# Asynchronous Distributed ADMM for Consensus Optimization

**Ruiliang Zhang** RZHANGAF@CSE.UST.HK
**James T. Kwok** JAMESK@CSE.UST.HK
Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong

## Abstract

Distributed optimization algorithms are highly attractive for solving big data problems. In particular, many machine learning problems can be formulated as the global consensus optimization problem, which can then be solved in a distributed manner by the alternating direction method of multipliers (ADMM) algorithm. However, this suffers from the straggler problem as its updates have to be synchronized. In this paper, we propose an asynchronous ADMM algorithm by using two conditions to control the asynchrony: partial barrier and bounded delay. The proposed algorithm has a simple structure and good convergence guarantees (its convergence rate can be reduced to that of its synchronous counterpart). Experiments on different distributed ADMM applications show that asynchrony reduces the time on network waiting, and achieves faster convergence than its synchronous counterpart in terms of the wall clock time.

## 1. Introduction

In this big data era, the data size is growing at an unprecedented scale. From videos in Youtube, security footage at airports to astronomical data collected at the large synoptic survey telescope, tons of data are being generated everyday everywhere. In a recent digital universe study by EMC, the world created about 1.8 zettabytes of data in 2011. Facebook alone, for example, is estimated to be creating 12 terabytes of data every day. The amount of data across the globe is also expected to double every two years, and will reach 35 zettabytes by 2020.

To alleviate this big data problem, the use of stochastic techniques has recently drawn a lot of interest. Most of them are based on variants of the stochastic gradient descent (Shalev-Shwartz et al., 2007). The idea is to replace the gradient over the whole data set by the gradient at a single sample (or over a small mini-batch of samples). Hence, its per-iteration complexity is much lower, and can scale to much larger data sets.

While the stochastic approach alleviates the big data problem by processing only a small sample subset in each iteration, an alternative is to use distributed processing. This is particularly natural for many big data applications, in which the data sets are too large to be stored or processed on one single computer. In distributed optimization algorithms, communication among the computing nodes is based on either shared memory (Niu et al., 2011) or distributed memory (Langford et al., 2009; Agarwal & Duchi, 2011; Ho et al., 2013; Li et al., 2013). In this paper, we will focus on algorithms using distributed memory, as they can often handle much larger data sets.

Consider minimizing a function $f(x)$ in a distributed computing environment with $N$ nodes. Assume that this function can be decomposed into $N$ components as

$$f(x) = \sum_{i=1}^{N} f_i(x), \tag{1}$$

where each $f_i$ is a local objective involving only the data subset residing on node $i$. This type of problems is often encountered in various areas such as machine learning, signal processing and wireless communication (Bertsekas & Tsitsiklis, 1989; Zhu et al., 2010). For example, in regularized risk minimization, $x$ is the model parameter to be estimated, and $f_i$ is the regularized risk functional defined on the data subset at node $i$.

The minimization of $f(x)$ can be reformulated as the following *global variable consensus optimization* problem (Boyd et al., 2011; Bertsekas & Tsitsiklis, 1989):

$$\min_{x_1,\dots,x_N,z} \sum_{i=1}^{N} f_i(x_i) \;:\; x_i = z,\; i = 1, 2, \dots, N, \tag{2}$$

where $z$ is the so-called consensus variable, and $x_i$ is node $i$'s local copy of the parameter to be learned. In a dis-

*Proceedings of the $31^{st}$ International Conference on Machine Learning*, Beijing, China, 2014. JMLR: W&CP volume 32. Copyright 2014 by the author(s).

tributed computing environment, this problem can be efficiently solved by the alternating direction method of multipliers (ADMM) algorithm (Boyd et al., 2011), which has been popularly used in various areas such as machine learning, computer vision and data mining. Essentially, one of the nodes, called the master, is responsible for updating the consensus variable $z$, while the remaining nodes are called workers. Each worker minimizes its local objective $f_i$ (in parallel) based on its data subset; and sends the updated local copy $x_i$ to the master. The master, in turn, updates $z$ by driving the $x_i$'s into consensus, and then distributes the updated value back to the workers, and the process re-iterates.

However, updates in this distributed ADMM algorithm have to be synchronized (Boyd et al., 2011). In other words, the master needs to wait for all the workers to finish their $x_i$ updates before it can proceed. This is at odds with the decentralized nature of distributed computing. Moreover, when the workers have different delays (because of difference in processing speeds, network delays, etc.), one has to wait for the slowest worker to complete its update before the next iteration can proceed. This problem of "straggler"[1] allows the system to move forward only at the pace of the slowest worker. Besides, if some processors fail, which is often not surprising in real-world data centers, a synchronous algorithm will come to an immediate halt.

In contrast to synchronous algorithms, asynchronicity allows more independence of the nodes, a more flexible design and is also more robust to individual node failures. Preliminary success of the asynchronous strategy has been recently demonstrated in (Langford et al., 2009; Agarwal & Duchi, 2011; Niu et al., 2011; Ho et al., 2013; Li et al., 2013), though they are mostly interested in distributed gradient descent methods and variants.

Motivated by these recent advances, we propose in this paper an asynchronous distributed ADMM algorithm for the global variable consensus optimization problem. There are two essential ingredients. (i) Instead of requiring full synchronization on all the workers in each ADMM iteration, a partial synchronization is only needed. (ii) While updates from the faster workers will be incorporated more often by the master, we require that updates from the slow workers cannot be older than a certain maximum delay.

