Doubly Sparse Asynchronous Learning for Stochastic Composite Optimization

Runxue Bao, **Xidong Wu**, **Wenhan Xian** and **Heng Huang**[*]
Electrical and Computer Engineering Department, University of Pittsburgh, PA, USA
{runxue.bao, xidong_wu, wex37, heng.huang}@pitt.edu

Abstract

Parallel optimization has become popular for large-scale learning in the past decades. However, existing methods suffer from huge computational cost, memory usage, and communication burden in high-dimensional scenarios. To address the challenges, we propose a new accelerated doubly sparse asynchronous learning (DSAL) method for stochastic composite optimization, under which two algorithms are proposed on shared-memory and distributed-memory architecture respectively, which only conducts gradient descent on the nonzero coordinates (data sparsity) and active set (model sparsity). The proposed algorithm can converge much faster and achieve significant speedup by simultaneously enjoying the sparsity of the model and data. Moreover, by sending the gradients on the active set only, communication costs are dramatically reduced. Theoretically, we prove that the proposed method achieves the linear convergence rate with lower overall complexity and can achieve the model identification in a finite number of iterations almost surely. Finally, extensive experimental results on benchmark datasets confirm the superiority of our proposed method.

1 Introduction

Big data with massive samples and numerous features recently widely exists in many real-world machine learning tasks, which often contains superfluous features [Gui and Li, 2005]. In the past decades, many models with sparse regularization have achieved great successes in high-dimensional scenarios by encouraging the model sparsity [Tibshirani, 1996; Ng, 2004; Yuan and Lin, 2006; Bao *et al.*, 2019]. Let $A = [a_1, \cdots, a_n]^\top \in \Re^{n\times p}$, in this paper, we consider the following composite optimization problem:

$$\min_{x\in\Re^p} \mathcal{P}(x) := \mathcal{F}(x) + \lambda\Omega(x), \tag{1}$$

where x is the model coefficient, $\Omega(x)$ is the block-separable sparsity-inducing norm, $\mathcal{F}(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(a_i^\top x)$ is the data-fitting loss, and λ is the regularization parameter. We denote $\mathcal{F}_i(x) = f_i(a_i^\top x)$ for simplicity. Given partition $\mathcal{G}$ of the coefficients, we denote the sub-matrix of A with the columns of $\mathcal{G}_j$ as $A_j \in \Re^{n\times|\mathcal{G}_j|}$ and have $\Omega(x) = \sum_{j=1}^{q}\Omega_j(x_{\mathcal{G}_j})$.

Inspired by the emerging parallel computing systems such as multi-core computer and distributed architecture, parallel learning has been actively studied in machine learning community due to its capability for tackling large-scale applications [Wang *et al.*, 2012; Wu *et al.*, 2013]. In literature, many parallel learning methods have been proposed to accelerate different optimization algorithms on shared-memory architecture [Langford *et al.*, 2009; Recht *et al.*, 2011; Reddi *et al.*, 2015; Zhao and Li, 2016; Leblond *et al.*, 2017; Meng *et al.*, 2017; Pedregosa *et al.*, 2017; Zhou *et al.*, 2018; Leblond *et al.*, 2018; Nguyen *et al.*, 2018; Joulani *et al.*, 2019; Stich *et al.*, 2021] and distributed-memory architecture [Agarwal and Duchi, 2012; Dean *et al.*, 2012; Zhang and Kwok, 2014; Lian *et al.*, 2015; Zhang *et al.*, 2016]. For smooth optimization problems, asynchronous stochastic methods, e.g., Hogwild! [Recht *et al.*, 2011], Lock-Free SVRG [Reddi *et al.*, 2015], AsySVRG [Zhao and Li, 2016], and ASAGA [Leblond *et al.*, 2017], were proposed. Furthermore, AsyProxSVRG [Meng *et al.*, 2017], ProxASAGA [Pedregosa *et al.*, 2017], and AsyMiG [Zhou *et al.*, 2018] were proposed to solve non-smooth composite optimization problems. However, most existing methods focus on improving the algorithm efficiency in terms of sample complexity.

To solve the problems with huge samples and features, an asynchronous doubly stochastic method was proposed in [Meng *et al.*, 2017], which performs coordinate updates without considering any sparsity and thus can be very slow. Moreover, [Leblond *et al.*, 2017] proposed an asynchronous sparse incremental gradient method, which does not enjoy the sparsity of the model to accelerate the training and thus huge requirement of computational complexity hinders its application on large-scale learning. Therefore, existing parallel methods still suffer huge burden for computation, memory, and communication costs to solve high-dimensional models.

In high-dimensional setting, sparsity is a key property that can be exploited to accelerate the training. Sparsity can be broken down into two aspects: data sparsity and model sparsity. In terms of the data sparsity, the dataset usually has a huge number of features in which most of the elements are zero. If the feature representation of the current sam-

[*]This work was partially supported by NSF IIS 1845666, 1852606, 1838627, 1837956, 1956002, 2040588.

Reference	Model	Dynamic	Safe	Distributed	Scalablity	Data Sparsity
PE-AGCD [Li *et al.*, 2016a]	Lasso	No	No	No	No	No
DSAL (Ours)	Problem (1)	Yes	Yes	Yes	Yes	Yes

Table 1: Comparison between PE-AGCD and our DSAL method. "Distributed" represents whether it can work on distributed-memory architecture. "Data Sparsity" represents whether it can benefit from the sparsity of data.

ple is zero, the stochastic gradient *w.r.t.* the corresponding feature must be zero. It is an effective method to accelerate the training by avoiding the useless updates of these gradients. On the other aspect, the regularization usually enforces the model sparsity where the coefficients of the associated features are zeroes at the optimum. The model sparsity can be exploited to reduce the problem dimensionality by pre-identifying inactive features [Ndiaye *et al.*, 2017; Bao *et al.*, 2020]. Discarding these features can save much computation without any loss of accuracy. Therefore, accelerating high-dimensional models by exploiting the sparsity is promising and sorely needed for large-scale problems.

