# Distributed Optimization: synchronous and asynchronous ADMM

Zhenyuan Liu, 26968476

## 1 Introduction

Despite the superior performance of many state-of-the-art algorithms, solving huge-scale convex optimization problems in serial is infeasible. Distributed optimization has thus become indispensable in solving such problems. In other problems, many agents seek to minimize the sum of the objective functions collectively, yet they only have access to local objective functions and decision variables. Distributed optimization is a necessary tool to solve such problems. This project explores one of the distributed optimization algorithm: Alternating Direction Method of Multipliers (ADMM)[1] in solving huge-scale LASSO problems. In addition to the standard (synchronous) ADMM algorithm, asynchronous (communication-avoiding) ADMM algorithms [2][3] are investigated as well.

## 2 Distributed ADMM algorithm

A thorough analysis of the ADMM algorithm (dual ascent, dual decomposition, augmented Lagrangian and the method of multipliers, convergence proof etc.) can be found in[1]. In this section, we briefly introduce the distributed ADMM algorithm in solving a LASSO problem.

### 2.1 Synchronous ADMM algorithm

In ADMM form, the LASSO problem $\frac{1}{2}||Ax-b||_2^2+\lambda||x||_1$ can be written as:

$$\min_{x,z} \quad f(x)+g(z) \quad s.t. \quad x-z=0 \tag{1}$$

where $f(x)=\frac{1}{2}||Ax-b||_2^2$, $A\in R^{m\times n}$ is the data matrix, $b\in R^{m\times 1}$ is the observed value vector, $x\in R^{n\times 1}$ is the decision variable, $g(z)=\lambda||z||_1$, $z\in R^{n\times 1}$ is the consensus decision variable, and $\lambda$ is the regularization coefficient. $A$ and $b$ can be partitioned by samples (rows), leading to the decomposed version of the above problem:

$$\min_{x_i,z} \quad \sum_{i=0}^{N-1} f_i(x_i)+g(z) \quad s.t. \quad x_i-z=0, \quad \forall i=0,...,N-1 \tag{2}$$

where $f_i(x_i)=\frac{1}{2}||A_ix_i-b_i||_2^2$, $A_i\in R^{m_i\times n}$ is the local data matrix with $m_i=\frac{m}{N}$, $b_i\in R^{m_i\times 1}$ is the local observed value vector, $x_i\in R^{n\times 1}$ is the local decision variable, $g(z)=\lambda||z||_1$, $z\in R^{n\times 1}$ is the global consensus decision variable. When the above optimization problem is solved, all local $x_i$ and

$z$ are equal to each other. Following from [1], the distributed ADMM algorithm for the LASSO problem consists of the following updates:

$$\begin{aligned} x_i^{k+1} &:= \arg\min_{x_i} \ \frac{1}{2}||A_i x_i - b_i||_2^2 + \ \frac{\rho}{2}||x_i - z^k + u_i^k||_2^2 \\ z^{k+1} &:= S_{\lambda/\rho N}(\overline{x}^{k+1} + \overline{u}^k) \\ u_i^{k+1} &:= u_i^k + x_i^{k+1} - z^{k+1} \end{aligned} \tag{3}$$

where $S$ is the soft-thresholding operator, and $\rho$ is a coefficient regarding the augmented Lagrangian, which is set to 1. The the dual ascent step ($u$-update, step 3) and the $x$-minimization step (step 1) are performed locally in each rank. After that, all $x_i$ and $u_i$ are collected by the master rank to calculate $\overline{x}$ and $\overline{u}$, followed with the $z$-update step (step 2). The $x$-minimization step has a closed-form solution:

$$x_i^{k+1} = (A_i^T A_i + \rho I)^{-1}(A_i^T b_i + \rho(z^k - u_i^k)) \tag{4}$$

which can be calculated easily using matrix algebra tools such as Numpy. For the $x$-minimization step, we can calculate the LU decomposition of $A_i^T A_i + \rho I$ and cache the L/U matrices to expedite the computation of $x_i^{k+1}$, because $A_i^T A_i + \rho I$ remains the same for a given rank. Note that we can use the matrix inversion lemma to factor $\frac{1}{\rho} A_i A_i^T + I$ instead, if the local $A_i$ matrix is fat (i.e. $m_i < n$)[1].

