Distributed and Asynchronous Stochastic Gradient Descent with Variance Reduction

Yuewei Ming[a,*], Yawei Zhao[a], Chengkun Wu[a], Kuan Li[a], Jianping Yin[b]

[a] *College of Computer, National University of Defense Technology, Changsha 410073, China*
[b] *State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, 410073, China*

Abstract

Stochastic Gradient Descent (SGD) with variance reduction techniques has been proved powerful to train the parameters of various machine learning models. However, it cannot support the distributed systems trivially due to the intrinsic design. Although conventional studies such as PetuumSGD perform well for distributed machine learning tasks, they mainly focus on the optimization of the communication protocol, which does not exploit the potential benefits of a specific machine learning algorithm. In the paper, we analyze the asynchronous communication protocol in PetuumSGD, and propose a distributed version of variance reduced SGD named *DisSVRG*. DisSVRG adopts the variance reduction technique to update the parameters in a model. After that, those newly learned parameters across nodes are shared by using the asynchronous communication protocol. Besides, we accelerate DisSVRG by using the adaptive learning rate with an acceleration factor. Additionally, an adaptive sampling strategy is proposed in DisSVRG. The proposed methods greatly reduces the wait time during the iterations, and accelerates the convergence of DisSVRG significantly. Extensive empirical studies verify that DisSVRG converges faster than the state-of-the-art variants of SGD, and gains almost linear speedup in a cluster.

Keywords: Stochastic gradient descent, Variance reduction, Asynchronous

* Corresponding author
Email address: ywming@nudt.edu.cn (Yuewei Ming)

1. Introduction

Machine learning based applications such as image recognition [1], speech recognition [2] and text processing [3] proliferate in the era of Big Data. Those underlying machine learning models are usually complex and big with a large number of parameters which can be trained or learned from a large amount of training data. For example, it is possible to train a large scale deep neural network which consists of millions or even billions of parameters by feeding it with terabytes of training data. Furthermore, it is worth noting that most of the machine learning algorithms are iteratively convergent, which means those algorithms need many rounds of iterative calculations to update the parameters of their models. Considering the complexity of the underlying model, the huge size of the training data and the massive amount of computation, an efficient training method is vitally important for performing a large scale machine learning task.

Many machine learning algorithms can be described by the optimization problem like (1),

$$\min_{\omega} f(\omega), \quad f(\omega) = \frac{1}{n}\sum_{i=1}^{n} f_i(\omega) + R(\omega). \tag{1}$$

Here, $f(\omega)$ is generally called the loss function. $f_i(\omega)$ with $1 \leq i \leq n$ represents the objective loss corresponding to the ith instances. ω represents the parameters of a model, that is, the parameters needed to be updated during the iterations. n means the size of training data. $R(\omega)$ represents the regularization item which is used to avoid overfitting. The regularization item represents the prior knowledge about the problem. For example, when we want to obtain a sparse solution, that is, the optimal parameters contain many zeros, $R(\omega)$ is usually formulated as the L_1 norm, namely, $R(\omega) = \|\omega\|_1$. Besides, ridge regression modes use the L_2 norm as the regularization item, that is, $R(\omega) = \|\omega\|_2$. Some other regularizations can be used in the optimization objective [4, 5, 6], but it is out of the scope of the paper. We recommend readers to read those references for more details.

The loss function $f(\omega)$ can be minimized by updating the parameters iteratively, which is called the learning process. Conventionally, the gradient descent method is used to compute the global average gradient, i.e., $\nabla f(\omega_{t-1})$, and then uses it to update the parameters during an iteration. Here, t represents the tth iteration. Since $\nabla f(\omega_{t-1})$ needs n derivations, which are time-consuming, the gradient descent method is not practical for a large scale machine learning task . An alternative approach is Stochastic Gradient Descent (SGD) and its variants. SGD randomly samples an instance from the training data, and then use it to compute the local gradient, i.e., $\nabla f_i(\omega_{t-1})$, instead of the global average gradient. The parameters are thus updated by using the local gradient. Since $\nabla f_i(\omega_{t-1})$ merely needs one derivation for the local gradient during an iteration, it is efficient for the large scale machine learning tasks. However, the variance exists between the local gradient $\nabla f_i(\omega_{t-1})$ and the global average gradient $\nabla f(\omega_{t-1})$, which is denoted by the stochastic noise in the paper equivalently. The variance slows the convergence of the loss function i.e., $f(\omega)$, for a machine learning algorithm. Specifically, when the parameters are close to the optimum, it is difficult to decrease the loss function due to the variance. Conventionally, the variance is reduced by using a decaying learning rate to update the parameters. That is, the value of the learning rate is decreased when the iteration proceeds. Although the variance can be reduced by the decaying learning rate, the small learning rate unavoidably impairs the convergence performance.

Recently, the SGD and its variants have been widely used to train the parameters of a model in distributed memory systems [7, 8, 9]. Those versions of SGD generally train and update parameters in a parameter server system by using a cluster. The nodes in the parameter server system are categorized into servers and workers. First the workers pull the parameters from the servers, Second, the underlying calculations of the gradients are conducted by workers. Those updates will be then pushed to servers, and be aggregated on servers for updating the global parameters. Finally, those newly learned global parameters will be shared with work-

ers. Since there exists much communication between workers and servers, almost all the distributed versions of SGD like PetuumSGD focus on the optimization of communication [9]. In specific, PetuumSGD has proposed the asynchronous communication protocol denoted by Staleness Synchronous Protocol (SSP) to conduct communication across nodes. Since SSP is designed for the general iteratively convergent machine learning algorithms, it does not exploit the potential benefits of SGD to accelerate the iterative calculations. For example, PetuumSGD updates the parameters by using a decaying learning rate. The learning rate in PetuumSGD in the current iteration denoted by η will become 0.95η in the next iteration. When the parameters are close to the optimum, the loss function is difficult to be decreased due to the extremely small learning rate. In a nutshell, even though PetuumSGD adopts SSP to optimize the communication between workers and servers, the decay learning rate slows its convergence.