The rest of this paper is organized as follows. Section 2 reviews existing works on synchronous and asynchronous distributed algorithms that are based on ADMM. Section 3 describes the proposed asynchronous distributed algorithm, with convergence analysis provided in Section 4. In particular, it is shown that when the proposed asynchronous algorithm reduces to a synchronous one, its convergence rate also reduces to the standard $\mathcal{O}(\frac{1}{T})$ rate for ADMM (He & Yuan, 2012). Finally, experiments on three different ADMM applications are presented in Section 5, and the last section gives some concluding remarks.

[1] This problem (sometimes called "the curse of the last reducer" (Suri & Vassilvitskii, 2011)) is also widely known in MapReduce, which requires a similar full synchronization in its reduce step.

## 2. Related Work

### 2.1. Synchronous Distributed Consensus ADMM

We start with the augmented Lagrangian of problem (2):

$$L(\{x_i\}, z, ) = \sum_{i=1}^{N} f_i(x_i) + \langle \lambda_i, x_i - z\rangle + \frac{\beta}{2}\|x_i - z\|^2,$$

where $\lambda_i$'s are the Lagrangian multipliers, $\beta > 0$ is the penalty parameter, and $\langle\cdot,\cdot\rangle$ denotes the inner product. At the $k$th iteration, the values of $x_i$ and $z$ (denoted $x_i^k$ and $z^k$) are updated by minimizing $L(\{x_i\}, z)$ w.r.t. $x_i$ and $z$. Unlike the method of multipliers, these are minimized in an alternating manner, which allows the problem to be more easily decomposed. The resulting ADMM update is (Boyd et al., 2011):

$$x_i^{k+1} = \arg\min_x f_i(x) + \langle \lambda_i^k, x\rangle + \frac{\beta}{2}\|x - z^k\|^2, \quad (3)$$

$$z^{k+1} = \arg\min_z \sum_{i=1}^{N} -\langle \lambda_i^k, z\rangle + \frac{\beta}{2}\|x_i^{k+1} - z\|^2, \quad (4)$$

$$\lambda_i^{k+1} = \lambda_i^k + \beta(x_i^{k+1} - z^{k+1}). \quad (5)$$

The above update can be easily implemented in a distributed system with one master and $N$ workers. Each worker $i$ is responsible for updating its $(x_i, \lambda_i)$ using (3) and (5). The updated $x_i^{k+1}$'s are then sent to the master, which is responsible for updating the consensus variable $z$ and distributing its updated value back to the workers. Note that as the $(x_i, \lambda_i)$'s are local to each worker, their updates can be performed by all the workers in parallel. However, they have to be synchronized in that the master has to wait for the $x_i$ updates from all the $N$ workers. This also necessitates the use of a global clock $k$. In the sequel, this distributed consensus ADMM algorithm will be called synchronous ADMM (sync-ADMM). The whole update procedures for the master and workers are shown in Algorithms 1 and 2, respectively. Recently, it has been shown that this can be well implemented in distributed computing environments such as MPI or MapReduce (Lubell-Doughtie & Sondag, 2013).

### 2.2. Decentralized Distributed ADMM

Recently, a number of related ADMM-based distributed algorithms have been proposed. They are decentralized in that there is no master, and the workers coordinate among

**Algorithm 1** Synchronous ADMM (sync-ADMM): Processing by the master.

1: initialize: $k = 0$.
2: **repeat**
3: **repeat**
4: wait;
5: **until** receive updates from all $N$ workers;
6: update $z^{k+1}$ by (4);
7: broadcast $z^{k+1}$ to all the workers;
8: $k \leftarrow k + 1$;
9: **until** termination;
10: **output** $z^k$.

**Algorithm 2** Synchronous ADMM (sync-ADMM): Processing by worker $i$.

1: initialize: $k = 0, \lambda_i^0 = 0$.
2: **repeat**
3: update $x_i^{k+1}$ using (3);
4: send $\lambda_i^k$ and $x_i^{k+1}$ to the master;
5: **repeat**
6: wait;
7: **until** receive the updated $z^{k+1}$ from the master;
8: update $\lambda_i^{k+1}$ using (5);
9: **until** termination.

themselves. For example, Mota et al. (2013) developed a communication-efficient distributed algorithm extended from the multi-block ADMM algorithm. A fixed sequence is used to define the order in which workers are updated. Wei & Ozdaglar (2012) proposed another decentralized ADMM algorithm, but again, the worker updates are sequential. To alleviate the order problem, Wei & Ozdaglar (2013) proposed the asynchronous ADMM algorithm, in which workers are partitioned into groups according to their interconnection pattern. At each iteration, one of the groups is randomly activated, and workers therein are allowed to update.

These algorithms are different from the proposed algorithm in several aspects. First, they are decentralized, while ours is a centralized algorithm which requires a master. Second, their asynchrony is in the sense that only a selected worker (or group of workers) is allowed to update at each iteration. However, this implicitly requires the maintenance of a global clock, and each group needs to be aware of each other's progress. Moreover, decentralized ADMM algorithms are highly dependent on the network topology. In this paper, we consider the star topology with one central node connecting to all the workers. Each decentralized ADMM iteration then has two steps: (i) the workers optimize their local objectives and send updates to the central node; (ii) the central node uses the workers' updates to optimize its local objective, and then broadcast the result. Similar to the sync-ADMM, the central node still needs to wait for all worker updates.

## 3. Distributed Asynchronous Consensus ADMM

In this section, we present the asynchronous distributed ADMM algorithm for the global variable consensus optimization problem. In the sequel, it will be simply called asynchronous ADMM (async-ADMM).

