Fast Asynchronous Parallel Stochastic Gradient Descent: A Lock-Free Approach with Convergence Guarantee

Shen-Yi Zhao and **Wu-Jun Li**
National Key Laboratory for Novel Software Technology
Department of Computer Science and Technology, Nanjing University, China
zhaosy@lamda.nju.edu.cn, liwujun@nju.edu.cn

Abstract

Stochastic gradient descent (SGD) and its variants have become more and more popular in machine learning due to their efficiency and effectiveness. To handle large-scale problems, researchers have recently proposed several parallel SGD methods for multicore systems. However, existing parallel SGD methods cannot achieve satisfactory performance in real applications. In this paper, we propose a fast asynchronous parallel SGD method, called AsySVRG, by designing an asynchronous strategy to parallelize the recently proposed SGD variant called stochastic variance reduced gradient (SVRG). AsySVRG adopts a lock-free strategy which is more efficient than other strategies with locks. Furthermore, we theoretically prove that AsySVRG is convergent with a linear convergence rate. Both theoretical and empirical results show that AsySVRG can outperform existing state-of-the-art parallel SGD methods like Hogwild! in terms of convergence rate and computation cost.

Introduction

Assume we have a set of labeled instances $\{(\mathbf{x}_i, y_i)|i = 1, \dots, n\}$, where $\mathbf{x}_i \in \mathbb{R}^d$ is the feature vector for instance i, d is the feature size and $y_i \in \{1, -1\}$ is the class label of $\mathbf{x}_i$. In machine learning, we often need to solve the following regularized empirical risk minimization problem:

$$\min_{\mathbf{w}} \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{w}), \tag{1}$$

where $\mathbf{w}$ is the parameter to learn, $f_i(\mathbf{w})$ is the loss function defined on instance i which is often with a regularization term to avoid overfitting. For example, $f_i(\mathbf{w})$ can be $\log(1+e^{-y_i\mathbf{x}_i^T\mathbf{w}}) + \frac{\lambda}{2}\|\mathbf{w}\|^2$ which is known as the logistic loss plus a regularization term, or $\max\left\{0, 1-y_i\mathbf{x}_i^T\mathbf{w}\right\} + \frac{\lambda}{2}\|\mathbf{w}\|^2$ which is known as the regularized hinge loss in support vector machine (SVM). Besides $\frac{\lambda}{2}\left\|\mathbf{w}\right\|^2$, the regularization term can also be $\lambda\left\|\mathbf{w}\right\|_1$ or some other forms.

Due to their efficiency and effectiveness, stochastic gradient descent (SGD) and its variants (Xiao 2009; Duchi and Singer 2009; Roux, Schmidt, and Bach 2012; Johnson and Zhang 2013; Mairal 2013; Shalev-Shwartz and Zhang 2013; Liu et al. 2014; Nitanda 2014; Zhang and Kwok 2014) have recently attracted much attention to solve machine learning problems like that in (1). Many works have proved that SGD and its variants can outperform traditional batch learning algorithms such as gradient descent or Newton methods in real applications.

In many real-world problems, the number of instances n is typically very large. In this case, the traditional sequential SGD methods might not be efficient enough to find the optimal solution for (1). On the other hand, clusters and multicore systems have become popular in recent years. Hence, to handle large-scale problems, researchers have recently proposed several distributed SGD methods for clusters and parallel SGD methods for multicore systems. Although distributed SGD methods for clusters like those in (Zinkevich, Smola, and Langford 2009; Duchi, Agarwal, and Wainwright 2010; Zinkevich et al. 2010; Zhang, Zheng, and T. Kwok 2015) are meaningful to handle very large-scale problems, there also exist a lot of problems which can be solved by a single machine with multiple cores. Furthermore, even in distributed settings with clusters, each machine (node) of the cluster typically have multiple cores. Hence, how to design effective parallel SGD methods for multicore systems has become a key issue to solve large-scale learning problems like that in (1).

There have appeared some parallel SGD methods for multicore systems. The round-robin scheme proposed in (Zinkevich, Smola, and Langford 2009) tries to order the processors and then each processor update the variables in order. Hogwild! (Recht et al. 2011) is an asynchronous approach for parallel SGD. Experimental results in (Recht et al. 2011) have shown that Hogwild! can outperform the round-robin scheme in (Zinkevich, Smola, and Langford 2009). However, Hogwild! can only achieve a sub-linear convergence rate. Hence, Hogwild! is not efficient (fast) enough to achieve satisfactory performance. PASSCoDe (Hsieh, Yu, and Dhillon 2015) and CoCoA (Jaggi et al. 2014) are also asynchronous approaches for parallel SGD. However, both of them are formulated from the dual coordinate descent (ascent) perspective, and hence it can only be used for problems whose dual functions can be computed. Although some works, such as Hogwild! and PASSCoDe, have empirically found that in some cases the lock-free strategies are much more efficient than other strategies with locks, no works have theoretically proved the convergence of the lock-free strategies. For example, both Hogwild! and PASSCoDe are proved to be convergent under the assumption that there are some locks to guarantee that over-writing operation would never happen. However, both of them have not provided theoretical guarantee for the convergence under the lock-free case.