To address the challenges above, in this paper, we propose a novel accelerated doubly sparse asynchronous learning (DSAL) method for stochastic composite optimization and apply it on shared-memory (Sha-DSAL) and distributed-memory (Dis-DSAL) architecture, which can accelerate the training significantly without any loss of accuracy. Specifically, DSAL not only conducts the elimination to discard inactive features to enjoy the model sparsity, but also conducts sparse proximal gradient update with the nonzero partial gradients to enjoy the data sparsity. To further accelerate DSAL, we reduce the gradient variance over the active set and thus can utilize a constant step size to achieve the linear convergence. On distributed-memory system, to reduce the computational burden of the server, we utilize a decouple strategy to improve the scalability of DSAL *w.r.t.* the number of workers.

Contributions. The main contributions of our work are summarized as follows:

- We propose a new doubly sparse asynchronous stochastic gradient method for stochastic composite optimization, which is easy-to-implement on the both shared-memory and distributed-memory architecture. To the best of our knowledge, this is the first work of asynchronous stochastic gradient method to simultaneously enjoy the model sparsity and data sparsity.
- We rigorously prove the DSAL method can achieve a linear convergence rate $O(\log(1/\epsilon))$, reduce the per-iteration cost from $O(p)$ to $O(r)$ where $r \ll p$, and achieve a lower overall computational complexity under the strongly convex condition. Moreover, we prove almost sure finite time identification of the active set.
- We empirically show that our proposed method can simultaneously achieve significant acceleration and linear speedup property.

1.1 Related Works

Parallel method (PE-AGCD) in [Li *et al.*, 2016a] handles the high-dimensional setting by conducting the parallel elimination (PE) step before Asynchronous Grouped Coordinate Descent (AGCD). However, the proposed method has several disadvantages. First, PE is only conducted once prior to the optimization algorithm. Second, only safe static rule [Ghaoui *et al.*, 2010] used in PE is safe and the strong rule [Tibshirani *et al.*, 2012] used in PE is unsafe, which means it could discard features wrongly. Third, PE-AGCD is deterministic on samples and thus cannot scale well for large n. Fourth, PE-AGCD cannot enjoy the sparsity of the data. Lastly, PE-AGCD is limited to Lasso regression. Table 1 summarizes the advantages of our DSAL over PE-AGCD. First, DSAL is dynamic and conducted during the whole training process. Thus, DSAL can accelerate and meanwhile benefit from the convergence of the optimization algorithm. Second, our DSAL is safe for the training. Third, DSAL is stochastic and can scale well on both samples and features. Fourth, DSAL can enjoy the data sparsity by performing sparse proximal updates. Lastly, DSAL can solve Problem (1).

2 Proposed Method

We first propose the accelerated doubly sparse asynchronous learning (DSAL) method and then apply it to shared-memory and distributed-memory architecture respectively.

2.1 Doubly Sparse Asynchronous Learning

To enjoy the sparsity of the model coefficient, we first give a naive implementation of DSAL on shared-memory architecture in Alg. 1. During the training, DSAL only need solve a sub-problem with constantly decreasing size by discarding useless features. Specifically, Alg. 1 has two loops. We denote the original problem as $\mathcal{P}_0$ and the full set as $\mathcal{B}_0$. At the s-th iteration of the outer loop, we denote the active set as $\mathcal{B}_s$ and suppose $\mathcal{B}_s$ has q_s active blocks with total p_s active features. Then we can compute the dual y^s as

$$y^s = -\nabla \mathcal{F}(x^0_{\mathcal{B}_s}) / \max(1, \Omega^D(A^\top_{\mathcal{B}_s} \nabla \mathcal{F}(x^0_{\mathcal{B}_s}))/\lambda), \qquad (2)$$

where dual norm $\Omega^D(u) = \max_{\Omega(v)\leq 1} \langle v, u\rangle$. Then, for $\forall j \in \mathcal{B}_s$, $\mathcal{B}_{s+1}$ is updated by:

$$\Omega^D_j(A_j^\top y^s) + \Omega^D_j(A_j)\sqrt{2L(\mathcal{P}(x^0_{\mathcal{B}_s}) - D(y^s))} \geq n\lambda, \qquad (3)$$

where L is the Lipschitz constant and $D(y^s)$ is the dual objective of $\mathcal{P}_s$ ([Ndiaye *et al.*, 2017; Bao *et al.*, 2020]).

For the inner loop, all the updates are conducted on $\mathcal{B}_{s+1}$. First, each thread inconsistently reads $\hat{x}^t_{\mathcal{B}_{s+1}}$ from the shared memory and randomly chooses sample i to compute the stochastic gradient on $\mathcal{B}_{s+1}$. Then, the proximal step is conducted with the stochastic gradient. By (3), we can constantly reduce the model size and the parameter size to accelerate the training by exploiting the sparsity of the model. Since each variable x_i discarded by the elimination must be zero for the optimum solution, this method is safe for the training.