Meanwhile, a new technique of the variance reduction is proposed to speed up the convergence of SGD [10, 11, 12]. Such variance reduction technique reduces the variance of SGD, and keeps SGD converging at a constant rate. However, those versions of SGD are designed to be used in one node instead of a cluster. When the amount of parameters or the size of training data is extremely huge so that they cannot be stored in a node, the underlying variance reduction technique will not work. For instance, a deep network may have billions or even trillions of parameters, which cannot be stored in a node. Therefore, such versions of SGD are incapable of performing the train of the paramters for a large scale machine learning task, or handling a large amount of training data.

In this paper, we design a distributed and asynchronous version of variance reduced SGD denoted by DisSVRG for large scale machine learning tasks. It is worth noting that DisSVRG adopts the asynchronous communication protocol, i.e., Staleness Synchronous Protocol (SSP). In order to obtain a fast convergence, DisSVRG is accelerated by using a learning rate with an acceleration factor. It is unavoidable that the fast workers will spend much time on waiting for the slow workers when

performing iterative calculations in a cluster, which is also known as the "straggler problem". The straggler problem wastes much time for the fast workers, thus leads to the slow convergence of a machine learning algorithm. In order to reduce the wait time, we propose an adaptive sampling strategy to alleviate the straggler problem. Specifically, we dynamically adjust the random sampling strategy during the iterations. When the worker is faster than other workers, it will sample more instances for the next iteration, which will take the faster worker more time to compute the local gradient. Thus, the slow workers have chance to catch up with the faster works, and the wait time is reduced significantly. Finally, we conduct empirical studies on a High Performance Computing (HPC) cluster. The performance evaluation verifies that DisSVRG outperforms the state-of-the-art version of SGD, and obtains approximately linear speedup in the cluster.

The rest of this paper is organized as follows. Section 2 outlines the related work. Section 3 presents the preliminaries of our method. Section 4 illustrates the details of DisSVRG. Section 5 highlights the optimization of DisSVRG. Section 6 discusses the major difference between our work and the previous studies. Section 7 shows the performance evaluation. Section 8 concludes the paper.

2. Related work

With the proliferation of data, a complex and big model can be learned by feeding it with a huge size of training data. An approach for such a large scale learning is to use a cluster to train the parameters of the underlying model [7, 8, 9]. Dean et al. propose a version of asynchronous and distributed SGD denoted by *DownpourSGD* in a parameter server system. DownpourSGD uses fully asynchronous communication protocol to conduct communication across nodes, which cannot guarantee the convergence. To increase the robustness of DownpourSGD, the Adagrad adaptive learning rate procedure is adopted [13]. However, the adopted learning rate is decayed with the iterations, and thus leads to the slow convergence. Besides, Li et al. and Xing et al. have proposed an implementation of the param-

eter server system, respectively. The similar asynchronous communication protocol denoted by SSP is adopted in both of their systems to share the updates of the parameters across nodes. SSP has been proved powerful in both theory and practice. However, SGD in [8] uses a constant learning rate without variance reduction technique, leading to slow convergence due to the variance. SGD in [9], i.e., PetuumSGD, adopts a decaying learning rate to reduce the variance, giving rise to slow convergence when the learning rate becomes small. Our version of the asynchronous and distributed SGD, i.e., DisSVRG, adopts SSP to implement the communication across nodes, and uses the variance reduction technique to reduce variance as well. Recently, Zhang et al. have proposed a new distributed variant of SGD denoted by SSGD in the paper [14]. SSGD is designed by combining the delayed proximal gradient and the stochastic variance reduced gradient. Although SSGD outperforms other previous variants of SGD because of the variance reduction technique, our proposed method DisSVRG has an advantage of the convergence performance over it. Additionally, there are some other impressive researches about the distributed machine learning in a cluster. For example, Aaron et al. focus on the straggler problem in the distributed settings, and propose FlexRR to solve the problem for iteratively convergent machine learning algorithms [15]. Li et al. propose an efficient mini-batch training mechanism to accelerate SGD in a cluster [16].

The variance reduced SGD denoted by *SVRG* is adopted in [10], which is effective to reduce the variance of SGD. However, SVRG is a serial version, and designed to be run on a single node. Thus, it is not suitable for a large scale machine learning task . Asynchronous and parallel versions of SVRG partially solve this problem [11, 12, 17, 18, 19]. Such SVRG versions use the lock-free method to update the parameters in parallel for multiple learning threads in a node. However, the design of such work targets at a multicore system on a single node. When the size of training data or the number of parameters is huge so that they can not be stored in one node, SVRG and those variants fail immediately. Our proposed

variance reduced SGD is designed for the large scale machine learning tasks in a cluster. We can partition the data into multiple blocks, and allocate them to multiple worker machines, which is suitable to accelerate SGD in a cluster. More recently, there are many new variants of the variance reduced SGD such as SAGA [20], S2GD [21], SVRG++ [22], Prox-SVRG [23]. Those great researches have proposed advanced variance reduction techniques, which can be used to improve our method. However, it is out of the scope of the paper, and we leave it as the future work.

3. Preliminaries

In this section, we present the preliminaries including the parameter server system and the data parallelism, and the variance reduced SGD.