In this paper, we propose a fast asynchronous parallel SGD method, called AsySVRG, by designing an asynchronous strategy to parallelize the recently proposed SGD variant called stochastic variance reduced gradient (SVRG) (Johnson and Zhang 2013). The contributions of AsySVRG can be outlined as follows:

- AsySVRG is a lock-free asynchronous parallel approach, which means that there is no read lock or update (write) lock in AsySVRG. Hence, AsySVRG can be expected to be more efficient than other approaches with locks.
- Although AsySVRG is lock-free, we can still theoretically prove that it is convergent with a linear convergence rate[1], which is faster than that of Hogwild!.
- The implementation of AsySVRG is simple.
- Empirical results on real datasets show that AsySVRG can outperform Hogwild! in terms of convergence rate and computation cost.

Preliminary

We use $f(\mathbf{w})$ to denote the objective function in (1), which means $f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{w})$. In this paper, we use $\|\cdot\|$ to denote the L_2-norm $\|\cdot\|_2$ and $\mathbf{w}_*$ to denote the optimal solution of the objective function.

Assumption 1. *The function $f_i(\cdot)$ $(i = 1, \ldots, n)$ in (1) is convex and L-smooth, which means that $\exists L > 0$, $\forall \mathbf{a}, \mathbf{b}$,*

$$f_i(\mathbf{a}) \leq f_i(\mathbf{b}) + \nabla f_i(\mathbf{b})^T(\mathbf{a} - \mathbf{b}) + \frac{L}{2}\|\mathbf{a} - \mathbf{b}\|^2,$$

or equivalently

$$\|\nabla f_i(\mathbf{a}) - \nabla f_i(\mathbf{b})\| \leq L\|\mathbf{a} - \mathbf{b}\|,$$

where $\nabla f_i(\cdot)$ denotes the gradient of $f_i(\cdot)$.

Assumption 2. *The objective function $f(\cdot)$ is μ-strongly convex, which means that $\exists \mu > 0$, $\forall \mathbf{a}, \mathbf{b}$,*

$$f(\mathbf{a}) \geq f(\mathbf{b}) + \nabla f(\mathbf{b})^T(\mathbf{a} - \mathbf{b}) + \frac{\mu}{2}\|\mathbf{a} - \mathbf{b}\|^2,$$

or equivalently

$$\|\nabla f(\mathbf{a}) - \nabla f(\mathbf{b})\| \geq \mu\|\mathbf{a} - \mathbf{b}\|.$$

Please note that Assumptions 1 and 2 are often satisfied by most objective functions in machine learning models, such as the logistic regression (LR) and linear regression with L_2-norm regularization.

[1]In our early work, the parallel SVRG algorithms with locks have been proved to be convergent (Zhao and Li 2015).

Approach

Assume that we have p threads (processors) which can access a shared memory, and $\mathbf{w}$ is stored in the shared memory. Furthermore, we assume each thread has access to a shared data structure for the vector $\mathbf{w}$ and has access to randomly choose any instance i to compute the gradient $\nabla f_i(\mathbf{w})$.

Our AsySVRG algorithm is presented in Algorithm 1. We can find that in the t^{th} iteration, each thread completes the following operations:

- By using a temporary variable $\mathbf{u}_0$ to store $\mathbf{w}_t$ (i.e., $\mathbf{u}_0 = \mathbf{w}_t$), all threads parallelly compute the full gradient $\nabla f(\mathbf{u}_0) = \frac{1}{n}\sum_{i=1}^{n}\nabla f_i(\mathbf{u}_0) = \frac{1}{n}\sum_{i=1}^{n}\nabla f_i(\mathbf{w}_t)$. Assume the gradients computed by thread a are denoted by ϕ_a which is a subset of $\{\nabla f_i(\mathbf{w}_t)|i = 1, \ldots, n\}$. We have $\phi_a \bigcap \phi_b = \varnothing$ if $a \neq b$, and $\bigcup_{a=1}^{p}\phi_a = \{\nabla f_i(\mathbf{w}_t)|i = 1, \ldots, n\}$.
- Then each thread parallelly runs an inner-loop in each iteration of which the thread reads the current value of $\mathbf{u}$, denoted as $\hat{\mathbf{u}}$, from the shared memory and randomly chooses an instance indexed by i to compute the vector

  $$\hat{\mathbf{v}} = \nabla f_i(\hat{\mathbf{u}}) - \nabla f_i(\mathbf{u}_0) + \nabla f(\mathbf{u}_0). \quad (2)$$

  Then update the vector

  $$\mathbf{u} \leftarrow \mathbf{u} - \eta\hat{\mathbf{v}},$$

  where $\eta > 0$ is a step size (or called learning rate).

Here, we use $\mathbf{w}$ to denote the parameter in the outer-loop, and use $\mathbf{u}$ to denote the parameter in the inner-loop. Before running the inner-loops, $\mathbf{u}$ will be initialized by the current value of $\mathbf{w}_t$ in the shared memory. After all the threads have completed the inner-loops, we take $\mathbf{w}_{t+1}$ to be the current value of $\mathbf{u}$ in the shared memory.