Algorithm 1 Sha-DSAL-Naive

```
1:  Input: $x^0_{\mathcal{B}_0} \in \Re^p$, step size $\eta$, inner loops $K$.
2:  for $s = 0$ to $S-1$ do
3:      All threads parallelly compute $\nabla\mathcal{F}(x^0_{\mathcal{B}_s})$
4:      Compute $y^s$ by (2) and $\mathcal{B}_{s+1}$ from $\mathcal{B}_s$ by (3)
5:      Update $A_{\mathcal{B}_{s+1}}, x^0_{\mathcal{B}_{s+1}}$
6:      For each thread, do:
7:      for $t = 0$ to $K-1$ do
8:          Read $\hat{x}^t_{\mathcal{B}_{s+1}}$ from the shared memory
9:          Randomly sample $i$ from $\{1, 2, \ldots, n\}$
10:         $v^s_t = \nabla f_i(a^\top_{i,\mathcal{B}_{s+1}} \hat{x}^t_{\mathcal{B}_{s+1}})$
11:         Update $x^{t+1}_{\mathcal{B}_{s+1}} = \text{prox}_{\eta\lambda\Omega}(\hat{x}^t_{\mathcal{B}_{s+1}} - \eta v^s_t)$
12:     end for
13:     $x_{\mathcal{B}_{s+1}} = x^K_{\mathcal{B}_{s+1}}, x^0_{\mathcal{B}_{s+1}} = x_{\mathcal{B}_{s+1}}$
14: end for
```

Variance Reduction on the Active Set

However, the variance of the gradient in Alg. 1 caused by stochastic sampling does not converge to zero. Thus, we have to use a diminishing step size and can only attain a sublinear convergence rate when $\mathcal{P}$ is strongly convex.

Since the full gradient has been computed for the elimination step, inspired by [Xiao and Zhang, 2014], we can adopt the variance-reduced technique over active set $\mathcal{B}_{s+1}$ without additional computational costs. Thus, we can adjust the gradient estimation as

$$v^s_t = \nabla f_i(a^\top_{i,\mathcal{B}_{s+1}} \hat{x}^t_{\mathcal{B}_{s+1}}) - \nabla f_i(a^\top_{i,\mathcal{B}_{s+1}} x^0_{\mathcal{B}_{s+1}}) + \nabla\mathcal{F}(x^0_{\mathcal{B}_{s+1}}), \quad (4)$$

which can guarantee that the variance of stochastic gradients asymptotically converges to zero. Therefore, we can use a constant step size to achieve a linear convergence rate for strongly convex function.

Sparse Proximal Gradient Update

In practice, sparsity widely exists in large-scale datasets. To utilize the data sparsity, we only need to update the blocks with nonzero partial gradients. Thus, some blocks might be updated for more times while others for less times. Inspired by [Leblond *et al.*, 2017] for Proximal SAGA, we define a block-wise reweighting matrix to make a weighted projection on the blocks. Specifically, we define Ψ_i as the set of blocks that intersect the nonzero coefficients of ∇f_i. Let $n_\mathcal{G}$ be the number of occurrences that $\mathcal{G} \in \Psi_i$, if $n_\mathcal{G} > 0$, we define $d_\mathcal{G} = n/n_\mathcal{G}$. Otherwise, we ignore that block directly. Thus, we can define diagonal matrix $[D_i]_{\mathcal{G},\mathcal{G}} = d_\mathcal{G} I_{|\mathcal{G}|}$ for each block i and the gradient over $\mathcal{B}_{s+1}$ can be formulated as

$$v^s_t = \nabla f_i(a^\top_{i,\mathcal{B}_{s+1}} \hat{x}^t_{\mathcal{B}_{s+1}}) - \nabla f_i(a^\top_{i,\mathcal{B}_{s+1}} x^0_{\mathcal{B}_{s+1}}) + D_{i,\mathcal{B}_{s+1}} \nabla\mathcal{F}(x^0_{\mathcal{B}_{s+1}}). \quad (5)$$

Thus, we only need conduct a sparse update over the active set and the computational cost is further reduced.

On the other hand, the proximal operator of Problem (1) needs to update all the coordinates for each iteration. Considering the data sparsity again, we only need to update the blocks with nonzero partial gradients. Thus, we use a block-wise weighted norm $\phi_i(x) = \sum_{\mathcal{G}\in\Psi_i} d_\mathcal{G}\Omega_\mathcal{G}(x)$ to displace $\Omega(x)$ where $\mathbb{E}\phi_i(x) = \Omega(x)$. Thus, the new sparse proximal operator can be computed as

$$\text{prox}_{\eta\lambda\phi_i}(x') = \arg\min_x \frac{1}{2\eta}\|x - x'\|^2 + \lambda\phi_i(x). \quad (6)$$

Since we only need to update the blocks in Ψ_i, which could be much less than the full pass, we can save much computation and memory cost here. To sum it up, we can conduct the sparse gradient update and sparse proximal operator to accelerate the training by enjoying the sparsity of the data.

Decoupled Proximal Update

On distributed-memory architecture, multiple workers compute the gradients and send them to the server. The server computes the proximal operator. When the proximal step is time-consuming, doing this in the server would be the computational bottleneck of the whole algorithm. [Li *et al.*, 2016b] proposed a decoupled method to off-load the computations of the proximal step to workers. Thus, the server only does simple addition computation, which can achieve a sublinear converge rate and perform better than the coupled method.

To relieve the computation cost of the server in our algorithm, the proximal mapping step is computed by workers and the server only needs to do the element-wise computation. The workers conduct the proximal operator as $x^{t+1}_{\mathcal{B}_{s+1}} = \text{prox}_{\eta\lambda\phi_i}(x^t_{\mathcal{B}_{s+1}} - \eta v^s_t)$ and send the difference $\delta^s_t = \text{prox}_{\eta\lambda\phi_i}(x^{d(t)}_{\mathcal{B}_{s+1}} - \eta v^s_t) - x^{d(t)}_{\mathcal{B}_{s+1}}$ between the parameter $x^{d(t)}_{\mathcal{B}_{s+1}}$ and the output of the proximal operator to the server. Therefore, the server only does simple addition computation, which makes the algorithm suitable to parallelize to achieve linear speedup property and can be accelerated via increasing the number of workers.