3.1. Parameter server and data parallelism

In data-parallel machine learning, the data set D is partitioned into P blocks. The blocks of data are assigned to the worker machines which are indexed by $p = 1, \ldots, P$. We denote the pth data block by D_p. The data parallelism updates the model parameters as follows:

$$\omega^s = G\left(\omega^{s-1}, \sum_{p=1}^{P} \Delta(\omega^{s-1}, D_p)\right). \tag{2}$$

Here, $\Delta(\cdot)$ means the update of the parameters which is obtained according to the data partition D_p on a worker. In specific, the worker needs to pull the initial parameters from the model server. After that, it computes the update of the parameters, i.e. $\Delta(\omega^{s-1}, D_p)$. $G(\cdot)$ represents the aggregation of the updates of the parameters on a server. Specifically, the server collects all the updates from the workers, and then conduct the aggregation [**?**] [**?**]. Taking SGD as an example, the update rule can be

$$\omega^s = \omega^{s-1} + \sum_{p=1}^{P} \eta \Delta(\omega^{s-1}, D_p), \tag{3}$$

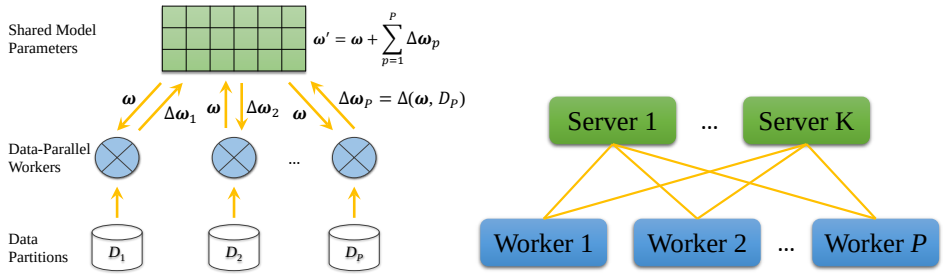


Figure 1: Illustration of data parallelism and Parameter Server topology

where $G(\cdot)$ is initialized as an additive operation, and $\Delta(\omega^{s-1}, D_p)$ is usually initialized as the product of the learning rate η and the gradient according to the data partition D_p. As shown in Figure 3.1, servers and workers interact via a bipartite topology. The model parameters ω can be divided and stored on multiple servers and thus not limited by a single machine's memory in a parameter server system. Every worker pulls the parameters from the server, and then obtain the update of the parameters. Finally, those updates are pushed to the servers, and are aggregated on those servers. The servers can collaborate with the workers to utilize CPUs on all machines when we conduct a large scale machine learning task [**?**] [**?**].

3.2. Variance reduced SGD

The variance reduced SGD uses a variance reduced gradient to reduce the stochastic noise during the update of the parameter. The variance reduced gradient is formulated as

$$v_t = \nabla f_{i_t} - \nabla \tilde{f}_{i_t} + \nabla \tilde{f}. \tag{4}$$

Here, i_t represents the index of an instance which is picked at the tth iteration. ∇f_{i_t} represents the stochastic gradient, $\nabla \tilde{f}_{i_t}$ represents the noise reducer, and $\nabla \tilde{f}$ represents the stale full gradient. First, ∇f_{i_t} usually leads to much stochastic noise because of it is different from the full gradient. The stochastic noise is denoted by variance in the paper. Compared to the gradient descent, the convergence of the SGD is a victim of the variance, and usually converges slowly. To overcome the weakness of the SGD, the variance gradient uses $\nabla \tilde{f}_{i_t}$ and $\nabla \tilde{f}$ to reduce the

variance due to the following property:

$$\mathbb{E}v_t = \mathbb{E}(\nabla f_{i_t} - \nabla \tilde{f}_{i_t} + \nabla \tilde{f}) = \nabla f. \tag{5}$$

That is to say, the variance reduced gradient is equivalent to the full gradient at every iteration in expectation. It is worthy noting that $\nabla \tilde{f}$ is a stale full gradient, which is updated at the start of an epoch and kept fixed during the iterations in an epoch. Therefore, the variance reduced gradient reduces the stochastic noise significantly, but leads to less computational cost. Extensive empirical studies show that the variance reduced gradient leads to the comparable computational cost of SGD, but obtains the equivalent convergence performance of gradient descent.

4. System implementation

In the section, we present the details of the system implementation including the algorithm and the distributed mechnism.

4.1. Overview

Our distributed and asynchronous SGD denoted by DisSVRG is presented in Algorithm 1. DisSVRG is organized by the epochs of iterations (the outer *for* loop at Line 2). In every epoch, the instances are picked randomly (the inner *for* loop at Line 6), and the update rule of the parameters (Lines 8-9) uses a variance reduced gradient. DisSVRG is launched by the servers, and all the workers will be informed by message passing. Once a worker receives the message from a server, it pulls a copy of the global parameters from a server, and begins conducting the calculations during the epoch. The random sampling strategy is conducted by the workers. When a worker randomly samples an instance from the training data, it computes the variance reduced gradient (Line 8), and updates the local parameters with the local gradient (Line 9). When the local parameters have been updated, the newly learned parameters will be sent to a server. The inter-node communication is conducted by the asynchronous communication protocol, i.e., SSP. The server

Algorithm 1 DisSVRG

1: Initialize $\tilde{\omega}^0$. \\ Pull the global parameters by all workers from the servers.
2: **for** $s = 1, 2, ...$ **do** \\ Asynchronously update the parameters by all workers.
3: $\tilde{\omega} = \tilde{\omega}^{s-1}$.
4: $\tilde{f}(\tilde{\omega}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\tilde{\omega})$.
5: $\omega_0 = \tilde{\omega}$.
6: **for** $t = 1, 2, ..., m$ **do**
7: randomly sample $i_t \in \{0, 1, 2, ..., m\}$.
8: $v_t = \nabla f_{i_t}(\omega_{t-1}) - \nabla f_{i_t}(\tilde{\omega}) + \tilde{f}(\tilde{\omega})$. \\Variance reduced gradient.
9: $\omega_t = \omega_{t-1} - \eta v_t$. \\Update the parameters with variance reduction gradient.
10: $\tilde{\omega}^s = \omega_m$. \\Push the newly learned parameters to the servers, and aggregate them with the global parameters.

receives these learned parameters and aggregates them. The aggregated parameters are the latest global parameters which will be sent to workers for the next iteration. The details of communication across nodes and aggregation of parameters will be demonstrated in Section 4.

4.2. Distributed implementation

DisSVRG is designed for the distributed memory systems where the nodes can be categorized into workers and servers. Every worker caches a copy of the parameters which is called the *local parameters*; while all the servers maintain one copy of the parameters which are called the *global parameters*. The global parameters will be pulled by a worker and be used as the initial parameters for an iteration. Meanwhile, such initial parameters will replace the stale local parameters on the worker and become its new local parameters.