Algorithm 1 will degenerate to stochastic variance reduced gradient (SVRG) (Johnson and Zhang 2013) if there exists only one thread (i.e., $p = 1$). Hence, Algorithm 1 is a parallel version of SVRG. Furthermore, in Algorithm 1, all

Algorithm 1 AsySVRG

```
Initialization: $p$ threads, initialize $\mathbf{w}_0, \eta$;
for $t = 0, 1, 2, ...$ do
    $\mathbf{u}_0 = \mathbf{w}_t$;
    All threads parallelly compute the full gradient $\nabla f(\mathbf{u}_0) = \frac{1}{n}\sum_{i=1}^{n}\nabla f_i(\mathbf{u}_0)$;
    $\mathbf{u} = \mathbf{w}_t$;
    For each thread, do:
    for $m = 1$ to $M$ do
        Read current value of $\mathbf{u}$, denoted as $\hat{\mathbf{u}}$, from the shared memory. And randomly pick up an $i$ from $\{1, \ldots, n\}$;
        Compute the update vector: $\hat{\mathbf{v}} = \nabla f_i(\hat{\mathbf{u}}) - \nabla f_i(\mathbf{u}_0) + \nabla f(\mathbf{u}_0)$;
        $\mathbf{u} \leftarrow \mathbf{u} - \eta\hat{\mathbf{v}}$;
    end for
    Take $\mathbf{w}_{t+1}$ to be the current value of $\mathbf{u}$ in the shared memory;
end for
```

threads read and write the shared memory without any locks. Hence, Algorithm 1 is a lock-free asynchronous approach to parallelize SVRG, which is called AsySVRG in this paper.

An Equivalent Synthetic Process

In the lock-free case, we do not use any lock whenever one thread reads $\mathbf{u}$ from the shared memory or updates (writes) $\mathbf{u}$ in the shared memory. Hence, the results of $\mathbf{u}$ seem to be totally disordered, which makes the convergence analysis very difficult.

In this paper, we find that there exists a *synthetic* process to generate the final value of $\mathbf{u}$ after all threads have completed their updates in the inner-loop of Algorithm 1. It means that we can generate a sequence of synthetic values of $\mathbf{u}$ with some order to get the final $\mathbf{u}$, based on which we can prove the convergence of the lock-free AsySVRG in Algorithm 1.

Synthetic Write (Update) Sequence

The key step in the inner-loop of Algorithm 1 is $\mathbf{u} \leftarrow \mathbf{u} - \eta\hat{\mathbf{v}}$, which can be rewritten as follows:

$$\mathbf{u} \leftarrow \mathbf{u} + \Delta, \tag{3}$$

where $\Delta = -\eta\hat{\mathbf{v}}$ is the update vector computed by each thread.

Let $\mathbf{u} = (u^{(1)}, u^{(2)}, \dots, u^{(d)})$ with $u^{(i)}$ denoting the ith element of $\mathbf{u}$. Since each thread has a local count, we use $\Delta_{i,j}$ to denote the j^{th} update vector produced by the i^{th} thread ($i = 1, 2, \dots, p$, $j = 1, 2, \dots, M$), $t_{i,j}^{(k)}$ to denote the time that $u^{(k)}$ is changed by $\Delta_{i,j}^{(k)}$. Without loss of generality, we assume all threads will update the elements in $\mathbf{u}$ in the order from $u^{(1)}$ to $u^{(d)}$, which can be easily implemented by the program. Hence, we have

$$\forall i, j, \qquad t_{i,j}^{(1)} < t_{i,j}^{(2)} < \ldots < t_{i,j}^{(d)} \tag{4}$$

$$\forall i, k, l, \qquad \textit{if} \quad m < n, \qquad t_{i,m}^{(k)} < t_{i,n}^{(l)} \tag{5}$$

and these $\{t_{i,j}^{(1)}\}$s are different from each other since $u^{(1)}$ can be changed by only one thread at an absolute time. We sort these $\{t_{i,j}^{(1)}\}$s in an increasing order and use $\Delta_0, \Delta_1, \dots, \Delta_m, \dots, \Delta_{\tilde{M}-1}$, where $\tilde{M} = p \times M$, to denote the corresponding update vectors. It is more useful that we sort $t_{i,j}^{(1)}$ than that we sort $t_{i,j}^{(k)} (k > 1)$ because before $t_{i,j}^{(1)}$, the update vector $\Delta_{i,j}$ has not been applied to $\mathbf{u}$. Furthermore, we will benefit from such a sort when we discuss what would be read by one thread. Since we do not use any lock, over-writing may happen when one thread is performing the update in (3). The real update vector can be written as $\mathbf{B}_m\Delta_m$, where $\mathbf{B}_m$ is a diagonal matrix whose diagonal entries are 0 or 1. If $\mathbf{B}_m(k, k) = 0$, then $\Delta_m^{(k)}$ is over-written by other threads. If $\mathbf{B}_m(k, k) = 1$, then $u^{(k)}$ is updated by $\Delta_m^{(k)}$ successfully without over-writing. However, we do not know what the exact $\mathbf{B}_m$ is. It can be seen as a random variable. Then (3) can be rewritten as

$$\mathbf{u} \leftarrow \mathbf{u} + \mathbf{B}_m\Delta_m$$

When all the inner-loops of all threads are completed, we can get the current $\mathbf{u}$ in the shared memory, which can be presented as

$$\mathbf{u} = \mathbf{u}_0 + \sum_{i=0}^{\tilde{M}-1} \mathbf{B}_i\Delta_i \tag{6}$$

According to (6) and the definition of Δ_m, we can define a synthetic write (update) sequence $\{\mathbf{u}_m\}$ with $\mathbf{u}_0 = \mathbf{w}_t$, and for $m = 1 : \tilde{M}$,

$$\mathbf{u}_m = \mathbf{u}_0 + \sum_{i=0}^{m-1} \mathbf{B}_i\Delta_i \tag{7}$$

It is easy to see that

$$\mathbf{u}_{m+1} = \mathbf{u}_m + \mathbf{B}_m\Delta_m.$$

We can also find that $\mathbf{u}_m$ is the value which can be got after the update vectors $\Delta_0, \dots, \Delta_{m-1}$ have been completely applied to $\mathbf{u}$ in the shared memory.