### 3.1. Master and Worker Clocks

As for the sync-ADMM in Section 2.1, the master is responsible for updating the consensus variable $z$, while each worker $i$ is responsible for updating the local primal variable $x_i$ and local dual variable $\lambda_i$. However, as the proposed algorithm is fully asynchronous, the master keeps a clock $k$, which starts from zero and is incremented by 1 after each $z$ update. Similarly, every worker also has its own clock $k_i$, which starts from zero and is incremented by 1 after each $\lambda_i$ update. All the clocks $k$ and $\{k_i\}_{i=1}^N$ are run independently. Let $x_i^{k_i}, \lambda_i^{k_i}$ be the values of $x_i$ and $\lambda_i$ when worker $i$'s clock is at $k_i$; and $z^k$ be the value of $z$ when the master's clock is at $k$.

### 3.2. Updating $x$ by the Worker

We first consider a particular worker $i$ (at time $k_i$). Using the most recent[2] $z$ value (denoted $\tilde{z}_i$) received by $i$ from the master, it updates its local copy $x_i$ analogous to (3), as

$$x_i^{k_i+1} = \arg\min_x \; f_i(x) + \langle \lambda_i^{k_i}, x \rangle + \frac{\beta}{2}\|x - \tilde{z}_i\|^2. \quad (6)$$

Moreover, as the workers have different speeds, the $\tilde{z}_i$'s are in general different. In other words, as in recent distributed asynchronous optimization algorithms (Langford et al., 2009; Agarwal & Duchi, 2011; Ho et al., 2013), some workers may be using out-of-date versions of the consensus variable. The new $x_i^{k_i+1}$, together with $\lambda_i^{k_i}$, are sent to the master. Worker $i$ then waits for the next $z$ update from the master before further processing (see Section 3.4).

### 3.3. Updating $z$ by the Master

The master waits for the workers' $\{(x_i, \lambda_i)\}$ updates before it can update $z$. Recall that for the sync-ADMM, this can proceed only after the $\{x_i\}$ updates from all $N$ workers have finished. In distributed systems, this mechanism is called a *barrier*, and is the simplest synchronization primitive (Albrecht et al., 2006). However, as discussed in Section 1, it suffers from the straggler problem and allows the system to move forward only at the pace of the slowest

[2] On initialization, the worker does not obtain the $z$ value from the master, and uses a default $z^0$ value instead.

worker. To alleviate this problem, we relax it to a *partial barrier* (Albrecht et al., 2006). Specifically, the master only needs to wait for a minimum of $S$ updates, where $S$ ($\geq 1$) can be much smaller than $N$. The synchronous ADMM can be regarded as using the extreme setting of $S = N$. Moreover, recall that some workers are using out-of-date versions of $z$. Consequently, their $(x_i, \lambda_i)$ updates are also out-of-date, an issue that the master has to cope with.

Besides, the master needs to wait for another precondition to be satisfied before it can proceed. Note that if we only rely on a partial barrier with small $S$, updates from the slow workers will be incorporated into $z$ much less often than those from the faster workers. To ensure sufficient freshness of all the updates, we enforce a *bounded delay* condition. Specifically, update from every worker has to be serviced by the master at least once every $\tau$ iterations, where $\tau \geq 1$ is a user-defined parameter. In other words, the $(x_i, \lambda_i)$ update from every worker $i$ can at most be $\tau$ clock cycles old (according to the master's clock). In the implementation, a counter $\tau_i$ is kept by the master for each worker $i$. When $(x_i, \lambda_i)$ from worker $i$ arrives at the master, the corresponding $\tau_i$ is reset to 1; otherwise, $\tau_i$ is incremented by 1 as the master's clock $k$ increments.

Note that a similar idea has been used in the machine learning community. In (Langford et al., 2009; Agarwal & Duchi, 2011), a cyclic-delay architecture is used in which workers communicate with the master or each other with fixed numbers of delayed cycles. This is also similar to *bounded staleness* in (Ho et al., 2013), though in (Ho et al., 2013) it is the worker that receives a possibly staled version of the parameter, while here it is the master that receives a possibly out-of-date $(x_i, \lambda_i)$ update, which in turn is computed using a possibly out-of-date consensus variable.

When both the partial barrier and bounded delay conditions are met, the master can proceed with the $z$ update. Let $\Phi^k$ be the set of workers whose $(x_i, \lambda_i)$ updates have arrived at the master at (master's) iteration $k$. Analogous to (4), the master updates $z$ as

$$\begin{aligned} z^{k+1} &= \arg\min_z \sum_{i=1}^N \langle -\hat{\lambda}_i, z\rangle + \frac{\beta}{2}\|\hat{x}_i - z\|^2 \\ &= \frac{1}{N}\sum_{i=1}^N \left(\hat{x}_i + \frac{1}{\beta}\hat{\lambda}_i\right), \end{aligned} \tag{7}$$

where $\hat{x}_i$ (resp. $\hat{\lambda}_i$) is the most recent $x_i$ (resp. $\lambda_i$) received from worker $i$ by the master. Note that though as few as only $S$ fresh updates have arrived, the update in (7) is still based on all the $\{(\hat{x}_i, \hat{\lambda}_i)\}_{i=1}^N$. Hence, it is possible that many of these $(\hat{x}_i, \hat{\lambda}_i)$'s are out-of-date.

Finally, the master's clock $k$ is incremented by 1, and it sends the updated $z^{k+1}$ back to only the workers in $\Phi^k$.