### 2.2 Asynchronous ADMM algorithm

In the synchronous ADMM algorithm, the master rank waits until it receives the updated $x_i$ and $u_i$ from all ranks at each iteration. As a result, the performance is limited by the slowest worker. Zhang[2] proposed an asynchronous ADMM algorithm that only requires **partial** synchronization, in which the master rank performs the $z$-update step as soon as it receives the updated $x_i$ and $u_i$ from a preset number of ranks. Under assumption 1, their algorithm has a convergence rate of $\mathcal{O}(\frac{N\tau}{TS})$, where $N$ is the number of ranks, $S$ is the minimum number of ranks required for partial synchronization, $\tau$ is the maximum allowed delay of slow workers (in number of steps), and $T$ is the number of master iterations. This convergence rate agrees with the general $\mathcal{O}(1/T)$ convergence rate from [3].

**Assumption 1:** At any master iteration $k$, updates from the $N$ workers have the same probability of arriving at the master.

## 3 Implementation

A python implementation of the synchronous and asynchronous ADMM algorithms using MPI4PY[4] was used to solve huge scale LASSO problems using Intel Xeon "Haswell" processor nodes from Cori[5]. The code used in this project referenced the MATLAB and C code of Boyd[1].
In the code of Boyd[1], *AllReduce* is used to collect $x_i$ and $u_i$ to compute the new $z$. The code for the current project adopts a more general implementation, which is applicable to both the synchronous and asynchronous ADMM algorithms. In the code of this project, all ranks (except the master rank itself) sends the updated $x_i$ and $u_i$ to the master rank using non-blocking *isend*. In the master rank, *Iprobe* is used to check if the message sent by a given rank is received. Once

the number of messages received by the master rank is larger than or equal to $S$, the master rank moves to the $z$-update step. It then sends the updated $z$ to all ranks which have contributed to the latest $z$-update step, finishing one iteration.
When $S = N$, the master rank waits until the updated $x_i$ and $u_i$ from all ranks are received, which is equivalent to the synchronous ADMM algorithm. When $S < N$, the ADMM algorithm is asynchronous as it only requires partial synchronization. In this case, some slow ranks may not have contributed to the most recent update step of $z$ and thus don't receive the most up-to-date $z$. As a result, these ranks are outdated. Any rank is allowed to be at most $\tau$ steps behind the master rank. If a given rank is outdated for $\tau$ steps, the master rank will not move to the $z$-update step until it receives the updated $x_i$ and $u_i$ from this rank. This ensures that slow ranks can also contribute to the update of the global consensus decision variable $z$.
The data used in all computer experiments are generated randomly: $x^{true}$ is generated randomly and is the same for all ranks, which is also sparse; $A_i$ are generated randomly in each rank, with the seed set to the rank number; $b_i = A_i x^{true} + v_i$, where $v_i$ is a random noise vector; $\lambda$ is set to 0.5. Initially, $x_i$, $u_i$ and $z$ are all set to zero vectors.

# 4 Results

For all the computer experiments in this section, we take the average of the last two runs of a total of three runs. In some cases, the first run takes a significant longer time, possibly due to the initialization of the hardware and MPI etc. So we don't use the results from the first run. Ideally, the results would be more accurate if we do more runs and take the average. But we use a total of three runs due to the limited computing hours.
First we solve an enormous LASSO problem for the purpose of demonstration. In this enormous problem, $A$ has $m = 2,457,600$ samples and $n = 20,000$ features, with a size about 400 GB! The synchronous ADMM algorithms converges in 92 steps (746 seconds). This shows the excellent performance of the ADMM algorithm. Moreover, such a big problem cannot be solved serially at all. Due to the limited computing hours, we solve a smaller but still huge LASSO problem for all other computer experiments: $A$ has $m = 614,400$ samples and $n = 20,000$ features, with a size about 100 GB.