Server: The servers control the iterations by using the asynchronous communication protocol, i.e., SSP. Since the runtime environment of nodes in a cluster varies a lot, the time overhead of an epoch for different workers varies. Therefore, there

Algorithm 2 Server

1: Initialize $\tilde{\omega}^0$.
2: **while** true **do**
3: **if** receive a pull request from the worker p. **then**
4: $e_p = p.epoch$.
5: **if** all the workers have finished $(e_p - \tau)$th epoch. **then**
6: send a copy of the global parameters to the worker p.
7: **if** receive a push request from the worker p. **then**
8: receive the newly learned parameters from the worker p.
9: aggregate the newly learned parameters with the global parameters on the server.

are fast and slow workers which conduct an epoch fast and slow respectively. It is worth noting that DisSVRG may not converge if all the workers update parameters in a fully asynchronous way. Therefore, we set a delay bound, i.e., τ. The delay τ is used to synchronize all the workers. For instance, when the fastest worker finishes the tth iteration, and the slowest worker does not finish the $(t-\tau)$th iteration, the fastest worker will be forced to stop and wait for the slowest one. In specific, the fastest worker can not pull a copy of the global parameters from the servers, and thus has to wait for the slowest worker. Until the slowest worker finishes the $(t-\tau)$th iteration, the fastest worker will re-start to conduct the iterations. When a server receives the newly learned parameters from a worker, it will aggregate them with the global parameters on the server. After that, the latest parameters on the server will be pulled by the worker for the next iteration. The details are illustrated in Algorithm 2.

Worker: Workers in a cluster conduct machine learning tasks in asynchronous way. They pull the parameters from the servers by message passing. If a copy of the global parameters is pulled to the workers, those workers begin iterative calculations. During an epoch, workers first randomly pick an instance from the training data, then use the instance to compute the gradient. All the workers are

Algorithm 3 Worker

while true **do**
  send a pull reworkerquest to a server.
  while true **do**
    if receive a copy of the global parameters from the server. **then**
      cache it as the new local parameters, i.e., $\tilde{\omega}$.
      $\nabla \tilde{f}(\tilde{\omega}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\tilde{\omega})$.
      for $t = 0, 1, 2, ..., m$ **do**
        randomly sample a non-negative number i with $i \in \{0, 1, 2, ..., n\}$.
        $v_t = \nabla f_{i_t}(\omega_{t-1}) - \nabla f_{i_t}(\tilde{\omega}) + \nabla \tilde{f}(\tilde{\omega})$.
        $\omega_t = \omega_{t-1} - \eta v_t$.
      send ω_m to a server.

independent with peers when conducting iterative calculations. When an epoch is finished, a worker has learned the new parameters, and will send those newly learned parameters to a server. It is the servers that control when to synchronize among the workers.

Aggregation: When the workers push their newly learned parameters to the servers, those parameters will be aggregated with the global parameters on the servers. The average between the newly learned parameters and the global parameters will be identified as the latest global parameters, and will wait to be pushed to all the workers for the next iteration.

Update rule: As illustrated in Algorithm 3, DisSVRG is significantly different from the standard SGD because of the variance reduction technique. In the standard SGD, the update rule is shown as follows.

$$v_t = \nabla f_{i_t}(\omega_{t-1}) \tag{6}$$

v_t in the standard SGD is not the global gradient, which leads to variance, and slows the convergence of the loss function unavoidably. Instead, the variance

reduction technique is used in DisSVRG effectively reduce the variance [10]. Considering the ith round of the iterations, since i_t is randomly picked, it holds that

$$\mathbb{E}\nabla f_{i_t}(\omega_{t-1}) = \nabla f(\omega_{t-1}), \tag{7}$$

and

$$\mathbb{E}(-\nabla f_{i_t}(\tilde{\omega}) + \nabla \tilde{f}(\tilde{\omega})) = -\nabla \tilde{f}(\tilde{\omega}) + \nabla \tilde{f}(\tilde{\omega}) = 0. \tag{8}$$

Therefore,

$$\mathbb{E}(v_t) = \mathbb{E}(\nabla f_{i_t}(\omega_{t-1}) - \nabla f_{i_t}(\tilde{\omega}) + \tilde{f}(\omega)) = \mathbb{E}(\nabla f_{i_t}(\omega_{t-1})) = \frac{1}{n}\nabla f(\omega_{t-1}). \tag{9}$$

It is obvious that the variance of DisSVRG is reduced as expected. Unlike PetuumSGD and DownpourSGD, DisSVRG can converge fast without decaying the learning rate during the iterations. Benefiting from the variance reduction technique, DisSVRG can converge at a constant rate.

5. Optimization of DisSVRG

In the section, we optimize the proposed method DisSVRG by adopting an adaptive learning rate and sampling strategy.

5.1. Learning rate with an acceleration factor

Generally, variance reduction technique accelerates machine learning algorithms by using a constant learning rate. Even though the constant learning rate performs well for L-smooth and γ-strongly convex objection function, some intrinsic properties can be exploited to accelerate the convergence of the machine learning algorithms.

We introduce an acceleration factor σ with $\sigma = \| \nabla f_i(\omega) \|$ to the learning rate of DisSVRG. That is,

$$\eta = \eta_0 + \sigma, \qquad \sigma = \| \nabla f_i(\omega) \| . \tag{10}$$

The learning rate of DisSVRG contains two ingredients: the constant and the acceleration factor. The constant part of the learning rate get the parameters out of the local optimum; while the acceleration factor speedup the convergence of DisSVRG. This setting of the acceleration factor is reasonable for the following reasons. First, the acceleration factor will become large when the parameters are far from the optimum. If so, DisSVRG will converge fast accordingly. Second, the acceleration factor will not become so large that DisSVRG do not converge. Considering that the convex function f_i in the machine learning model is L-smooth, ∇f_i will not be changed out of a range for an iteration. That is, $\| \nabla f_i \| < C$. Here, C is a non-negative constant. Third, when the parameters are close to the global optimum, the acceleration factor σ will become close to zero. DisSVRG thus almost converges to the global optimum at a constant rate just like the original design in [10].