**Algorithm 3** Asynchronous ADMM (async-ADMM): Processing by the master.

1: initialize: $k = 0, \hat{x}_i = 0, \hat{\lambda}_i = 0, i = 1, 2, \ldots, N$.
2: **repeat**
3:   **repeat**
4:     wait;
5:   **until** receive a minimum of $S$ updates from the workers **and** $\max(\tau_1, \tau_2, \ldots, \tau_N) \leq \tau$;
6:   **for** worker $i \in \Phi^k$ **do**
7:     $\tau_i \leftarrow 1$;
8:     $\hat{x}_i \leftarrow$ newly received $x_i$ from worker $i$;
9:     $\hat{\lambda}_i \leftarrow$ newly received $\lambda_i$ from worker $i$;
10:   **end for**
11:   **for** worker $i \notin \Phi^k$ **do**
12:     $\tau_i \leftarrow \tau_i + 1$;
13:   **end for**
14:   update $z^{k+1}$ by (7);
15:   broadcast $z^{k+1}$ to all the workers in $\Phi^k$;
16:   $k \leftarrow k + 1$;
17: **until** termination;
18: **output** $z^k$.

In other words, those workers whose updates are not received in this iteration will not be aware of this $z$ update. A side benefit is that some communication bandwidth can be saved. The whole procedure for the master is shown in Algorithm 3.

**Remark** When $S = N$ or $\tau = 1$, the partial synchronization reduces back to full synchronization. Clearly, the proposed algorithm also reduces to sync-ADMM.

#### 3.3.1. Example

Figure 1 shows an example of how the asynchronous ADMM algorithm works, with $S = 2$ and $\tau = 10$. When the master's clock is at 14, updates from workers 3 and 4 arrive and the master commits an update to $z$. When the clock is at 21, though workers 1 and 5 have both arrived (and so meets the partial barrier condition), the $(x_2, \lambda_2)$ update of worker 2 has resided in the master for 10 iterations. As $\tau = 10$, workers 1 and 5 have to wait until a new update from worker 2 arrives.

### 3.4. Updating $\lambda$ by the Worker

After receiving the updated $\tilde{z}_i$ from the master, worker $i$ resumes its operation and updates its local copy of the dual variable in a manner analogous to (5):

$$\lambda_i^{k_i+1} = \lambda_i^{k_i} + \beta(x_i^{k_i+1} - \tilde{z}_i). \tag{8}$$

Finally, it increments its local clock $k_i$ by 1, and update its local $x_i$ as described in Section 3.2. The whole procedure for the worker is shown in Algorithm 4.

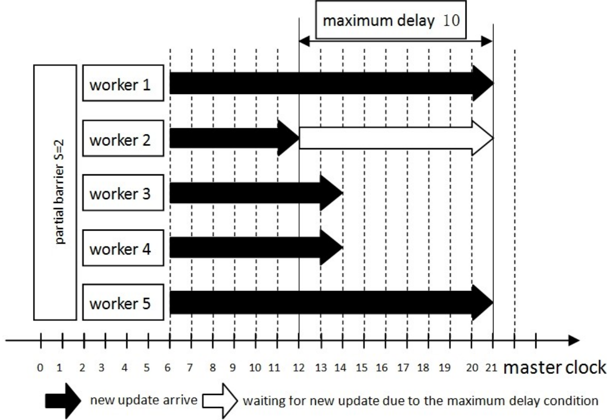



*Figure 1.* An example showing the operation of the partial barrier and bounded delay.

**Algorithm 4** Asynchronous ADMM (async-ADMM): Processing by worker $i$.

1: initialize: $\lambda_i^0 = 0, k_i = 0$.
2: **repeat**
3: update $x_i^{k_i+1}$ using (6);
4: send $\lambda_i^{k_i}$ and $x_i^{k_i+1}$ to the master;
5: **repeat**
6: wait;
7: **until** receive $\tilde{z}_i$ from the master;
8: update $\lambda_i^{k_i+1}$ using (8);
9: $k_i \leftarrow k_i + 1$;
10: **until** termination.

### 3.5. Discussion

In regularized risk minimization, each $f_i(x)$ in (1) can be decomposed as $\tilde{f}_i(x) + g(x)$, where $\tilde{f}_i$ is the risk and $g$ is the regularizer. Thus, (2) can also be written as

$$\min_{x_1,\dots,x_N,z} \sum_{i=1}^N \tilde{f}_i(x_i) + g(z) : x_i = z,\ i = 1,\dots,N. \quad (9)$$

This can still be solved by ADMM (Boyd et al., 2011), and the processing of $g$ is moved from the $x_i$ update (by the worker) to the $z$ update (by the master). However, the master will then run slower and the system's throughput may decrease. Hence, in this paper, the formulation in (2) is preferred.

## 4. Convergence Analysis

In this section, we provide convergence analysis for the async-ADMM algorithm. Typically, the sending and receiving of the worker updates are non-deterministic and depend on a number of factors, such as network bandwidth and traffic, processor configuration and work load, etc. To simplify analysis, we make the following assumption:

**Assumption 4.1** *At any master iteration $k$, updates from the $N$ workers have the same probability of arriving at the master.*

Assume that the master clock $k$ has run for $T$ iterations, and each worker clock $k_i$ for $T_i$ iterations. Let $z_i^{k_i}$ be the $\tilde{z}_i$ received by worker $i$ at its $k_i$th iteration. Moreover, for worker $i$, let $\bar{x}_i = \frac{1}{T_i}\sum_{k_i=0}^{T_i-1} x_i^{k_i}$ be the average of all $x_i$'s generated throughout its $T_i$ iterations. Similarly, let $\bar{z} = \frac{1}{T}\sum_{k=0}^{T-1} z^k$ be the average of all $z$'s generated by the master throughout its $T$ iterations.