2.2 DSAL on Shared-Memory and Distributed-Memory Architecture

On shared-memory architecture, our Sha-DSAL algorithm is in Alg. 2. Suppose we have l cores, in the outer loop, all threads parallelly compute $\nabla\mathcal{F}(x^0_{\mathcal{B}_s})$ and y^s, and perform the elimination. With new set $\mathcal{B}_{s+1}$, we update $A_{\mathcal{B}_{s+1}}$, $x^0_{\mathcal{B}_{s+1}}$, and $\nabla\mathcal{F}(x^0_{\mathcal{B}_{s+1}})$. In the inner loop, multiple threads update the parameter over $\mathcal{B}_{s+1}$ asynchronously, which means the parameter can be read and written without locks. Specifically, each thread inconsistently read $\hat{x}^t_{\mathcal{B}_{s+1}}$ from the shared memory and choose sample i. As (5), we compute the gradient v^s_t over $\mathcal{B}_{s+1}$. Then we conduct the proximal step, compute the update δ^s_t, and add it to the shared memory.

Notably, first, at the s-th iteration, by exploiting the sparsity of the model, our Alg. 2 only solve sub-problem $\mathcal{P}_{s+1}$ over $\mathcal{B}_{s+1}$, which is much more efficient than training the full model. The full gradients at the s-th iteration in our algorithm is only computed with p_s coefficients, which is much less than $O(p)$ in practice. Second, by exploiting the data sparsity, we only conduct sparse gradient update and sparse proximal update, which is very efficient for large-scale dataset. Third, by reducing the gradient variance, we can use a constant step size to achieve a linear convergence rate. Lastly, all

Algorithm 2 Sha-DSAL

```
1:  Input: $x^0_{\mathcal{B}_0} \in \Re^p$, step size $\eta$, inner loops $K$.
2:  for $s = 0$ to $S-1$ do
3:      All threads parallelly compute $\nabla\mathcal{F}(x^0_{\mathcal{B}_s})$
4:      Compute $y^s$ by (2) and $\mathcal{B}_{s+1}$ from $\mathcal{B}_s$ by (3)
5:      Update $A_{\mathcal{B}_{s+1}}, x^0_{\mathcal{B}_{s+1}}, \nabla\mathcal{F}(x^0_{\mathcal{B}_{s+1}})$
6:      For each thread, do:
7:      for $t = 0$ to $K-1$ do
8:          Read $\hat{x}^t_{\mathcal{B}_{s+1}}$ from the shared memory
9:          Randomly sample $i$ from $\{1, 2, \ldots, n\}$
10:         Compute $v^s_t$ by (5)
11:         $\delta^s_t = \text{prox}_{\eta\lambda\phi_i}(\hat{x}^t_{\mathcal{B}_{s+1}} - \eta v^s_t) - \hat{x}^t_{\mathcal{B}_{s+1}}$
12:         $x^{t+1}_{\mathcal{B}_{s+1}} = x^t_{\mathcal{B}_{s+1}} + \delta^s_t$
13:     end for
14:     $x_{\mathcal{B}_{s+1}} = x^K_{\mathcal{B}_{s+1}}, x^0_{\mathcal{B}_{s+1}} = x_{\mathcal{B}_{s+1}}$
15: end for
```

the threads work asynchronously, which is very easy to parallelize. Hence, the computation, memory, and communication costs of large-scale training can be effectively reduced.

On distributed-memory architecture, our Dis-DSAL algorithm is summarized in Alg. 3 and 4. Suppose we have one server node and l local worker nodes where each worker stores n_k samples. When the flag is True, in the outer loop of the server, the server broadcasts the flag and $x^0_{\mathcal{B}_s}$ to the workers. At the worker node, worker k receives $x^0_{\mathcal{B}_s}$ from the server, computes the gradient over n_k samples, and then sends them to the server. With the gradients received from all the workers, the server computes the full gradients and send them to the workers. Then, the server computes y^s, performs the elimination to obtain $\mathcal{B}_{s+1}$, and then send it to the workers. The worker receives $\nabla\mathcal{F}(x^0_{\mathcal{B}_s})$ and $\mathcal{B}_{s+1}$ from the server. With $\mathcal{B}_{s+1}$, the worker updates $A_{i\in n_k,\mathcal{B}_{s+1}}, x^0_{\mathcal{B}_{s+1}}$ and $\nabla\mathcal{F}(x^0_{\mathcal{B}_{s+1}})$. When the flag is False, the server broadcasts the flag to the workers. At the workers, the algorithm optimizes over $\mathcal{B}_{s+1}$. Multiple workers update the parameter asynchronously. Specifically, each worker receives stale parameter $x^{d(t)}_{\mathcal{B}_{s+1}}$ from the server. The worker first chooses sample i from $\{1, 2, \ldots, n_k\}$ and compute v^s_t over $\mathcal{B}_{s+1}$. Following the decoupling strategy, we compute the proximal step and the update at the worker node. Finally, the worker send it to server. In the inner loop of the server, $x^{t+1}_{\mathcal{B}_{s+1}}$ is updated by δ^s_t from workers via only simple addition computation.

Similar to Alg. 2, our Alg. 3 and 4 is computationally efficient by exploiting the sparsity of the model and the dataset and reducing the gradient variance. Moreover, the time-consuming proximal step from the server is off-loaded to all the workers, which is easy to parallelize. Overall, the computation, memory, and communication costs could be effectively reduced.

3 Theoretical Analysis

We provide the rigorous theoretical analysis for DSAL on shared-memory architecture, which can be easily extended to distributed memory. The proof is provided in the appendix.