## 4.1 Synchronous ADMM algorithm: Strong scaling experiments

Generally, people are more interested in solving huge-scale problems using the ADMM algorithm. Therefore, we are more interested in the strong scaling performance of the ADMM algorithm.
In this part, we solve a LASSO problem: $A$ has $m = 614,400$ samples and $n = 20,000$ features, with a size about 100 GB. This problem is solved using 64 to 512 cores, i.e. $m_i$ ranges from 9,600 to 1,200. The number of features $n$ is set to 20,000 so that the limiting factor for the $x$-minimization step is $m_i$, since the $x$-minimization step is in polynomial time of $\min(m_i, n)$.
When more cores are used, the local $A_i$ matrix is smaller and thus less informative, requiring more iterations to converge, as shown in the left panel of Figure 1. On the other hand, each iteration is much cheaper in terms of computation, when $A_i$ is smaller. Besides, the communication cost increases as the number of cores increases. These three factors combined lead to the results in the right panel of Figure 1. The time elapsed till convergence decreases from $N = 64$ to $N = 128$ and

$N = 256$, it then increases from $N = 256$ to $N = 512$.

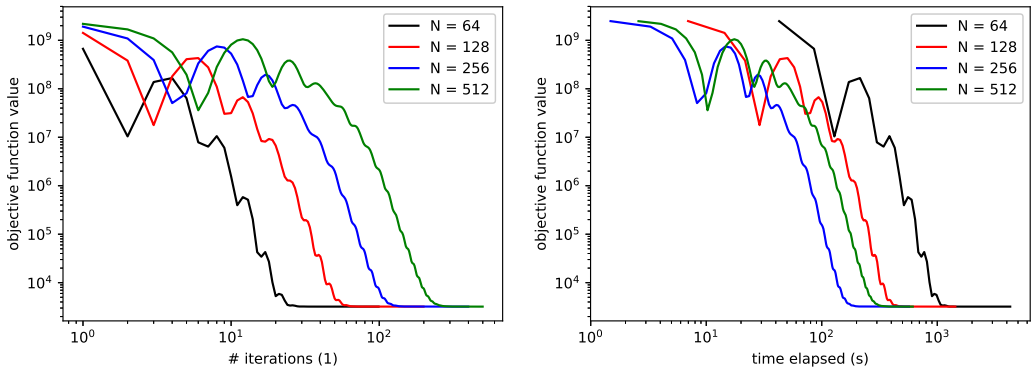


Figure 1: Objective function value vs. # iterations and time elapsed

The breakdown of computation and communication time of the master rank is plotted in Figure 2. The computation time decreases steadily as the number of cores increases (as $m_i$ decreases). Meanwhile, communication time increases slightly until $N = 256$ and then increases abruptly from 256 to 512. In addition to the more complex communication structure, the rank-to-rank variability in the computation time of the $x$-minimization step also increases as the number of cores increases, although the work load is approximately the same for all ranks. These two reasons explain the sudden jump of communication time from $N = 256$ to $N = 512$. In the synchronous ADMM algorithm, the master rank must wait until it receives the updated $x_i$ and $u_i$ from the slowest rank at each iteration. As more cores are used, the delay caused by the slowest rank becomes larger. In this problem, communication inefficiency prevents us from using more than 256 cores.

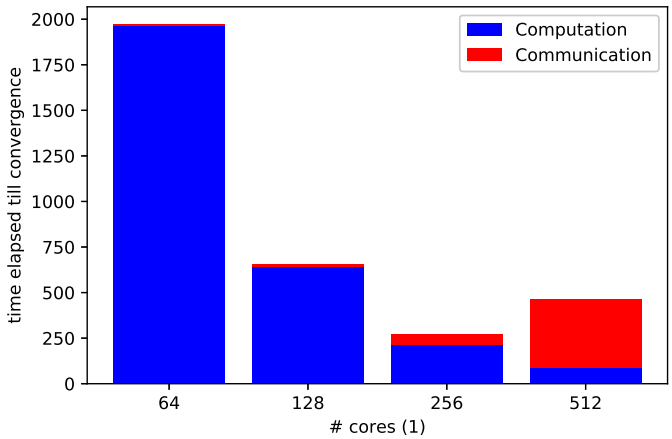


Figure 2: Computation and communication time till convergence vs # cores

As shown in Figure 3, it takes 46 steps (2,012 seconds ) to converge using 64 cores and 376 steps (463 seconds) to converge using 512 cores. The optimal number of cores is 256: it takes 191 steps (274 seconds) to converge. Note that the strong scaling efficiency is higher than 1 (from $N = 64$ to $N = 256$, the strong scaling efficiency is 1.84) because the $x$-minimization step is in polynomial time of $\min(m_i, n)$. These results show that the total computation time can be reduced significantly using distributed ADMM with a carefully chosen number of cores.