Please note that the sequence $\{\mathbf{u}_m\}$ is synthetic and the whole $\mathbf{u}_m = (u_m^{(1)}, u_m^{(2)}, \dots, u_m^{(d)})$ may never occur in the shared memory, which means that we cannot obtain any $\mathbf{u}_m$ ($m = 1, 2, \dots, \tilde{M} - 1$) or the average sum of these $\mathbf{u}_m$ during the running of the inner-loop. What we can get is only the final value of $\mathbf{u}_{\tilde{M}}$ after all threads have completed their updates in the inner-loop of Algorithm 1. Hence, we can find an equivalent synthetic update process with some order to generate the same value as that of the disordered lock-free update process.

Read Sequence

We use $\hat{\mathbf{u}}_m$ to denote the parameter read from the shared memory which is used to compute Δ_m by a thread. Based on the synthetic write sequence $\{\mathbf{u}_m\}$, $\hat{\mathbf{u}}_m$ can be written as $\hat{\mathbf{u}}_m = \mathbf{u}_{a(m)} + \sum_{i=a(m)}^{b(m)} \mathbf{P}_{m,i-a(m)}\Delta_i$. Here, $a(m) < m$ denotes some time point when the update vectors $\Delta_0, \dots, \Delta_{a(m)-1}$ have been completely applied to $\mathbf{u}$ in the shared memory. $\{\mathbf{P}_{m,i-a(m)}\}$ are diagonal matrices whose diagonal entries are 0 or 1. $b(m) \geq a(m)$ denotes another time point. $\sum_{i=a(m)}^{b(m)} \mathbf{P}_{m,i-a(m)}\Delta_i$ means that besides $\mathbf{u}_{a(m)}$, $\hat{\mathbf{u}}_m$ also includes some other new update vectors from time point $a(m)$ to $b(m)$. According to the definition of Δ_i, all $\{\Delta_i\}$ $(i \geq m)$ have not been applied to $\mathbf{u}$ at the time point when one thread gets $\hat{\mathbf{u}}_m$. Then, we can set $b(m) < m$. Hence, we can reformulate $\hat{\mathbf{u}}_m$ as

$$\hat{\mathbf{u}}_m = \mathbf{u}_{a(m)} + \sum_{i=a(m)}^{m-1} \mathbf{P}_{m,i-a(m)}\Delta_i. \tag{8}$$

The matrix $\mathbf{P}_{m,i-a(m)}$ means that $\hat{\mathbf{u}}_m$ may read some components of these new $\{\Delta_i\}$ $(a(m) \leq i < m)$, including those which might be over-written by some other threads. It can also be seen as a random variable. $\mathbf{P}_{m,i-a(m)}$ and $\mathbf{B}_i$ are not necessary to be equal to each other.

An Illustrative Example

We give a lock-free example to illustrate the above synthetic process. Assume $\mathbf{u} = (1, 1)$ is stored in the shared memory. There are three threads which we use Ta, Tb and Tc to denote. The task of Ta is to add 1 on each component of $\mathbf{u}$, the task of Tb is to add 0.1 on each component of $\mathbf{u}$, and the task of Tc is to read $\mathbf{u}$ from the shared memory. If the final result of $\mathbf{u} = (2, 2.1)$, one of the possible update sequences of $\mathbf{u}$ can be presented as follows:

Time	$\mathbf{u}$
Time 0	(1, 1)
Time 1	(1.1, 1)
Time 2	(2, 1)
Time 3	(2, 2)
Time 4	(2, 2.1)

It is easy to see that over-writing happens at Time 2. At Time 1, Tb is faster then Ta. At Time 3, Ta is faster than Tb. Then we can define

$$\mathbf{u}_0 = (1, 1)$$
$$\mathbf{u}_1 = \mathbf{u}_0 + \mathbf{B}_0\Delta_0 = (1, 1.1)$$
$$\mathbf{u}_2 = \mathbf{u}_1 + \mathbf{B}_1\Delta_1 = (2, 2.1)$$

where

$$\Delta_0 = (0.1, 0.1)^T,\ \Delta_1 = (1, 1)^T,$$

and

$$\mathbf{B}_0 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, \mathbf{B}_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

We can find that the whole $\mathbf{u}_1 = (1, 1.1)$ never occurs in the shared memory, which means that it is synthetic. The result of Tc can be any $\mathbf{u}$ at Time 0, Time 1, ..., Time 4, and even some other value such as $(1.1, 2.1)$. The $\mathbf{u}$ at Time 1 can also be read by Tc although the $u^{(0)}$ written by Tb at Time 1 was over-written by Ta. It is easy to verify that any result of Tc can be presented as the format of (8). For example, if Tc reads the $\mathbf{u} = (1.1, 2.1)$, then the read value can be written as:

$$(1.1, 2.1) = \mathbf{u}_0 + \mathbf{P}_0\Delta_0 + \mathbf{P}_1\Delta_1$$

where

$$\mathbf{P}_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \mathbf{P}_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