Algorithm 3 Dis-DSAL (Server Node)

```
1:  for $s = 0$ to $S-1$ do
2:      flag = True
3:      Broadcast flag and $x^0_{\mathcal{B}_s}$ to all workers
4:      Receive gradients from all workers
5:      $\nabla\mathcal{F}(x^0_{\mathcal{B}_s}) = \frac{1}{n}\sum_{k=1}^{l}\nabla\mathcal{F}_k(x^0_{\mathcal{B}_s})$
6:      Compute $y^s$ by (2) and update $\mathcal{B}_{s+1} \subseteq \mathcal{B}_s$ by (3)
7:      Broadcast $\mathcal{B}_{s+1}$ and $\nabla\mathcal{F}(x^0_{\mathcal{B}_s})$ to all workers
8:      flag = False
9:      Broadcast flag to all workers
10:     for $t = 0$ to $K-1$ do
11:         Receive $\delta^s_t$ from one worker
12:         Update $x^{t+1}_{\mathcal{B}_{s+1}} = x^t_{\mathcal{B}_{s+1}} + \delta^s_t$
13:     end for
14:     $x_{\mathcal{B}_{s+1}} = x^K_{\mathcal{B}_{s+1}}, x^0_{\mathcal{B}_{s+1}} = x_{\mathcal{B}_{s+1}}$
15: end for
```

Algorithm 4 Dis-DSAL (Worker Node k)

```
1:  if flag = True then
2:      Receive $x^0_{\mathcal{B}_s}$ from server
3:      Compute and send gradient $\nabla\mathcal{F}_k(x^0_{\mathcal{B}_s}) = \sum_{i\in n_k}\nabla f_i(a^\top_{i,\mathcal{B}_s} x^0_{\mathcal{B}_s})$
4:      Receive $\nabla\mathcal{F}(x^0_{\mathcal{B}_s})$ and $\mathcal{B}_{s+1}$ from server
5:      Update $A_{i\in n_k,\mathcal{B}_{s+1}}, x^0_{\mathcal{B}_{s+1}}, \nabla\mathcal{F}(x^0_{\mathcal{B}_{s+1}})$
6:  else
7:      Receive $x^{d(t)}_{\mathcal{B}_{s+1}}$ from server
8:      Randomly sample $i$ from $\{1, 2, \ldots, n_k\}$
9:      Compute $v^s_t = \nabla f_i(a^\top_{i,\mathcal{B}_{s+1}} x^{d(t)}_{\mathcal{B}_{s+1}}) - \nabla f_i(a^\top_{i,\mathcal{B}_{s+1}} x^0_{\mathcal{B}_{s+1}}) + D_{i,\mathcal{B}_{s+1}}\nabla\mathcal{F}(x^0_{\mathcal{B}_{s+1}})$
10:     Send $\delta_t = \text{prox}_{\eta\lambda\phi_i}(x^{d(t)}_{\mathcal{B}_{s+1}} - \eta v^s_t) - x^{d(t)}_{\mathcal{B}_{s+1}}$ to server
11: end if
```

3.1 Assumptions, Definitions, and Properties

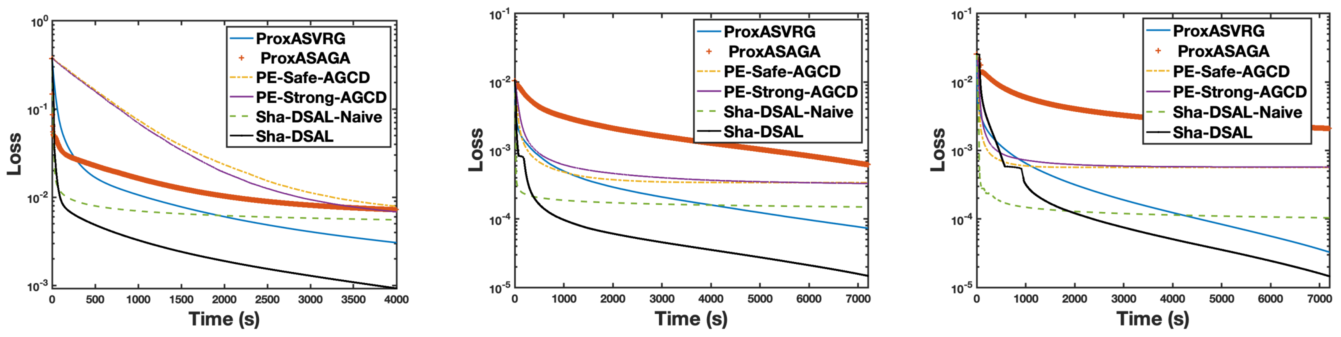
Assumption 1 (Strong Convexity)**.** *$\Omega(x)$ is convex and block separable. $\mathcal{F}(x)$ is μ-strongly convex, i.e., $\forall x, x' \in \Re^p$, we have $\mathcal{F}(x') \geq \mathcal{F}(x) + \nabla\mathcal{F}(x)^\top(x' - x) + \frac{\mu}{2}\|x' - x\|^2$.*

Assumption 2 (Lipschitz Smooth)**.** *Each $\mathcal{F}_i(x)$ is differentiable and Lipschitz gradient continuous with L, i.e., $\exists L > 0$, such that $\forall x, x' \in \Re^p$, we have $\|\nabla\mathcal{F}_i(x) - \nabla\mathcal{F}_i(x')\| \leq L\|x - x'\|$.*

Assumption 3 (Bounded Overlapping)**.** *There exists a bound τ on the number of iterations that overlap. The bound τ means each writing at iteration t is guaranteed to successfully performed into the memory before iteration $t + \tau + 1$.*