As illustrated in Fig. 2, we use the acceleration factor to conduct the linear regression tasks. Here, the evaluation test is conducted on a node instead of a cluster. The dataset is *YearPredictionMSD* which is the largest dataset we can find to conduct linear regression tasks. Other settings of the evaluation are presented in Section 7. We can get two important observations from Fig. 2. First, the learning rate with an acceleration factor makes the underlying machine learning algorithm converge faster than the constant learning rate without the acceleration factor. Second, the benefits become significant with a small basic learning rate. Shortly, the acceleration factor gives rise to obvious benefits to accelerate the convergence of a machine learning algorithm.

5.2. Adaptive sampling strategy

Since DisSVRG needs the sampling strategy to update the parameters during an epoch, it is important to identify how many random updates in an epoch are appropriate. As illustrated in Algorithm 1, m represents the number of updates for an epoch. First, m cannot be given an extremely large number, which takes much time to update the local parameters during an epoch on a worker. Second, m

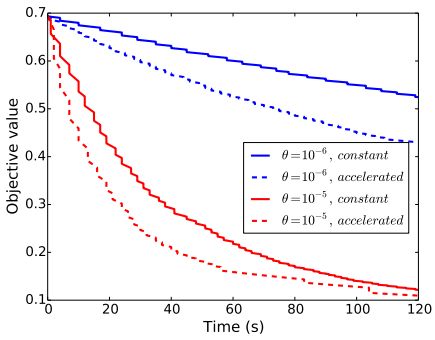
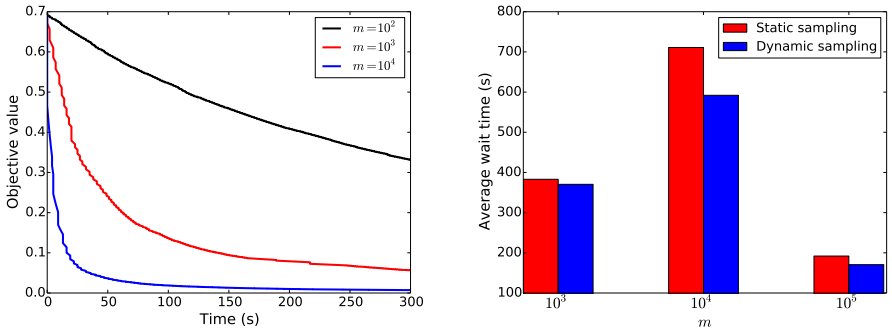


Figure 2: The learning rate with the acceleration factor makes DisSVRG converge fast significantly.

cannot be set a small value straightly. If so, DisSVRG has to conduct many epochs to reduce the value of the loss function. Considering that the average gradient of the loss function is needed to conduct an epoch, a small m means much computation of the average gradients, which is time-consuming. Additionally, it is worth noting that the workers in a cluster perform asynchronously due to the diversity of the runtime environment or the hardware in the heterogenous cluster. Although the asynchronous communication protocol, i.e., SSP, allows a delay bound to relax the synchronization among the workers, those fast workers have to wait for the slow peers when the delay is met. Such wait time impairs the convergence of DisSVRG. An approach to reduce the underlying wait time is to adjust the sampling strategy dynamically. Intuitively, the fast workers in the cluster should sample more instances during an epoch than the slow workers. That is, m in the fast workers should be larger than that in the slow workers.

We adopt an adaptive dynamic sampling strategy which dynamically adjust m. When an epoch is completed, the newly learned local parameters will be pushed to a server. The server will check the epochs of the worker. If the worker is too fast, it needs to wait for other slow peers. The fast workers will adjust m to be a large value, i.e., $m+\delta$. Here, δ is a non-negative integer with $\delta = 0.05m$ in DisSVRG. By using this adaptive strategy, the fast workers will sample more instances during an epoch, thus spend more time on computing the updates of parameters than the



(a) (b)

Figure 3: A large m leads to a fast convergence for DisSVRG, and reduces much wait time during iterations.

slow workers. The slow workers have chance to catch up with the fast workers. Therefore, the wait time is reduced as a result. This adaptive sampling strategy has many benefits. The most important benefit is that the fast workers reduce the wait time, and use it to converge DisSVRG. We conduct an evaluation test on a node by varying the value of m. Here, the dataset is still *YearPredictionMSD*, and the learning rate, i.e., η is set to be a constant with $\eta = 10^{-5}$. As illustrated in Fig. 3(a), it is obvious that a large m brings a fast convergence of the objective function. Therefore, a large m benefits to decrease the value of the loss function. Additionally, the wait time is compared in a cluster which consists of 5 nodes. As shown in Fig. 3(b), the average wait time has been evaluated by varying m. Here, the delay is set to be 0. It is obvious that the adaptive sampling strategy decreases the average wait time during iterations, and thus spend much time to accelerate the convergence of DisSVRG in reverse.

6. Discussion

The machine learning tasks such as deep learning are generally fed with an extremely large volume of training data. However, conventional serial versions of SGD cannot handle a huge training data on a single node within the available

time. Although some distributed machine learning systems such as Petuum [9] and DMTK[24] have been designed to solve this problem, those variants of SGD have their inner weakness, namely the variance. Such those distributed machine learning systems thus decrease the learning rate to reduce the variance, which leads to slow convergence of SGD. We do not aim to propose another a general platform for distributed machine learning algorithms, but focus on the optimization of SGD in a distributed system. Our implementation of the distributed SGD, i.e., DisSVRG, adopts the variance reduction technique to reduce the variance. Such variance reduction technique is the most difference between DisSVRG and PetummSGD. Although PetuumSGD adopts a decaying learning rate, it slows to converge the loss function. DisSVRG has been accelerated with two ingredients: the constant and the acceleration factor. The constant factor is effective to get DisSVRG out of the local optimum, and keeps converge at a constant rate. The acceleration factor will speedup the convergence of the machine learning algorithm. Even though DisSVRG adopts the same asynchronous communication protocol with PetuumSGD, it converges faster than PetuumSGD by adopting the powerful variance reduction technique.