**Theorem 4.2** *Let $(x^*, z^*)$ be the optimal (primal) solution of (2), and $\{\lambda_i^*\}_{i=1}^N$ the corresponding optimal dual solution. Then,*

$$\mathbb{E}\left[\sum_{i=1}^N f_i(\bar{x}_i) - f_i(x^*) + \langle \lambda_i^*, \bar{x}_i - \bar{z}\rangle\right] \leq \frac{N\tau}{2TS}\left\{\sum_{i=1}^N \beta\|z_i^0 - z^*\|^2 + \frac{1}{\beta}\|\lambda_i^0 - \lambda_i^*\|^2\right\}, \quad (10)$$

*where $z_i^0$ and $\lambda_i^0$ are the initial values of $z_i$ and $\lambda_i$, respectively, at worker $i$.*

The $\mathcal{O}(\frac{N\tau}{TS})$ convergence rate can be intuitively explained as follows.

- When $N$ is large, the data subset assigned to each worker gets smaller. Thus, each worker update is less informative, and more iterations are needed for convergence.
- A large $S$ means that information from more workers are collected in each master update, and so the number of iterations required for convergence is reduced.
- Recall that every worker will be serviced by the master at least $\frac{T}{\tau}$ times in $T$ master iterations. Hence, a large $\tau$ means that information from the slow workers are incorporated into the master very infrequently. Thus, again a larger $T$ is needed for convergence.

When $S = 1$, one only uses the bounded delay condition but not the partial barrier. This is similar to other distributed optimization algorithms such as (Agarwal & Duchi, 2011; Ho et al., 2013; Li et al., 2013). The following shows that a much tighter bound (by a factor of $N$) can be obtained.

**Corollary 4.3** *When $S = 1$,*

$$\mathbb{E}\left[\sum_{i=1}^N f_i(\bar{x}_i) - f_i(x^*) + \langle \lambda_i^*, \bar{x}_i - \bar{z}\rangle\right] \leq \frac{\tau}{2T}\left\{\sum_{i=1}^N \beta\|z_i^0 - z^*\|^2 + \frac{1}{\beta}\|\lambda_i^0 - \lambda_i^*\|^2\right\}. \quad (11)$$

When the workers and network are fast, updates from every worker can arrive at each iteration. Essentially, we then have $S = N$, and the bound in (10) becomes $\mathcal{O}(\frac{\tau}{T})$. Since all the $\tau_i$'s are always 1 in this case, we can simply set $\tau = 1$. The bound then reduces to $\mathcal{O}(\frac{1}{T})$, which is the same as that of ADMM (He & Yuan, 2012). Similarly, the proposed algorithm also reduces to sync-ADMM when $\tau = 1$. The master then has to wait for all the workers in each iteration. The partial barrier condition is always satisfied for any $1 \leq S \leq N$. In particular, we can set $S = N$, and recover the $\mathcal{O}(\frac{1}{T})$ convergence rate.

## 5. Experiments

In this section, we perform experiments on three different ADMM applications: network average consensus (Section 5.1), graph-guided fused lasso (Section 5.2), and low-rank matrix factorization (Section 5.3). To reduce statistical variability, results are averaged over 5 repetitions.

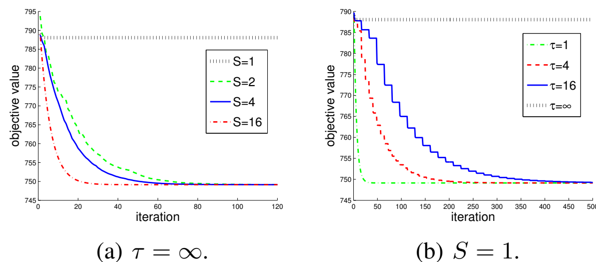
We use a cluster of 18 computing nodes interconnected with a gigabit Ethernet. Each node has 4 AMD Opteron 2216 (2.4GHz) processors and 16GB memory. The master and each worker process take up one core. The algorithms are implemented in C++, with the Armadillo v3.920.3 library[3] linked to LAPACK/BLAS[4] for efficient computation. Moreover, the Message Passing Interface (MPI) implementation MPICH v3.0.4[5] is used for inter-processor communication. Empirically, assumption 4.1 is observed to hold for this cluster setup.

### 5.1. Network Average Consensus

In this experiment, we have $N = 16$ workers, each with a vector $\theta_i \in \mathbb{R}^{100}$. The elements of $\theta_i$ are drawn i.i.d. from the normal distribution with zero mean and unit variance. The task is to find the average of all $\theta_i$'s. This can be formulated as the optimization problem: $\min_x f(x) = \sum_{i=1}^{N} \|x - \theta_i\|^2$. Thus, $f_i(x)$ in (1) equals $\|x - \theta_i\|^2$.

#### 5.1.1. Convergence w.r.t. Number of Iterations

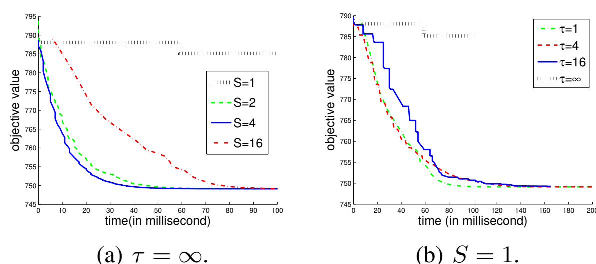
Figure 2 shows the convergence of the objective value at different settings. Figure 2(a) shows the case for $\tau = \infty$. Recall from Section 4 that the convergence rate is $\mathcal{O}(\frac{N\tau}{TS})$. Hence, as can be seen, a smaller $S$ takes more iterations for convergence (the case for $S = 1$ converges to a local solution instead of the global one. See the discussion in the next paragraph). In Figure 2(b), $S$ is fixed at 1. As can be seen, a larger $\tau$ leads to more iterations, which again agrees with the theoretical convergence rate. Moreover, recall that async-ADMM is the same as sync-ADMM when $S = N$




(a) $\tau = \infty$. (b) $S = 1$.