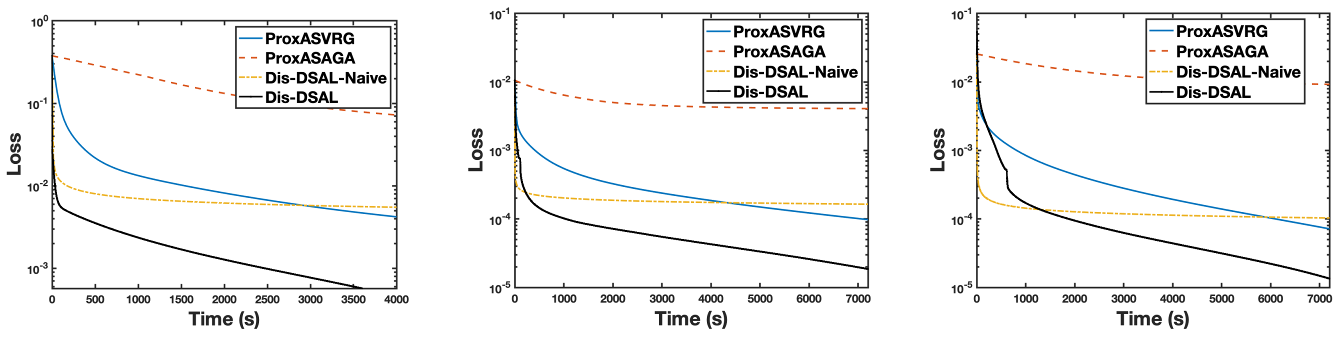
Remark 1. *Asm. 1 implies $\mathcal{P}(x)$ is also μ-strongly convex. Asm. 2 implies that $\mathcal{F}(x)$ is also Lipschitz gradient continuous. We denote $\kappa := L/\mu$ as the condition number. Asm. 3 means the delay that asynchrony may cause is upper bounded. All the assumptions are commonly seen in asynchronous methods [Pedregosa* et al.*, 2017].*

Definition 1 (Block Sparsity)**.** *We denote that the maximum frequency of occurrences Δ that a feature block belongs to the*



(a) KDD2010 (b) Avazu-app (c) Avazu-site

Figure 1: Convergence results on shared-memory architecture with 8 threads.



(a) KDD2010 (b) Avazu-app (c) Avazu-site

Figure 2: Convergence results on distributed-memory architecture with 8 workers.

extended support, which can be defined as $\Delta = \max_{\mathcal{G}\in\mathcal{B}} |\{i : \Psi_i \ni \mathcal{G}\}|/n$. *It is easy to verify that* $1/n \leq \Delta \leq 1$.

Property 1 (Independence). *We use the "after read" labeling in [Leblond* et al.*, 2017], which means we update the iterate counter after each thread fully reads the parameters. This means that* $\hat{x}^t_{\mathcal{B}_{s+1}}$ *is the* $(t+1)$*-th fully completed read. Given the "after read" global time counter, sample* i_r *is independent of* $\hat{x}^t_{\mathcal{B}_{s+1}}, \forall r \geq t$.

Property 2 (Unbiased Gradient Estimation). *Gradient* v_t *is an unbiased estimation of the gradient over set* $\mathcal{B}_{s+1}$ *at* $\hat{x}^t_{\mathcal{B}_{s+1}}$*, which is directly derived from Property 1.*

Property 3 (Atomic Operation). *The update* $x^{t+1}_{\mathcal{B}_{s+1}}$ *to shared memory in Alg. 2 is coordinate-wise atomic, which can address the overwriting problem caused by other threads.*

3.2 Theoretical Results

Theorem 1 (Convergence). *Suppose* $\tau \leq \frac{1}{10\sqrt{\Delta}}$*, let step size* $\eta = \min\{\frac{1}{24\kappa L}, \frac{\kappa}{2L}, \frac{\kappa}{10\tau L}\}$*, inner loop size* $K = \frac{4\log 3}{\eta\mu}$*, we have*

$$\mathbb{E}\left\|x_{\mathcal{B}_S} - x^*_{\mathcal{B}_S}\right\|^2 \leq (2/3)^S \left\|x_0 - x^*\right\|^2. \tag{7}$$

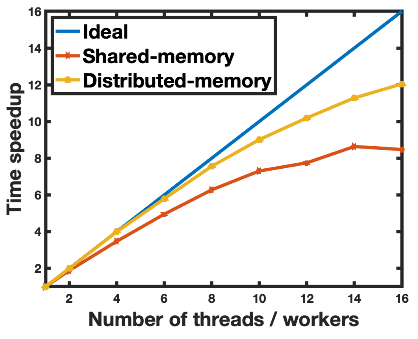
Remark 2. *Theorem 1 shows that DSAL can achieve a linear convergence rate* $O(\log(1/\epsilon))$.

Remark 3. *For the case* $\mathcal{F}(x)$ *is nonstrongly convex, we can slightly modify* $\Omega(x)$ *by adding a small perturbation, e.g.,* $\mu_f\|x\|^2$ *for smoothing where* $\mu_f > 0$*. We can treat* $\mathcal{F}(x) + \mu_f\|x\|^2$ *as the data-fitting loss and then the loss is* μ_f*-strongly convex. Denote* $\kappa := L/\mu_f$*, suppose* $\tau \leq \frac{1}{10\sqrt{\Delta}}$*, let* $\eta = \min\{\frac{1}{24\kappa L}, \frac{\kappa}{2L}, \frac{\kappa}{10\tau L}\}$*,* $K = \frac{4\log 3}{\eta\mu_f}$*, we have* $\mathbb{E}\left\|x_{\mathcal{B}_S} - x^*_{\mathcal{B}_S}\right\|^2 \leq (2/3)^S \left\|x_0 - x^*\right\|^2$.

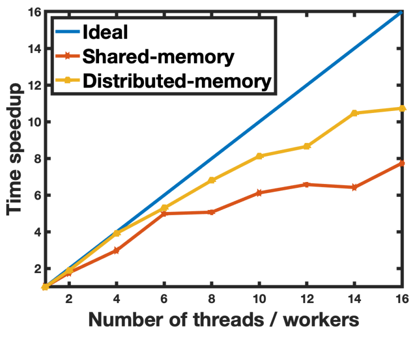
Theorem 2 (Elimination Ability). *Equicorrelation set [Tibshirani and others, 2013] is defined as* $\mathcal{B}^* := \{j \in \{1, 2, \ldots, q\} : \Omega^D_j(A_j^\top y^*) = n\lambda\}$*. As DSAL converges, there exists an iteration number* $S_0 \in \mathbb{N}$*, s.t.* $\forall s \geq S_0$*, any variable block* $j \notin \mathcal{B}^*$ *is eliminated by DSAL almost surely.*

Remark 4. *Suppose the size of set* $\mathcal{B}_s$ *is* p_s *and* p^* *is the size of the active features in* $\mathcal{B}^*$*, Theorem 2 shows we have* p_s *is decreasing and* $\lim_{s\to+\infty} p_s = p^*$.