Convergence Analysis

In this section, we will focus on the convergence analysis for the lock-free AsySVRG. Let $\hat{\mathbf{v}}_{i,j}$ denote the j^{th} stochastic gradient produced by the i^{th} thread ($i = 1, 2, \dots, p$, $j = 1, 2, \dots, M$), then $\Delta_{i,j} = -\eta\hat{\mathbf{v}}_{i,j}$. We can also give an order of these $\hat{\mathbf{v}}_{i,j}$ and define a synthetic sequence $\{\mathbf{u}_m\}$ according to the discussion in the above section:

$$\mathbf{u}_0 = \mathbf{w}_t,$$
$$\mathbf{u}_{m+1} = \mathbf{u}_m - \eta\mathbf{B}_m\hat{\mathbf{v}}_m. \tag{9}$$

In (9), the stochastic gradient $\hat{\mathbf{v}}_m = \nabla f_{i_m}(\hat{\mathbf{u}}_m) - \nabla f_{i_m}(\mathbf{u}_0) + \nabla f(\mathbf{u}_0)$, where $\hat{\mathbf{u}}_m$ is got from the shared memory by the thread which computes $\hat{\mathbf{v}}_m$ and i_m is the random index of the instance chosen by this thread. $\mathbf{B}_m$ is a diagonal matrix whose diagonal entries are 0 or 1.

To get our convergence result, we give some assumptions and definitions.

Assumption 3. *(Bound delay assumption)*
$0 \le m - a(m) \le \tau$.

Assumption 4. *The conditional expectation of* $\mathbf{B}_m$ *on* $\mathbf{u}_m$ *and* $\hat{\mathbf{u}}_m$ *is strictly positive definite, i.e.,* $\mathbb{E}[\mathbf{B}_m|\mathbf{u}_m, \hat{\mathbf{u}}_m] = \mathbf{B} \succ 0$ *with the minimum eigenvalue* $\alpha > 0$.

Assumption 5. *(Dependence assumption)* $\mathbf{B}_m$ *and* i_m *are conditional independent on* $\mathbf{u}_m$ *and* $\hat{\mathbf{u}}_m$*, where* i_m *is the index of the randomly chosen instance.*

Assumption 3 means that when one thread gets the $\hat{\mathbf{u}}_m$, at least $\Delta_0, \Delta_1, \dots, \Delta_{m-\tau-1}$ have been completely applied (updated) to $\mathbf{u}$. The τ is a parameter for bound delay. In real applications, we cannot control the bound delay since we do not have any lock. In our experiments of logistic regression, we find that we do not need the bound delay. The phenomenon can be explained as that our threads are stable and the process of computing full gradient can be seen as a "delay", although such a "delay" is relatively large.

According to the definition of $\mathbf{u}_m$ and $\hat{\mathbf{u}}_m$, both of them are determined by these random variables $\mathbf{B}_l, \Delta_l, i_l$ ($l \le m-1$) and $\{\mathbf{P}_{m,j}\}$ ($0 \le j < m - a(m)$). So we can take conditional expectation about $\mathbf{B}_m$ on $\mathbf{u}_m$ and $\hat{\mathbf{u}}_m$, which is $\mathbb{E}[\mathbf{B}_m|\mathbf{u}_m, \hat{\mathbf{u}}_m] = \mathbf{B}$ in Assumption 4. Since $\mathbf{B}_m$ is a diagonal positive semi-definite matrix, then the expectation of $\mathbf{B}_m$ is still a diagonal positive semi-definite matrix. Assumption 4 only needs $\mathbf{B}$ to be a strictly positive definite matrix, which means that the minimum eigenvalue of $\mathbf{B}$ is strictly positive. According to Assumption 4, for each thread, after it has got a $\hat{\mathbf{u}}$ from the shared memory, the probability that over-writing happens when updating on the k^{th} component of $\mathbf{u}$ is $1 - \mathbf{B}(k, k)$. If one of the eigenvalues of $\mathbf{B}$ is zero, it means that over-writing always happens on that component of $\mathbf{u}$, which is not common. Hence, Assumption 4 is reasonable.

In most modern hardware, $\mathbf{B}(k, k)$ is close to 1. Moreover, if we use atomic operation or update lock, $\mathbf{B}(k, k) = 1$. So Assumption 5 is also reasonable since the $\mathbf{B}_m$ is highly affected by the hardware but i_m is independent of the hardware.

Definition 1.

$$\mathbf{p}_i(\mathbf{x}) = \nabla f_i(\mathbf{x}) - \nabla f_i(\mathbf{u}_0) + \nabla f(\mathbf{u}_0) \tag{10}$$

$$q(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} \|\mathbf{p}_i(\mathbf{x})\|^2 \tag{11}$$

According to (10) and (11), it is easy to prove that $\mathbb{E}_i[\|\mathbf{p}_i(\mathbf{x})\|^2] = q(\mathbf{x})$ and $\hat{\mathbf{v}}_m = \mathbf{p}_{i_m}(\hat{\mathbf{u}}_m)$.

In the following content, we will prove the convergence of our algorithm. All the proofs can be found at the supplementary material[2].

[2]The supplementary material can be downloaded from http://cs.nju.edu.cn/lwj/paper/AsySVRG_sup.pdf.