DisSVRG improves SVRG with at least three aspects. First, we extend the serial SVRG to an asynchronous and distributed version by using the asynchronous consistency protocol, i.e., SSP. Second, comparing with the constant learning rate in SVRG, the learning rate in our SGD contains an acceleration factor which exploits the potential benefits of the loss function, and makes the machine learning algorithms converge fast. Third, SVRG needs to sample m instances randomly to update parameters. SVRG sets m multiple times of the size of training data, which is not practical for a large volume of training data. Instead, DisSVRG adopts an adaptive sampling strategy which adjusts the value of m by the runtime environment of the worker dynamically. Specifically, the fast workers will sample more instances during the next epoch than the slow peers. The slow workers thus have chance to catch up with the fast workers. Thus, wait time is reduced significantly,

which makes DisSVRG converge fast in the end. In a nutshell, considering the diversity of the runtime environment and the hardware in a cluster, DisSVRG is suitable to the practical scenarios.

Additionally, comparing with the version of SGD in [14] denoted by *SSGD*, our SGD adopts a more natural way to implement the distributed SGD with the variance reduction. SSGD implements the τ-delay bound inconsistent protocol within an epoch, but keeps fully consistent protocol among different workers. Therefore, SGD in [14] has at least two weaknesses. First, the fully consistency protocol for the workers is not suitable to the iterative convergence machine learning tasks [25, 26, 27]. Since the straggler problem usually exists among workers due to the variety of the system runtime environment or the hardware in the heterogenous cluster, the fast workers in [14] have to wait for the slow workers, and start the next iteration in a synchronous way. Considering that machine learning algorithms are iteratively convergent, the fully consistency protocol wastes too much time. Second, SSGD is coupled with specific hardware settings of clusters, which is less practical and natural. SSGD uses m learning threads to perform machine learning tasks where m is also the number of instances sampled in an epoch. That is, if the sampling strategy in SSGD adopts a large m, SSGD should be run in a cluster which can support m learning threads at a same time. Meanwhile, m in SSGD is $O(n)$ where n represents the size of the training data. Considering the huge size of training data, m is really large in SSGD, which is not practical in a real cluster. In fact, a practical SGD should be designed to be flexible to work in different clusters by merely adjusting its settings. Instead, m in DisSVRG is identified by the adaptive sampling mechanism, which is flexible to be adjusted in the runtime environment. This design of the sampling strategy shields the difference of a specific cluster, which is more general and natural.

7. Performance evaluation

In this section, we evaluate the performance of DisSVRG by using a regression problem and a classification problem on two datasets.

First we consider a regression problem:

$$\min \frac{1}{2n}\left(\frac{1}{1+e^{-\omega^{\mathrm{T}}x_i}}-y_i\right)^2. \tag{11}$$

Here, n is the size of training data. The dataset named *YearPredictionMSD*[1] is the biggest dataset on the LibSVM for the regression problem. It contains 463715 samples, and each sample has 90 dimensions.

Additionally, we consider a classification problem:

$$\min -\frac{1}{n}\sum_{i=1}^{n}\left(y_i \log\frac{1}{1+e^{-\omega x_i}}+(1-y_i)\log(1-\frac{1}{1+e^{-\omega x_i}})\right). \tag{12}$$

Similarly, n is the size of training data. The datasets named *dna*[2] is used for the evaluation tests. Every sample in the dataset has 200 dimensions. The number of samples in the *dna* is $50,000,000$.

We conduct the evaluation test on the HPC cluster of the *Tianhe-1* supercomputer which is located in the National Supercomputing Center in Changsha. We have a maximum of 128 computing nodes, and each such node is equipped with two Intel Xeon X5670 CPUs and one Nvidia M2050 GPU. Each a CPU has 6 cores; while GPUs are not used in the evaluation test. m is set to be 1000. The learning rate η is set to 10^{-6}. For the fairness of the evaluation test, we use the third-party open source distributed machine learning system denoted by DMTK [24] to conduct the performance evaluation. All the compared algorithms are implemented based on the DMTK.

The following algorithms will be used for comparison.

[1] `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`

[2] `ftp://largescale.ml.tu-berlin.de/largescale`

- **PetuumSGD:** The distributed version of SGD is implemented by using the asynchronous communication protocol, i.e., SSP [9]. The learning rate in PetuumSGD is decayed with a fixed factor 0.95 at the end of an epoch.

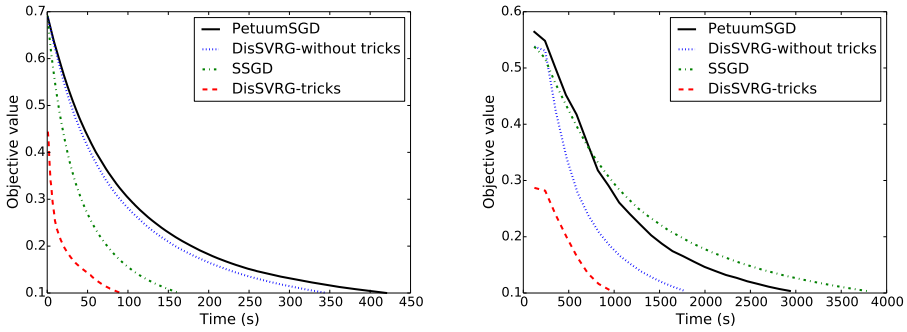
- **SSGD:** It is the state-of-the-art distributed version of SGD, which adopts the variance reduction technique [14]. The update rule in the SSGD has a variable θ which is used to update the parameters asynchronously. The details of SSGD can be referred in [14]. Here, we set $\theta = 0.5$.