*Figure 2.* Convergence of async-ADMM w.r.t. the number of (master) iterations on the network average consensus problem. Recall that sync-ADMM corresponds to $S = 16$ or $\tau = 1$.

or $\tau = 1$. Hence, sync-ADMM has the fastest convergence in terms of the number of iterations.




(a) $\tau = \infty$. (b) $S = 1$.

*Figure 3.* Convergence of async-ADMM w.r.t. the wall clock time on the network average consensus problem.

Interestingly, the curves in Figure 2(b) exhibit a staircase structure. Note that in this simple consensus problem, the computation costs at both the master and workers are very small.[6] Besides, for workers that reside in the same computing node as the master, their communication costs with the master are also negligible. Hence, these workers can quickly reach a local consensus among themselves, without waiting for updates from the more distant workers. As this local consensus is only based on the $\theta_i$'s of the participating workers, it can be very different from the true average. This accounts for the flat regions of the curve. The situation remains until the bounded delay condition kicks in, and updates from some distant workers arrive, leading to a new consensus (the "cliffs" of the curves), and the process repeats. Moreover, the larger the $\tau$, the longer it takes for the bounded delay condition to kick in, and the longer is the flat region. When $\tau = \infty$, little progress is observed.

#### 5.1.2. Convergence w.r.t. Time

On the other hand, the convergence behavior when measured w.r.t. the wall clock time shows a different picture.

[3] http://arma.sourceforge.net/

[4] http://www.netlib.org/

[5] http://www.mpich.org/

[6] It is easy to see that both master and worker updates reduce to the solving of quadratic equations, which have simple closed-form solutions.

Recall that each sync-ADMM iteration requires full synchronization, and thus takes longer than an async-ADMM iteration. Figure 3(a) shows the results for $\tau = \infty$. As can be seen, async-ADMM with $S > 1$ converges much faster than sync-ADMM. Figure 3(b) shows the case for $S = 1$. As can be seen, async-ADMM still has slower convergence than sync-ADMM. The reason, as discussed in Section 5.1.1, is that async-ADMM wastes a lot of iterations (and thus time) on reaching inaccurate local consensus. Hence, setting $S = 1$ may not be suitable in applications where the computation cost is much smaller than the worker-to-master communication cost.

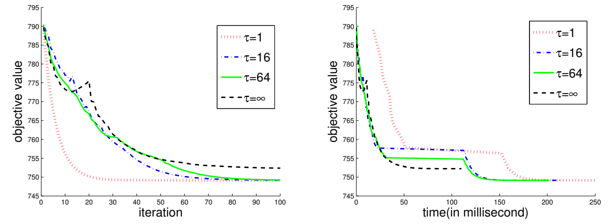
#### 5.1.3. Temporary Worker Failure

In this section, we simulate the situation where a worker fails temporarily. Specifically, one of the workers (say, $A$) is temporally suspended for 100 milliseconds at its 10th iteration. While the sync-ADMM has to come to an immediate halt, async-ADMM allows the system to proceed for $\tau$ more master iterations (and hopefully the faulty worker will be able to recover by then). The convergence behavior is shown in Figure 4. As can be seen from Figure 4(b), for async-ADMM with $\tau = 1, 16, 64$, their progress is delayed (as expected) but their objective values drop again when $A$ is resumed operation. However, for $\tau = \infty$, the algorithm only converges to a local solution when the master finishes its $T$ iterations.

As discussed in Section 3.3, the bounded delay condition guarantees that every worker will be serviced by the master at least $\frac{T}{\tau}$ times in $T$ master iterations. With a large $\tau$, a high degree of asynchrony can be ensured though at the expense that information in some of the workers may not be visited that often. With a sufficiently large $T$, the obtained solution is still guaranteed to be optimal (Section 4). However, when the master is only allowed to run for a fixed number of iterations (as is often the case in practice), the solution quality may be compromised if some workers are very slow or have intermittent failure. One approach to alleviate this problem is by employing data redundancy schemes (Dean & Ghemawat, 2008), which, however, is outside the scope of this paper.

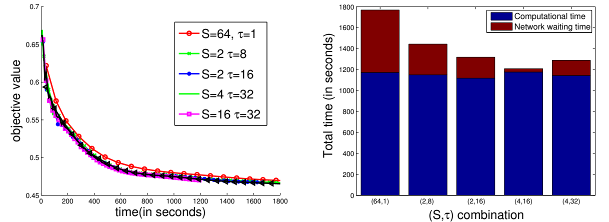
### 5.2. Graph-Guided Fused Lasso

In this section, we perform classification experiments with a variant of the generalized lasso model (Tibshirani & Taylor, 2011): $\min_x \frac{1}{L}\sum_{i=1}^{L} \ell_i(x) + \lambda\|Ax\|_1$, where $L$ is the number of samples, $\ell_i$ is the logistic loss (which is more appropriate than the square loss in classification), $\lambda$ is the regularization parameter and $A$ is is a penalty matrix specifying the desired structured sparsity pattern of $x$. With different settings of $A$, this can be reduced to models such as the fused lasso, trend filtering, and wavelet smooth-




(a) Convergence w.r.t. the number of (master) iterations. (b) Convergence w.r.t. the wall clock time.

*Figure 4.* Simulation with one worker suffers temporary failure (with $S = 2$).