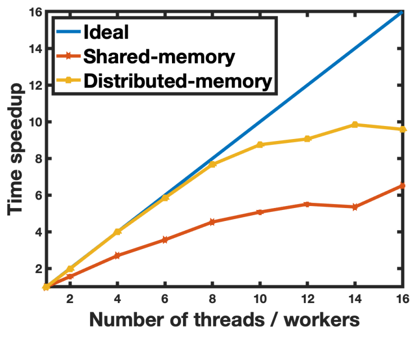
Remark 5. *To sum it up, by the elimination, the cost at each iteration is reduced from* $O(p)$ *to* $O(p_s)$*. Moreover, by the sparse update, only the nonzero coefficients of set* $\mathcal{B}_s$ *is updated and thus the cost at each iteration is further reduced from* $O(p_s)$ *to* $O(p'_s)$ *where* p'_s *is the size of the nonzero coefficients. In the high-dimensional setting, we have* $p^* \ll p$*,* $p_s \ll p$ *and* $p'_s \ll p$*. Thus, with constantly decreasing* p_s *and the sparse update, our DSAL can reduced the complexity from* $O(p)$ *to* $O(r)$ *where* r *is the mean of* p'_s *for* $s = 1, 2, \ldots$*, which can accelerate the training at a large extent in practice.*



(a) KDD2010



(b) Avazu-app



(c) Avazu-site

Figure 3: Convergence results on shared-memory architecture with 8 threads.

4 Experiments

4.1 Experimental Setup

We compare our method with competitive methods on three large-scale datasets. Although DSAL can work broadly, we focus on Lasso, which is the most popular case for sparse regression. Specifically, Lasso solves

$$\min_{x\in\Re^p} \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}(y_i - a_i^\top x)^2 + \lambda\|x\|_1. \tag{8}$$

On shared-memory architecture, we compare six asynchronous methods: 1) PE-Strong-AGCD: parallel strong elimination in [Li *et al.*, 2016a]; 2) PE-Safe-AGCD: parallel static safe elimination in [Li *et al.*, 2016a]; 3) ProxASAGA [Pedregosa *et al.*, 2017]; 4) ProxASVRG [Meng *et al.*, 2017]; 5) Sha-DSAL-Naive; 6) Our Sha-DSAL. ProxASAGA and ProxASVRG are popular asynchronous method with linear convergence. On distributed-memory architecture, we compare four asynchronous methods: 1) ProxASAGA [Leblond *et al.*, 2018]; 2) ProxASVRG [Meng *et al.*, 2017]; 3) Dis-DSAL-Naive (See appendix); 4) Our Dis-DSAL.

Dataset	Sample Size	Attributes	Sparsity
KDD 2010	19,264,097	1,163,024	7×10^{-6}
Avazu-app	14,596,137	1,000,000	10^{-5}
Avazu-site	25,832,830	1,000,000	10^{-5}

Table 2: Real-world datasets in the experiments.

We use three real-world datasets in Table 2, which are from LIBSVM [Chang and Lin, 2011] at https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/. We implement all the methods in C++. We employ OpenMP and OpenMPI as the parallel framework for shared-memory and distributed-memory architecture respectively. We run all the methods on 2.10 GHz Intel(R) Xeon(R) CPU machines. The inner loop size and the step size are chosen to obtain the best performance. Parameter λ is selected as $4*10^{-6}\lambda_{max}$, $2*10^{-3}\lambda_{max}$, and $1*10^{-3}\lambda_{max}$ for KDD 2010, Avazu-app, and Avazu-site dataset respectively where $\lambda_{\max}$ is a parameter that, for all $\lambda\geq\lambda_{\max}$, x^* must be 0.

4.2 Experimental Results and Discussions

Convergence Results. Figure 1 (a)-(c) provides the convergence results on shared-memory architecture. Our Sha-DSAL-Naive converges very fast at the initial stage because of the elimination ability and slows down later due to its sublinear rate. The results confirm that our Sha-DSAL always converge much faster than other methods. Figure 2 (a)-(c) provides the convergence results on distributed-memory architecture. The results also show that our Dis-DSAL always converge much faster than other methods. This is because our method can eliminate the features by exploiting the sparsity of the model, perform efficient sparse update by exploiting the data sparsity, achieve the linear convergence rate by reducing the gradient variance. Our Dis-DSAL also performs the decouple proximal update to reduce the workload of the server and reduces the communication costs.

Linear Speedup Property. Figure 3(a)-(c) presents the results of the speedup of Sha-DSAL with different number of threads on shared-memory architecture and Dis-DSAL with different number of workers on distributed-memory architecture. The results show our method can successfully achieve a nearly linear speedup when we increase the number of threads or workers, although the performance decreases when the number of processors or works increases. This is because theoretical analysis does not take the overheads for creating threads and distributing work for OpenMP and communication costs for OpenMPI into account.

5 Conclusion

In this paper, we propose the first doubly sparse asynchronous learning method for stochastic composite optimization and apply it on shared-memory and distributed-memory architecture respectively. Theoretically, we prove that our proposed method can achieve a linear convergence rate with lower overall complexity. Moreover, our method can eliminate almost all the inactive variables in a finite number of iterations almost surely. Finally, numerical results confirm the superiority of our method. Since we only consider the block-separable norm here, applying our method to non-separable norms, such as OWL regression [Bogdan *et al.*, 2015; Bao *et al.*, 2019], would be an interesting direction for future work.

References

[Agarwal and Duchi, 2012] Alekh Agarwal and John C Duchi. Distributed delayed stochastic optimization. In *IEEE CDC*, 2012.