- **DisSVRG-tricks:** DisSVRG is evaluated with all the optimization tricks.

- **DisSVRG-without-tricks:** DisSVRG is evaluated without any an optimization tricks.

7.1. Convergence

As illustrated in Fig. 4, the convergence performance of the algorithms has been evaluated. The delay is set to be 50 when those machine learning algorithms adopt the asynchronous communication protocol. All the datasets are handled on 32 workers. It is obvious that DisSVRG with the optimization tricks outperforms other algorithms for all the datasets. Even though DisSVRG does not use any an optimization trick, it always performs better than PetuumSGD, and gains a better performance than SSGD for the *dna* dataset. In specific, in order to decrease the loss function to 0.1, DisSVRG with all the optimization tricks spends one forth and one third time of the PetuumSGD for the datasets *YearPredictionMSD* and *dna*, respectively. The main reason is that DisSVRG adopts asynchronous communication protocol as well as the variance reduction technique to update the parameters, thus better than any of the existing algorithms. Additionally, the learning rate with an acceleration factor speeds up the convergence. Meanwhile, the adaptive sampling strategy significantly reduces the underlying wait time which is used to accelerate the convergence of DisSVRG in reverse.

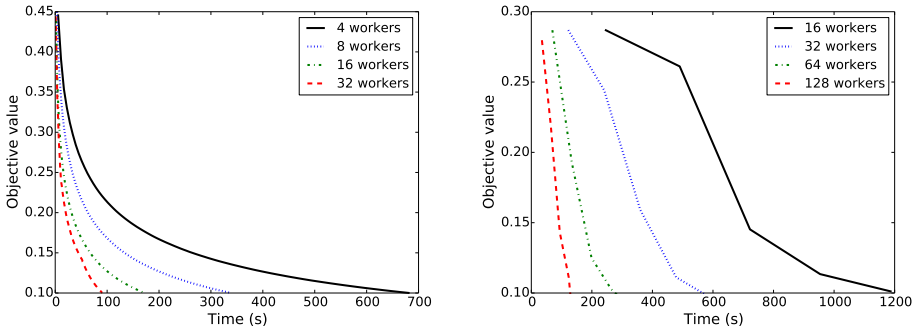


(a) *YearPredictionMSD* (b) *dna*

Figure 4: The performance of the convergence is compared by using 32 computing nodes.

7.2. Speedup

As illustrated in Fig. 5, we compare the convergence performance of DisSVRG by varying the number of workers in a cluster. It is obvious that DisSVRG converges fast with a large number of workers. During the iterations, the more workers are used to update the local parameters, the more newly learned parameters will be aggregated with the global parameters. After that, the global parameters which contains the newly learned updates of parameters will be shared with other workers for the next iteration, thus accelerating the convergence of other workers. In specific, DisSVRG obtains appriximately linear speedup when varying the number of workers. For instance, DisSVRG gains 19$\times$ speedup on the dataset *YearPredictionMSD* when using 32 workers. DisSVRG even keeps the linear speedup when using 128 workers for the dataset *dna*. The approximately linear speedup mainly benefits from the asynchronous communication protocol and the variance reduction technique. The asynchronous communication protocol relaxes the bound of the synchronization among workers, which alleviates the straggler problem, and thus decreases the wait time. The variance reduction technique reduces the variance of SGD, and keeps DisSVRG converging at a constant rate.

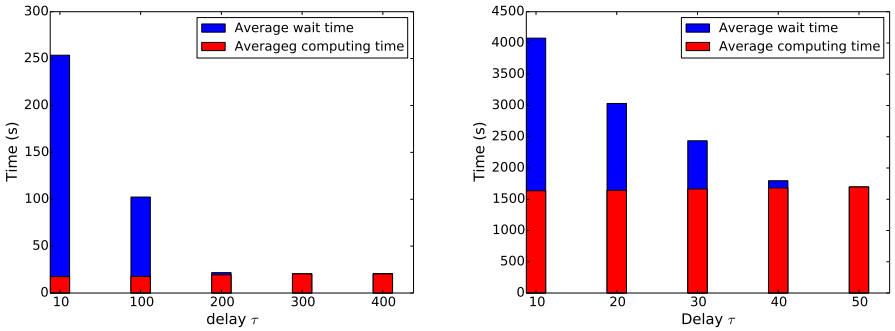


(a) *YearPredictionMSD* (b) *dna*

Figure 5: DisSVRG obtains almost linear speedup when varying the number of workers.

7.3. Wait time

As shown in Fig. 6, we evaluate the average time consumption of DisSVRG by varying the value of the delay τ. It is obvious when the delay τ is small, the average wait time is large. A small delay means a tight bound among workers, which usually leads to the wait time for the fast workers due to the asynchronous communication protocol. For example, when the delay is set to be 0, all the workers should be synchronized for each iteration, thus waiting the longest time. It is worth noting that the wait time decreases sharply when the delay becomes large. We conclude that a relax bound is effective to reduce the average wait time. Meanwhile, the average computing time will increase slightly with a relax bound of the delay. Although the fast workers have more freedom to conduct the iterations with the relax bound of the delay, the slow workers cannot obtain the newly learned parameters from the fast workers within the large delay. The slow workers thus cannot benefit from the fast peers. In a nutshell, the delay should not be identified either too small or too large. It is a tradeoff between the wait time and the computing time for the workers. Generally, the delay should be set to minimize the total time consumption of DisSVRG. In our evaluation tests, the delay τ should be set to be 200 for the dataset *YearPredictionMSD*, and 50 for the dataset *dna*.



(a) *YearPredictionMSD* (b) *dna*

Figure 6: The time consumption is compared by varying the delay τ for 128 workers.