Lemma 1. *$\forall \mathbf{x}, \mathbf{y}, r > 0, i$, we have*

$$\|\mathbf{p}_i(\mathbf{x})\|^2 - \|\mathbf{p}_i(\mathbf{y})\|^2 \leq \frac{1}{r}\|\mathbf{p}_i(\mathbf{x})\|^2 + rL^2\|\mathbf{x} - \mathbf{y}\|^2.$$

Lemma 2. *For any constant $\rho > 1$, we have $\mathbb{E}q(\hat{\mathbf{u}}_m) < \rho\mathbb{E}q(\hat{\mathbf{u}}_{m+1})$ if we choose r and η to satisfy that*

$$\frac{1}{1 - \frac{1}{r} - 9r(\tau+1)L^2\eta^2} < \rho$$

$$\frac{1}{1 - \frac{1}{r} - \frac{9r(\tau+1)L^2\eta^2(\rho^{\tau+1}-1)}{\rho-1}} < \rho.$$

Lemma 3. *With the assumption in Lemma 2 about r, ρ, η, we have $\mathbb{E}q(\hat{\mathbf{u}}_m) < \rho\mathbb{E}q(\mathbf{u}_m)$.*

Theorem 1. *With Assumption 1, 2, 3, 4, 5, and taking $\mathbf{w}_{t+1}$ to be the last one of $\{\mathbf{u}_m\}$, we have*

$$\mathbb{E}f(\mathbf{w}_{t+1}) - f(\mathbf{w}_*) \leq (c_1^{\tilde{M}} + \frac{c_2}{1-c_1})(\mathbb{E}f(\mathbf{w}_t) - f(\mathbf{w}_*)),$$

where $c_1 = 1 - \alpha\eta\mu + c_2$ and $c_2 = \eta^2(\frac{8\tau L^3\eta\rho^2(\rho^\tau-1)}{\rho-1} + 2L^2\rho)$, $\tilde{M} = p \times M$ is the total number of iterations of the inner-loop.

In Theorem 1, the constant $c_2 = O(\eta^2)$. We can choose η such that $\frac{c_2}{1-c_1} = O(\eta) < 1$ and $c_1 < 1$. Hence, our algorithm gets a linear convergence rate with a lock-free strategy.

Experiment

We choose logistic regression (LR) with a L_2-norm regularization term to evaluate our AsySVRG. Hence, the $f(\mathbf{w})$ is defined as follows:

$$f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\left[\log(1 + e^{-y_i\mathbf{x}_i^T\mathbf{w}}) + \frac{\lambda}{2}\|\mathbf{w}\|^2\right].$$

The experiments are conducted on a server with 12 Intel cores and 64G memory.

Dataset

Four datasets are used for evaluation. They are *rcv1, real-sim, news20*, and *epsilon*, which can be downloaded from the LibSVM website[3]. Detailed information is shown in Table 1, where *sparse* means the features have many zeroes and *dense* means the features have few zeroes. Since we only consider the speedup and convergence rate on the training data, we simply set the hyper-parameter $\lambda = 10^{-4}$ in $f(\mathbf{w})$ for all the experiments.

Table 1: Dataset

dataset	instances	features	memory	type
rcv1	20,242	47,236	36M	sparse
real-sim	72,309	20,958	90M	sparse
news20	19,996	1,355,191	140M	sparse
epsilon	400,000	2,000	11G	dense

[3]http://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/

Baselines

Hogwild! and our AsySVRG are formulated from the primal perspective, but PASSCoDe and CoCoA are formulated from the dual perspective. Hence, the most related work for our AsySVRG is Hogwild!, which is chosen as the baseline for comparison.

In particular, we compare AsySVRG with the following four variants:

- *Hogwild!-lock*: The lock version of Hogwild!, which need a lock (write-lock) when each thread is going to update (write) the parameter.
- *Hogwild!*: The lock-free version of Hogwild!, which does not use any lock during the whole algorithm.
- *AsySVRG-lock*: The lock version of AsySVRG, which need a lock (write-lock) when each thread is going to update (write) the parameter.
- *AsySVRG*: The lock-free version of AsySVRG, which does not use any lock during the whole algorithm.

Please note that we do not use any lock (read-lock) when each thread is reading parameter from the shared memory for AsySVRG and other baselines, which means all the threads perform *inconsistent read*.

We set M in Algorithm 1 to be $\frac{2n}{p}$, where n is the number of training instances and p is number of threads. When $p = 1$, the setting about M is the same as that in SVRG (Johnson and Zhang 2013). According to our theorems, the step size should be small. However, we can also get good performance with a relatively large step size in practice. For Hogwild!, in each epoch, each thread runs $\frac{n}{p}$ iterations. We use a constant step size γ, and we set $\gamma \leftarrow 0.9\gamma$ after every epoch. These settings are the same as those in the experiments of Hogwild! (Recht et al. 2011).

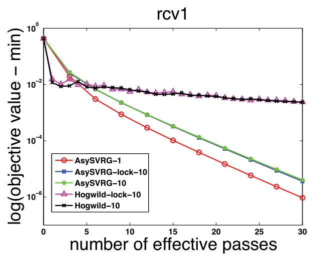
Result

Convergence Rate We get a suboptimal solution by stopping the algorithms when the gap between the training loss and the optimal solution $\min\{f(\mathbf{w})\}$ is less than 10^{-4}. For each epoch, our algorithm visits the whole dataset three times and Hogwild! visits the whole dataset only once. To make a fair comparison about the convergence rate, we study the change of objective function value versus the number of *effective passes*. One effective pass of the dataset means the whole dataset is visited once.