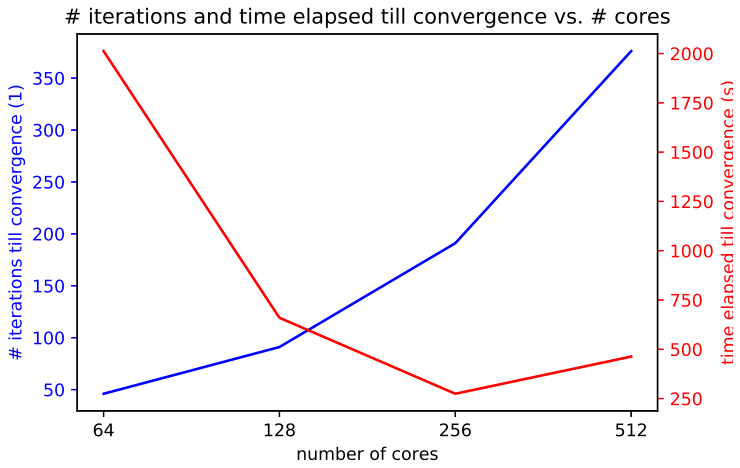


Figure 3: # iterations and time elapsed till convergence vs # cores

## 4.2 Asynchronous ADMM algorithm

### 4.2.1 Convergence issues of asynchronous ADMM

In this part, we use the asynchronous ADMM algorithm as described in section 3 to solve the same LASSO problem in section 4.1: $A$ has $m = 614,400$ samples and $n = 20,000$ features. With a total of 512 cores, each core deals with a local $A_i$ matrix with $m_i = 1,200$ samples and $n = 20,000$ features. The maximum allowed delay of slow workers $\tau$ (in number of steps) is set to 5. The minimum number of cores required for partial synchronization $S$ is set to 512, 448, 384 and 320, respectively. Note that the $S = 512$ case is equivalent to the synchronous ADMM algorithm. As shown in Figure 4, the asynchronous ADMM algorithm ($S = 448$, 384, and 320) doesn't converge after 500 iterations. In contrast, the synchronous ADMM algorithm ($S = 512$) converges.

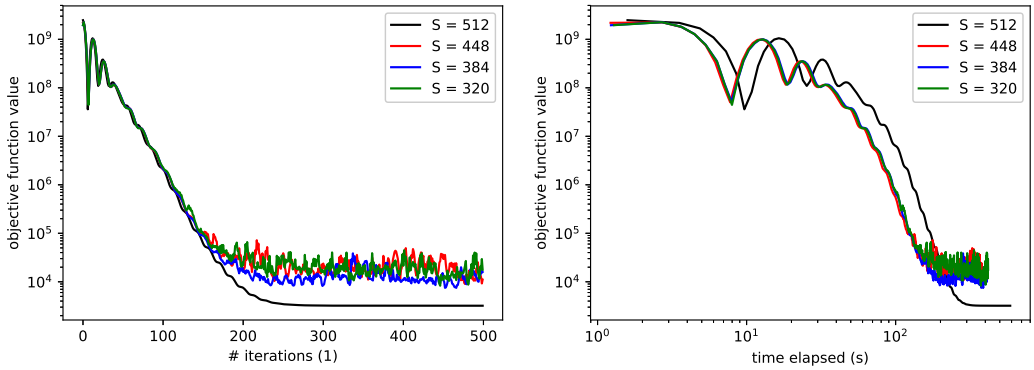


Figure 4: Objective function value vs. # iterations and time elapsed for different S