8. Conclusion

Distributed SGD is an effective way to solve the large scale learning problems. In this paper, we propose a version of distributed SGD named DisSVRG with combing the asynchronous communication protocol and the variance reduction technique. Exploiting the properties of the loss function, DisSVRG is optimized by using a learning rate with the acceleration factor. Additionally, in order to reduce the wait time caused by the straggler problem, we propose an adaptive sampling strategy during the iterations. The adaptive sampling strategy gives the slow workers a chance to catch up with the fast peers during iterations, thus alleviating the straggler problem. Extensive empirical studies show that DisSVRG converges faster than the state-of-the-art version of SGD, and can gain approximately linear speedup in a cluster.

Acknowledgment

We thank the National Supercomputing Center in Changsha for providing *Tianhe-1* supercomputer as our experiment platform. This work was supported by the National Natural Science Foundation of China (Project NO. 61672528, 61170287, 61232016, 61303189 and 31501073).

References

[1] A. Coates, A. Y. Ng, H. Lee, An analysis of single-layer networks in unsupervised feature learning, Journal of Machine Learning Research 15 (2011) 215–223.

[2] G. E. Dahl, D. Yu, L. Deng, A. Acero, Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition, Transactions on Audio, Speech, and Language Processing 20 (1) (2012) 30–42.

[3] R. Collobert, J. Weston, A unified architecture for natural language processing: Deep neural networks with multitask learning, in: Proc. ACM ICML, 2008, pp. 160–167.

[4] D. Vidaurre, C. Bielza, P. Larrañaga, A Survey of L1Regression, International Statistical Review 81 (3) (2013) 361–387.

[5] G. Gnecco, M. Gori, S. Melacci, M. Sanguineti, Learning with mixed hard/soft pointwise constraints., IEEE Transactions on Neural Networks & Learning Systems 26 (9) (2015) 2019.

[6] F. Cucker, S. Smale, On the mathematical foundations of learning, Bulletin of the American Mathematical Society 39 (1) (2002) 332.

[7] J. Dean, G. Corrado, R. Monga, K. C. 0010, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, A. Y. Ng, Large scale distributed deep networks., in: Proc. NIPS, 2012, pp. 1232–1240.

[8] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, B.-Y. Su, Scaling distributed machine learning with the parameter server, in: Proc. UENSIX OSDI, 2014, pp. 583–598.

[9] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, Y. Yu, Petuum: a new platform for distributed machine learning on big data, Transactions on Big Data 1 (2) (2015) 49–67.

[10] R. Johnson, T. Zhang, Accelerating stochastic gradient descent using predictive variance reduction, in: Advances in Neural Information Processing Systems, 2013, pp. 315–323.

[11] S.-Y. Zhao, W.-J. Li, Fast asynchronous parallel stochastic gradient decent, arXiv:1508.05711.

[12] S. J. Reddi, A. Hefny, S. Sra, B. Poczos, A. J. Smola, On variance reduction in stochastic gradient descent and its asynchronous variants, in: Advances in Neural Information Processing Systems, 2015, pp. 2647–2655.

[13] R. L. Cavalcante, I. Yamada, B. Mulgrew, An adaptive projected subgradient approach to learning in diffusion networks, Transactions on Signal Processing 57 (7) (2009) 2762–2774.

[14] S. Z. Zhang, Ruiliang, J. T. Kwok, Fast distributed asynchronous sgd with variance reduction, arXiv:1508.01633.

[15] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, E. P. Xing, Addressing the straggler problem for iterative convergent parallel ML, in: the Seventh ACM Symposium, ACM Press, New York, New York, USA, 2016, pp. 98–111.

[16] M. Li, T. Zhang, Y. Chen, A. J. Smola, Efficient mini-batch training for stochastic optimization, in: the 20th ACM SIGKDD international conference, ACM Press, New York, New York, USA, 2014, pp. 661–670.

[17] H. Mania, X. Pan, D. Papailiopoulos, B. Recht, K. Ramchandran, M. I. Jordan, Perturbed iterate analysis for asynchronous stochastic optimization, arXiv:1507.06970.

[18] X. Lian, Y. Huang, Y. Li, J. Liu, Asynchronous parallel stochastic gradient for nonconvex optimization, in: Advances in Neural Information Processing Systems, 2015, pp. 2719–2727.

[19] X. Pan, M. Lam, S. Tu, D. Papailiopoulos, C. Zhang, M. I. Jordan, K. Ramchandran, C. Re, B. Recht, CYCLADES: Conflict-free Asynchronous Machine LearningarXiv:1605.09721v1.

[20] A. Defazio, F. Bach, S. Lacoste-Julien, Saga: A fast incremental gradient method with support for non-strongly convex composite objectives, in: Advances in Neural Information Processing Systems, Montréal, Canda, 2014, pp. 1646–1654.

[21] J. Konečnỳ, P. Richtárik, Semi-stochastic gradient descent methods, arXiv preprint arXiv:1312.1666.

[22] Z. Allen-Zhu, Y. Yuan, Improved SVRG for non-strongly-convex or sum-of-non-convex objectives, in: International Conference on Machine Learning, New York, USA, 2016.

[23] L. Xiao, T. Zhang, A proximal stochastic gradient method with progressive variance reduction, SIAM Journal on Optimization 24 (4) (2014) 2057–2075.

[24] J. Yuan, F. Gao, Q. Ho, W. Dai, J. Wei, X. Zheng, E. P. Xing, T.-Y. Liu, W.-Y. Ma, Lightlda: Big topic models on modest computer clusters, in: Proc. WWW, 2015, pp. 1351–1361.

[25] W. Dai, A. Kumar, J. Wei, Q. Ho, G. A. Gibson, E. P. Xing, High-performance distributed ml at scale through parameter server consistency models, in: Proc. AAAI, 2015, pp. 79–87.

[26] M. Li, D. G. Andersen, A. J. Smola, K. Yu, Communication efficient distributed machine learning with the parameter server, in: Advances in Neural Information Processing Systems, 2014, pp. 19–27.

[27] W. Dai, J. Wei, X. Zheng, J. K. Kim, S. Lee, J. Yin, Q. Ho, E. P. Xing, Petuum: A framework for iterative-convergent distributed ml, arXiv:1312.7651.