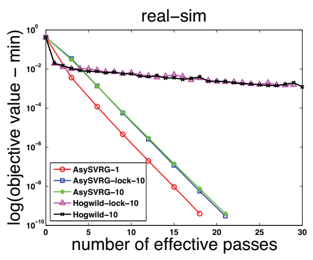
Figure 1 shows the convergence rate with respect to effective passes on four datasets. Here, AsySVRG-lock-10 denotes AsySVRG-lock with 10 threads. Similar notations are used for other variants of AsySVRG and Hogwild!. In particular, AsySVRG-1 is actually the original non-parallel version of SVRG (Johnson and Zhang 2013). We can find that the convergence rate of AsySVRG and its variants is almost linear (please note that the vertical axis is in a log scale), which is much faster than that of Hogwild! and its variants. Hence, the empirical results successfully verify the correctness of our theoretical results.

There also exists another interesting phenomenon that the curves of AsySVRG-1, AsySVRG-lock-10 and AsySVRG-10 are close to each other. The number of effective passes actually measures the computation cost. Hence, these curves mean that the corresponding methods need almost the same computation cost to get the same objective function value. For example, AsySVRG-1 and AsySVRG-10 need the same computation cost on news20 dataset according to the results in Figure 1. Hence, if there does not exist any other cost, AsySVRG-10 can be expected to be 10 times faster than AsySVRG-1 because AsySVRG-10 has 10 threads to parallelly (simultaneously) perform computation. However, AsySVRG-10 typically cannot achieve the speed of 10 times faster than AsySVRG-1 because there also exist other costs, such as inter-thread communication cost, for multiple-thread cases. Similarly, AsySVRG-lock-10 and AsySVRG-10 cannot have the same speed because AsySVRG-lock-10 need extra lock cost compared with AsySVRG-10. This will be verified in the next section about speedup.

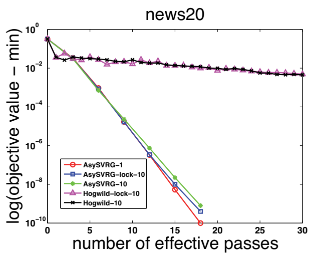
As mentioned above, besides the computation cost mainly reflected by the number of effective passes, there are other costs like communication cost and lock cost to affect the total CPU time in the multiple-thread case. We also compare the total CPU time between AsySVRG and Hogwild! both of which are with 10 threads. Since different hardware would lead to different CPU time, we use relative time for comparison. More specifically, we assume the time that Hogwild!-10 takes to get a sub-optimal solution with error $f(\mathbf{w}) - f(\mathbf{w}_*) < 0.01$ to be one (unit). Here, we use $f(\mathbf{w}) - f(\mathbf{w}_*) < 0.01$ rather than $f(\mathbf{w}) - f(\mathbf{w}_*) < 10^{-4}$ because Hogwild!-10 cannot achieve the accuracy of $f(\mathbf{w}) - f(\mathbf{w}_*) < 10^{-4}$ in some datasets. The convergence result with respect to total CPU time is shown in Figure 2. It is easy to see that AsySVRG has almost linear convergence rate, which is much faster than Hogwild!.



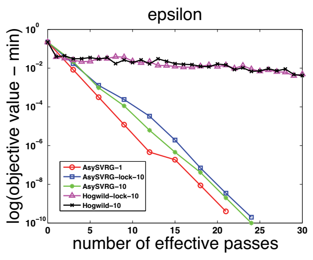
(a) rcv1



(b) realsim

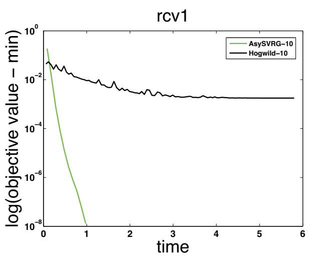


(c) news20

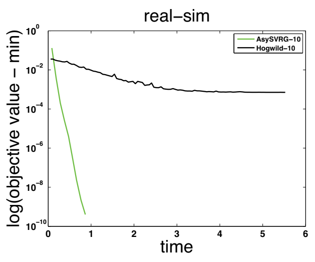


(d) epsilon

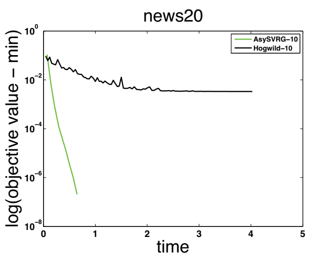
Figure 1: Convergence rate with respect to the number of effective passes (the vertical axis is in a log scale). Please note that in (a) and (b), the curves of AsySVRG-lock-10 and AsySVRG-10 overlap with each other.



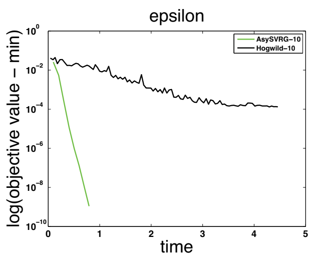
(a) rcv1



(b) realsim



(c) news20



(d) epsilon

Figure 2: Convergence rate with respect to CPU time (the vertical axis is in a log scale, and the horizonal axis is the ratio to the CPU time of Hogwild!-10 with the stopping condition $f(\mathbf{w}) - f(\mathbf{w}_*) < 0.01$).