(a) Convergence of the objective with time. (b) Breakdown into computation time and network waiting time.

*Figure 5.* Comparison of sync-ADMM and async-ADMM on the graph-guided fused lasso problem. Recall that the combination of $S = 64, \tau = 1$ corresponds the sync-ADMM.

ing. Here, we will focus on the graph-guided fused lasso (Kim et al., 2009), whose sparsity pattern is specified by a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ defined on the $d$ variates of $x$. By defining $A_{ij} = w_{ij}$ and $A_{ji} = -w_{ij}$ for any edge $(i, j) \in \mathcal{E}$, we have $\|Ax\|_1 = \sum_{(i,j)\in\mathcal{E}} w_{ij}|x_i - x_j|$ which penalizes the difference between any two neighboring $x_i, x_j$ in $\mathcal{G}$. Following (Ouyang et al., 2013), $\mathcal{G}$ is obtained by sparse inverse covariance selection (Banerjee et al., 2008).

We use the digits 4 and 9 from the *MNIST-8M*[7] data set, resulting in a total of $L = 1.6$ million 784-dimensional samples. These are partitioned uniformly and each of the $N$ workers is assigned $\frac{L}{N}$ samples. The local objective associated with worker $i$ is thus

$$f_i(x) = \frac{1}{L}\sum_{j\in\Omega_i} \ell_j(x) + \frac{\lambda}{N}\|Ax\|_1,$$

where $\Omega_i$ is the sample subset assigned to $i$. The subproblem in each worker is solved by the inexact ADMM algorithm (Zhang et al., 2011). As shown in (Ouyang et al., 2013; Suzuki, 2013), this is more efficient in this context than other state-of-the-art solvers.

Figure 5(a) compares the convergence speeds of sync-

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

ADMM and async-ADMM on a set of 64 workers. As can be seen, all four async-ADMM settings are faster than sync-ADMM in terms of wall clock time. Figure 5(b) shows the breakdown of total running time into computation time and network waiting time. As can be seen, while the different $(S, \tau)$ combinations have similar computation time, a smaller $S$ and/or larger $\tau$ allows for a higher degree of asynchrony, and thus less time on network waiting.

Next, we vary the number of workers (with $S = 2$ and $\tau = 32$). Figure 6 shows that async-ADMM is again faster than sync-ADMM. Note that with more workers in the cluster, the master needs to spend less time on waiting for at least $S$ worker updates to arrive. Hence, the network waiting time is significantly less than that of sync-ADMM.

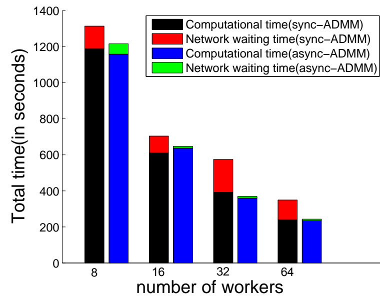



*Figure 6.* Computation/network waiting time for sync-ADMM and async-ADMM, with different numbers of workers on the graph-guide fused lasso problem.

### 5.3. Low-Rank Matrix Factorization

Though the focus of this paper is on convex problems, it is known that ADMM can also be efficiently used on nonconvex problems in practice (Boyd et al., 2011). In this section, we demonstrate the effectiveness of the proposed async-ADMM on one such nonconvex problem, namely, low-rank matrix factorization (Berry et al., 2007).

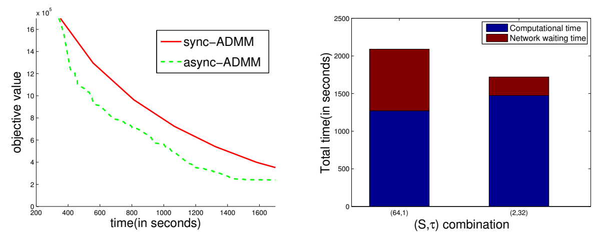
Given a matrix $M \in \mathbb{R}^{m \times n}$, the task is to decompose it into $LR^T$, where $L \in \mathbb{R}^{m \times r}$ and $R \in \mathbb{R}^{n \times r}$ have ranks $r \ll \min(m, n)$. Low-rank matrix factorization can be formulated as the following optimization problem: $\min_{L,R} \|M - LR^T\|_F^2 + \lambda_1 \|L\|_F^2 + \lambda_2 \|R\|_F^2$, where $\|\cdot\|_F$ is the Frobenius norm, and $\lambda_1, \lambda_2$ are regularization parameters. More generally, $M$ may also have missing entries, which can still be solved by ADMM (Ling et al., 2012).

In this experiment, we set $m = 10000, n = 64000$ and $r = 100$. We first generate the ground-truth $L^*$ and $R^*$, by drawing entries independently from the normal distribution with zero mean and unit variance, and then $M$ is obtained as $L^* R^{*T}$. For simplicity, there is no missing entry in $M$, and we set $\lambda_1 = \lambda_2 = 1$. The matrix $M$ is partitioned evenly across columns and then assigned to the $N = 64$ workers. The local objective associated with worker $i$ is

$$f_i(L) = \|M_i - LR_i^T\|_F^2 + \frac{\lambda_1}{N}\|L\|_F^2 + \lambda_2 \|R_i\|_F^2,$$

where $M_i$ and $R_i$ are column subsets of $M$ and $R$, respectively, assigned to worker $i$, and $L$ is the consensus variable. The update of each worker is based on the ADMM solver proposed in (Ling et al., 2012).

