: parameter sharding, wait-free BP, and DGC. Then, we measure the training throughput of each algorithm with 8, 16, 24 workers. DGC is an approximate optimization, unlike the other two, thus it may have negative (or positive) effect on the model accuracy. Thus, we also evaluate the effect of DGC on the model accuracy for each algorithm. Before the experiment, we empirically found the optimal ratio of PSs to workers with profiling in the following way. We tested three different ratios of PSs to workers in a virtual machine (a single PS (1:4), two PSs (2:4), and four PSs (4:4)), selected the optimal ratio, and used it for the following experiments. We conducted this experiment on the cluster with 10Gbps and 56Gbps networks. Note that the baseline for all the evaluation is the throughput of a single worker.

Results and discussions. Figure 4 shows the training throughput of the algorithms with the three optimizations are applied cumulatively for 8, 16, and 24 workers. First, let us examine the effect of parameter sharding. When we compare the evaluation results for ResNet-50 and VGG-16, the optimization has better impact for ResNet-50. Because we applied layer-wise sharding, VGG-16 having high variance in the parameter sizes in each layer cannot fully take advantage of the parameter sharding. This optimization is more effective for ASP and SSP than BSP. This is because the local aggregation in BSP reduces the portion for improvement by parameter sharding.

Next, let us examine the effect of wait-free BP. Wait-free BP appears less effective than it is reported [27] in Figure 4. It improves the training throughput by overlapping the computation of gradients with the communication of the computed gradients. Recently, the computing power of GPUs has been substantially improved; thus, the overlapping portion of the communication and computation becomes smaller, making the optimization less effective.

Finally, let us evaluate the effect of deep gradient compression (DGC). The main objective of DGC is to reduce the communication overhead by communicating important gradients only. Thus, if the communication dominates the computation in the training, DGC may be very effective. This is why DGC is more effective in the training for ResNet-50 than VGG-16 and on 10Gbps than 56Gbps. As shown in

TABLE IV
EFFECT OF DGC ON MODEL ACCURACY FOR RESNET-50 ON IMAGENET-1K.

	BSP	ASP	SSP ($s = 3$)	SSP ($s = 10$)
Without DGC	0.7511	0.7459	0.7282	0.6448
With DGC	0.7505	0.7440	0.7295	0.6542

Figure 4, the effect of DGC is larger in ASP and SSP than BSP. Because in BSP the local aggregation already reduces the communication overhead much, DGC has much smaller room for the improvement. In ASP and SSP, the effect of DGC is significant, especially for VGG-16 in a network with limited bandwidth (10Gbps). Note that when DGC is applied, the two algorithms, ASP and SSP, scale very well as the number of workers increases.

In order to evaluate the effect of DGC on the model accuracy, we also measure the accuracy of the model trained by each algorithm when DGC is applied. Table IV shows the top-1 accuracies of the algorithms with and without DGC. We observed that the accuracies of the models trained with DGC are comparable to or even better than those without DGC. It indicates that DGC has no negative effect on the model accuracy of each algorithm and at the same time it reduces the communication overhead significantly.

VII. CONCLUSIONS

In this paper, we evaluated seven distributed training algorithms (BSP, ASP, SSP, EASGD, AR-SGD, GoSGD, and AD-PSGD) in terms of the model accuracy and training performance. For the evaluation, we conducted extensive experiments using two CNN models in various cluster settings with various optimization techniques (parameter sharding, wait-free backpropagation, and deep gradient compression). Through our in-depth analysis of the evaluation results, we reported several interesting findings. For example, some of the distributed training algorithms have large negative impact on the model accuracy if they adopt an asymmetric and intermittent parameter aggregation to reduce the communication overhead required for aggregation. The communication overhead of some centralized asynchronous algorithms (especially, ASP and SSP) is much higher than we expected in a cluster with limited network bandwidth because of the PS bottleneck problem; they show training performance worse than even synchronous training algorithms (i.e., BSP and AR-SGD). Wait-free backpropagation is less effective than it is reported as GPUs become more powerful these days. We believe that our findings can be useful at industry in applying distributed training algorithms and also at academia in designing new algorithms for distributed DNN training.

VIII. ACKNOWLEDGEMENT

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1901-03.

REFERENCES

[1] Distributed machine learning toolkit. http://www.dmtk.io/.

[2] Distributed training in mxnet. https://mxnet.incubator.apache.org/api/faq/distributed_training.

[3] Distributed training with tensorflow. https://www.tensorflow.org/guide/distributed_training.

[4] MPICH: High Performance Portable MPI. https://www.mpich.org/.

[5] Writing Distributed Application with with PyTorch. https://pytorch.org/tutorials/intermediate/dist_tuto.html.

[6] M. Blot et al. Gosgd: Distributed optimization for deep learning with gossip exchange. *arXiv preprint arXiv:1804.01852*, 2018.

[7] C.-Y. Chen et al. Adacomp: Adaptive residual gradient compression for data-parallel distributed training. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 2827–2835, 2018.

[8] H. Cui et al. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the European Conference on Computer Systems (EUROSYS)*, pages 1–16, 2016.

[9] J. Dean et al. Large scale distributed deep networks. In *Proceedings of the Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[10] J. Devlin et al. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[11] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

[12] S. Ghadimi et al. Mini-batch stochastic approximation methods for nonconvex stochastic composite optimization. *Mathematical Programming*, 155(1-2):267–305, 2016.

[13] 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.

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

[15] L.-Y. Ho et al. Adaptive communication for distributed deep learning on commodity gpu cluster. In *Proceedings of the International Symposium on Cluster, Cloud and Grid Computing*, pages 283–290. IEEE, 2018.

[16] Q. Ho et al. More effective distributed ml via a stale synchronous parallel parameter server. In *Proceedings of the Advances in Neural Information Processing Systems*, pages 1223–1231, 2013.

[17] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 482–491, 2003.

[18] M. Li et al. Scaling distributed machine learning with the parameter server. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 583–598, 2014.

[19] X. Lian et al. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *Proceedings of the Advances in Neural Information Processing Systems*, pages 5330–5340, 2017.

[20] X. Lian et al. Asynchronous decentralized parallel stochastic gradient descent. In *Proceedings of the international Conference on Machine Learning (ICML)*, pages 3049–3058, 2018.

[21] Y. Lin et al. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

[22] D. Narayanan et al. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, 2019.

[23] R. Olga et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[24] B. Recht et al. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, pages 693–701, 2011.

[25] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[26] S. Zhang, A. E. Choromanska, and Y. LeCun. Deep learning with elastic averaging sgd. In *Proceedings of the Advances in Neural Information Processing Systems*, pages 685–693, 2015.

[27] H. Zhang et al. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 181–193, 2017.