Speedup We compare between AsySVRG-lock and the lock-free AsySVRG in terms of speedup to demonstrate the advantage of lock-free strategy. The definition of speedup is as follows:

$$speedup = \frac{CPU\ time\ with\ 1\ thread}{CPU\ time\ with\ p\ threads}.$$

Here, the stopping condition is $f(\mathbf{w}) - f(\mathbf{w}_*) < 10^{-4}$. The speedup results are shown in Figure 3, where we can find that our lock-free AsySVRG achieves almost a linear speedup. However, the speedup of AsySVRG-lock is much worse than AsySVRG. The main reason is that besides the computation cost, AsySVRG-lock need extra lock cost compared with AsySVRG.

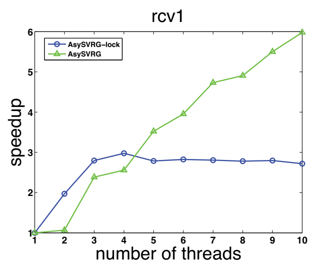
Conclusion

In this paper, we have proposed a novel asynchronous parallel SGD method with a lock-free strategy, called AsySVRG, for multicore systems. Both theoretical and empirical results show that AsySVRG can outperform other state-of-the-art methods.

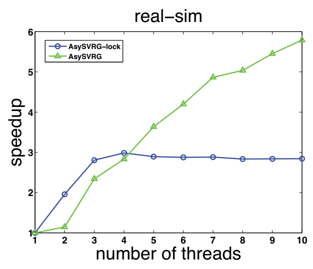
Future work will design asynchronous distributed SVRG methods on clusters of multiple machines by using similar techniques in AsySVRG.

Acknowledgements

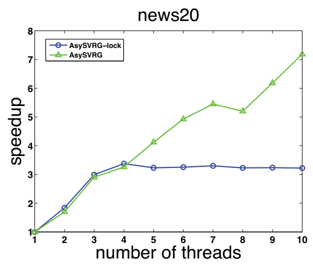
This work is supported by the NSFC (No. 61472182), and the Fundamental Research Funds for the Central Universities (No. 20620140510).



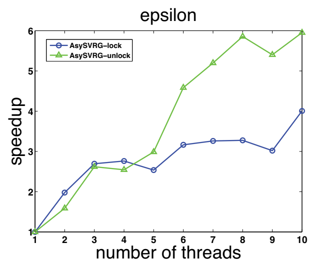
(a) rcv1



(b) realsim



(c) news20



(d) epsilon

Figure 3: Speedup results.

References

Duchi, J. C.; Agarwal, A.; and Wainwright, M. J. 2010. Distributed dual averaging in networks. In *Proceedings of the Advances in Neural Information Processing Systems*, 550–558.

Duchi, J. C., and Singer, Y. 2009. Efficient online and batch learning using forward backward splitting. *Journal of Machine Learning Research* 10:2899–2934.

Hsieh, C.; Yu, H.; and Dhillon, I. S. 2015. Passcode: Parallel asynchronous stochastic dual co-ordinate descent. In *Proceedings of the 32nd International Conference on Machine Learning*, 2370–2379.

Jaggi, M.; Smith, V.; Takac, M.; Terhorst, J.; Krishnan, S.; Hofmann, T.; and Jordan, M. I. 2014. Communication-efficient distributed dual coordinate ascent. In *Proceedings of the Advances in Neural Information Processing Systems*. 3068–3076.

Johnson, R., and Zhang, T. 2013. Accelerating stochastic gradient descent using predictive variance reduction. In *Proceedings of the Advances in Neural Information Processing Systems*, 315–323.

Liu, J.; Wright, S. J.; Ré, C.; Bittorf, V.; and Sridhar, S. 2014. An asynchronous parallel stochastic coordinate descent algorithm. In *Proceedings of the 31th International Conference on Machine Learning*, 469–477.

Mairal, J. 2013. Optimization with first-order surrogate functions. In *Proceedings of the 30th International Conference on Machine Learning*, 783–791.

Nitanda, A. 2014. Stochastic proximal gradient descent with acceleration techniques. In *Proceedings of the Advances in Neural Information Processing Systems*, 1574–1582.

Recht, B.; Re, C.; Wright, S.; and Niu, F. 2011. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the Advances in Neural Information Processing Systems*, 693–701.

Roux, N. L.; Schmidt, M. W.; and Bach, F. 2012. A stochastic gradient method with an exponential convergence rate for finite training sets. In *Proceedings of the Advances in Neural Information Processing Systems*, 2672–2680.

Shalev-Shwartz, S., and Zhang, T. 2013. Stochastic dual coordinate ascent methods for regularized loss. *Journal of Machine Learning Research* 14(1):567–599.

Xiao, L. 2009. Dual averaging method for regularized stochastic learning and online optimization. In *Proceedings of the Advances in Neural Information Processing Systems*, 2116–2124.

Zhang, R., and Kwok, J. T. 2014. Asynchronous distributed ADMM for consensus optimization. In *Proceedings of the 31th International Conference on Machine Learning*, 1701–1709.

Zhang, R.; Zheng, S.; and T. Kwok, J. 2015. Fast distributed asynchronous sgd with variance reduction. *arXiv:1508.01633*.

Zhao, S.-Y., and Li, W.-J. 2015. Fast asynchronous parallel stochastic gradient decent. *arXiv:1508.05711*.

Zinkevich, M.; Weimer, M.; Smola, A. J.; and Li, L. 2010. Parallelized stochastic gradient descent. In *Proceedings of the Advances in Neural Information Processing Systems*, 2595–2603.

Zinkevich, M.; Smola, A. J.; and Langford, J. 2009. Slow learners are fast. In *Proceedings of the Advances in Neural Information Processing Systems*, 2331–2339.