Figure 7(a) shows the convergence of the objective with wall clock time. As can be seen, async-ADMM (with $S = 2$ and $\tau = 32$) again converges faster than sync-ADMM. Figure 7(b) shows the breakdown of total running time into computation time and network waiting time. As can be seen, the speedup by async-ADMM mainly comes from the significant reduction in network waiting.




(a) Convergence of the objective with wall clock time.

(b) Breakdown into communication time and computation time.

*Figure 7.* Comparison of sync-ADMM (with $S = 64, \tau = 1$) and async-ADMM on the low-rank matrix factorization problem.

## 6. Conclusion

Existing asynchronous distributed optimization algorithms are mainly limited to the gradient descent and its variants. In this paper, we extended asynchronous distributed processing to the ADMM algorithm for the global variable consensus problem. It uses two conditions, partial barrier and bounded delay, to control the asynchrony. Besides, the traditional synchronous ADMM algorithm can be regarded as a special case. The proposed algorithm is easy to implement, has theoretical convergence guarantees, and is also faster than its synchronous counterpart in practice. As many machine learning problems can be formulated as a global variable consensus problem, it opens new opportunities for these models to be learned more efficiently in distributed computing environments. In the future, we will also compare with asynchronous distributed gradient-based algorithms such as (Ho et al., 2013; Li et al., 2013).

## Acknowledgments

This research was supported in part by the Research Grants Council of the Hong Kong Special Administrative Region (Grant 614513).

## References


Agarwal, A. and Duchi, J.C. Distributed delayed stochastic optimization. In *Advances in Neural Information Processing Systems 24*, 2011.

Albrecht, J.R., Tuttle, C., Snoeren, A.C., and Vahdat, A. Loose synchronization for large-scale networked systems. In *Proceedings of the USENIX Annual Technical Conference*, pp. 301–314, 2006.

Banerjee, O., El Ghaoui, L., and d'Aspremont, A. Model selection through sparse maximum likelihood estimation for multivariate Gaussian or binary data. *Journal of Machine Learning Research*, 9:485–516, 2008.

Berry, M.W., Browne, M., Langville, A.N., Pauca, V.P., and Plemmons, R.J. Algorithms and applications for approximate nonnegative matrix factorization. *Computational Statistics & Data Analysis*, 52(1):155–173, 2007.

Bertsekas, D.P. and Tsitsiklis, J.N. *Parallel and Distributed Computation*. Prentice Hall, 1989.

Boyd, S., Parikh, N., Chu, E., Peleato, B., and Eckstein, J. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2011.

Dean, J. and Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

He, B. and Yuan, X. On the $O(1/n)$ convergence rate of the Douglas-Rachford alternating direction method. *SIAM Journal on Numerical Analysis*, 50(2):700–709, 2012.

Ho, Q., Cipar, J., Cui, H., Lee, S., Kim, J.K., Gibbons, P.B., Gibson, G.A., Ganger, G., and Xing, E. More effective distributed ML via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems 26*, pp. 1223–1231, 2013.

Kim, S., Sohn, K.-A., and Xing, E.P. A multivariate regression approach to association analysis of a quantitative trait network. *Bioinformatics*, 25(12):i204–i212, 2009.

Langford, J., Smola, A., and Zinkevich, M. Slow learners are fast. In *Advances in Neural Information Processing Systems 22*, 2009.

Li, M., Andersen, D.G., and Smola, A. Distributed delayed proximal gradient methods. In *NIPS Workshop on Optimization for Machine Learning*, 2013.

Ling, Q., Xu, Y., Yin, W., and Wen, Z. Decentralized low-rank matrix completion. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, pp. 2925–2928, 2012.

Lubell-Doughtie, P. and Sondag, J. Practical distributed classification using the alternating direction method of multipliers algorithm. In *Proceedings of the International Conference on Big Data*, 2013.

Mota, J, Xavier, J, Aguiar, P, and Puschel, Markus. D-admm: A communication-efficient distributed algorithm for separable optimization. *IEEE Transactions on Signal Processing*, 61(10):2718–2723, 2013.

Niu, F., Recht, B., Ré, C., and Wright, S.J. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24*, 2011.

Ouyang, H., He, N., Tran, L., and Gray, A.G. Stochastic alternating direction method of multipliers. In *Proceedings of the 30th International Conference on Machine Learning*, pp. 80–88, 2013.

Shalev-Shwartz, S., Singer, Y., and Srebro, N. Pegasos: Primal estimated sub-gradient solver for SVM. In *Proceedings of the 24th International Conference on Machine Learning*, pp. 807–814, 2007.

Suri, S. and Vassilvitskii, S. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, pp. 607–614, 2011.

Suzuki, T. Dual averaging and proximal gradient descent for online alternating direction multiplier method. In *Proceedings of the 30th International Conference on Machine Learning*, pp. 392–400, 2013.

Tibshirani, R.J. and Taylor, J. The solution path of the generalized lasso. *Annals of Statistics*, 39(3):1335–1371, 2011.

Wei, E. and Ozdaglar, A. Distributed alternating direction method of multipliers. In *Proceedings of the 51st Annual Conference on Decision and Control*, pp. 5445–5450, 2012.

Wei, E. and Ozdaglar, A. On the $O(1/k)$ convergence of asynchronous distributed alternating direction method of multipliers. Preprint arXiv:1307.8254, 2013.

Zhang, X., Burger, M., and Osher, S. A unified primal-dual algorithm framework based on Bregman iteration. *Journal of Scientific Computing*, 46(1):20–46, 2011.

Zhu, H., Cano, A., and Giannakis, G.B. Distributed consensus-based demodulation: Algorithms and error analysis. *IEEE Transactions on Wireless Communications*, 9(6):2044–2054, 2010.