[Bao *et al.*, 2019] Runxue Bao, Bin Gu, and Heng Huang. Efficient approximate solution path algorithm for ordered weighted l_1-norm with accuracy guarantee. In *IEEE ICDM*, 2019.

[Bao *et al.*, 2020] Runxue Bao, Bin Gu, and Heng Huang. Fast oscar and owl with safe screening rules. In *ICML*, 2020.

[Bogdan *et al.*, 2015] Małgorzata Bogdan, Ewout Van Den Berg, Chiara Sabatti, et al. Slope—adaptive variable selection via convex optimization. *Ann. Appl. Stat*, 2015.

[Chang and Lin, 2011] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM TIST*, 2011.

[Dean *et al.*, 2012] Jeffrey Dean, Greg Corrado, Rajat Monga, et al. Large scale distributed deep networks. In *NeurIPS*, 2012.

[Ghaoui *et al.*, 2010] Laurent El Ghaoui, Vivian Viallon, and Tarek Rabbani. Safe feature elimination in sparse supervised learning. Technical report, 2010.

[Gui and Li, 2005] Jiang Gui and Hongzhe Li. Penalized cox regression analysis in the high-dimensional and low-sample size settings, with applications to microarray gene expression data. *Bioinformatics*, 2005.

[Joulani *et al.*, 2019] Pooria Joulani, András György, and Csaba Szepesvari. Think out of the "box": Generically-constrained asynchronous composite optimization and hedging. *NeurIPS*, 2019.

[Langford *et al.*, 2009] John Langford, Alexander J Smola, and Martin Zinkevich. Slow learners are fast. In *NeurIPS*, 2009.

[Leblond *et al.*, 2017] Rémi Leblond, Fabian Pedregosa, and Simon Lacoste-Julien. Asaga: asynchronous parallel saga. In *AISTATS*, 2017.

[Leblond *et al.*, 2018] Rémi Leblond, Fabian Pedregosa, and Simon Lacoste-Julien. Improved asynchronous parallel optimization analysis for stochastic incremental methods. *JMLR*, 2018.

[Li *et al.*, 2016a] Qingyang Li, Shuang Qiu, Shuiwang Ji, et al. Parallel lasso screening for big data optimization. In *ACM SIGKDD*, 2016.

[Li *et al.*, 2016b] Yitan Li, Linli Xu, Xiaowei Zhong, and Qing Ling. Make workers work harder: decoupled asynchronous proximal stochastic gradient descent. *arXiv preprint arXiv:1605.06619*, 2016.

[Lian *et al.*, 2015] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *NeurIPS*, 2015.

[Meng *et al.*, 2017] Qi Meng, Wei Chen, Jingcheng Yu, et al. Asynchronous stochastic proximal optimization algorithms with variance reduction. In *AAAI*, 2017.

[Ndiaye *et al.*, 2017] Eugene Ndiaye, Olivier Fercoq, Alexandre Gramfort, et al. Gap safe screening rules for sparsity enforcing penalties. *JMLR*, 2017.

[Ng, 2004] Andrew Y Ng. Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *ICML*, 2004.

[Nguyen *et al.*, 2018] Lam Nguyen, Phuong Ha Nguyen, Marten Dijk, et al. Sgd and hogwild! convergence without the bounded gradients assumption. In *ICML*, 2018.

[Pedregosa *et al.*, 2017] Fabian Pedregosa, Rémi Leblond, and Simon Lacoste-Julien. Breaking the nonsmooth barrier: A scalable parallel method for composite optimization. In *NeurIPS*, 2017.

[Recht *et al.*, 2011] Benjamin Recht, Christopher Ré, Stephen J Wright, et al. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NeurIPS*, 2011.

[Reddi *et al.*, 2015] Sashank J Reddi, Ahmed Hefny, Suvrit Sra, et al. On variance reduction in stochastic gradient descent and its asynchronous variants. In *NeurIPS*, 2015.

[Stich *et al.*, 2021] Sebastian Stich, Amirkeivan Mohtashami, and Martin Jaggi. Critical parameters for scalable distributed learning with large batches and asynchronous updates. In *AISTATS*, 2021.

[Tibshirani and others, 2013] Ryan J Tibshirani et al. The lasso problem and uniqueness. *Electron. J. Stat.*, 2013.

[Tibshirani *et al.*, 2012] Robert Tibshirani, Jacob Bien, Jerome Friedman, et al. Strong rules for discarding predictors in lasso-type problems. *JRSS*, 2012.

[Tibshirani, 1996] Robert Tibshirani. Regression shrinkage and selection via the lasso. *JRSS*, 1996.

[Wang *et al.*, 2012] Gang Wang, Derek Hoiem, and David Forsyth. Learning image similarity from flickr groups using fast kernel machines. *IEEE TPAMI*, 2012.

[Wu *et al.*, 2013] Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding. Data mining with big data. *IEEE TKDE*, 2013.

[Xiao and Zhang, 2014] Lin Xiao and Tong Zhang. A proximal stochastic gradient method with progressive variance reduction. *SIOPT*, 2014.

[Yuan and Lin, 2006] Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *JRSS*, 2006.

[Zhang and Kwok, 2014] Ruiliang Zhang and James Kwok. Asynchronous distributed admm for consensus optimization. In *ICML*, 2014.

[Zhang *et al.*, 2016] Ruiliang Zhang, Shuai Zheng, and James T Kwok. Asynchronous distributed semi-stochastic gradient optimization. In *AAAI*, 2016.

[Zhao and Li, 2016] Shen-Yi Zhao and Wu-Jun Li. Fast asynchronous parallel stochastic gradient descent: A lock-free approach with convergence guarantee. In *AAAI*, 2016.

[Zhou *et al.*, 2018] Kaiwen Zhou, Fanhua Shang, and James Cheng. A simple stochastic variance reduced algorithm with fast convergence rates. In *ICML*, 2018.