This is caused by the fact that Assumption 1 in section 2.2 doesn't hold. Empirically, updates from the $N$ workers have a **different** probability of arriving at the master core at any master iteration $k$. In the asynchronous ADMM algorithm, the master rank waits until it receives the updated $x_i$ and $u_i$ from at least $S$ ranks. It's possible that a given rank doesn't contribute to the master $z$-update step at a given iteration. In this case, this rank doesn't receive the most up-to-date $z$ and uses a outdated $z$ for its local computation. In general, the number of updates for a given rank may be less than the number of master iterations. The number of updates for different ranks are plotted using histograms in Figure 5. In the case of $S = 512$ (the synchronous case), all ranks have exactly 500 updates, contributing equally to the convergence of the algorithm. However, their contributions differ in cases when $S < 512$, i.e. in the asynchronous cases. For example, when

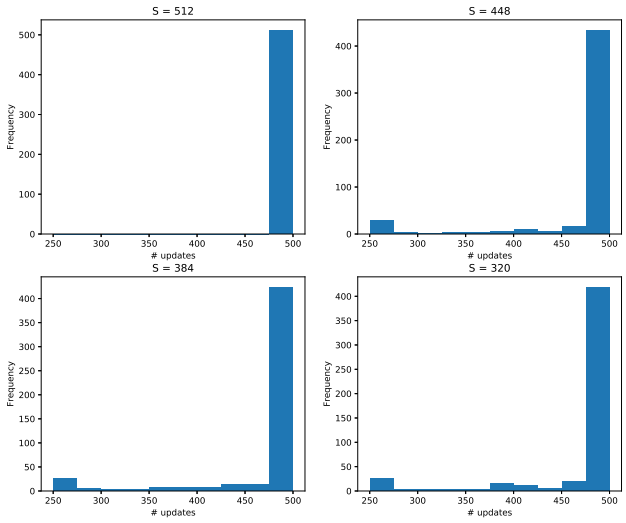


Figure 5: Objective function value vs. # iterations and time elapsed for different S

$S = 448$, most ranks update 500 times, whereas others update only 251 times for a total of 500 iterations. This evidence shows that Assumption 1 in section 2.2 doesn't hold: the convergence of the asynchronous ADMM algorithm in section 2.2 doesn't hold as well because the information in certain ranks haven't been sufficiently incorporated into the $z$-update in the master rank, causing a fluctuation of the objective function value.

### 4.2.2 Hybrid ADMM algorithm

To deal with the convergence problem of the asynchronous ADMM algorithm, an hybrid ADMM algorithm is proposed in this project, in which $S$ increases to $N$ gradually. Specifically, $S$ increments by a preset number $\Delta S$ in each master iteration after the number of master iterations reaches a preset cutoff value $k_{cut}$. The hybrid algorithms blends the convergence guarantee of the synchronous ADMM algorithm and the fast speed of the asynchronous ADMM algorithms.
Once again, we solve the same problem in section 4.1: $A$ has $m = 614,400$ samples and $n = 20,000$ features. We first use use $N = 512$ cores, each core dealing with a local $A_i$ matrix with $m_i = 1,200$ samples and $n = 20,000$ features. The two hyper-parameters for the hybrid ADMM algorithm is selected as $\Delta S = 3$, and $k_{cut} = 100$. These hyper-parameters are selected roughly without much effort in tuning.
As shown in Figure 6, the hybrid ADMM algorithm converges because it becomes fully synchronous after a certain number of iterations. It also converges faster than the synchronous ADMM algorithm ($S = 512$) because it's much faster in the asynchronous phase of the hybrid ADMM algorithm. When $S = 512$, it takes 463 seconds to converge, longer than the 357 seconds required when $S = 320$ initially.
In this particular problem with $N = 512$ cores, the hybrid ADMM algorithm outperforms the synchronous ADMM algorithm. However, it should be noted that this is due to the fact that communication has become a bottleneck when $N = 512$ cores are used. In section 4.1, we have shown that the communication time makes up the majority of the total run time when $N = 512$. Specifically, the communication time is 378 seconds, whereas the total run time is 463 seconds till convergence when $S = 512$ (the synchronous case), as shown in Table 1. If we solve the same problem using 256 or 128 cores, then communication would not be a bottleneck: there is no edge in using the hybrid ADMM algorithm.

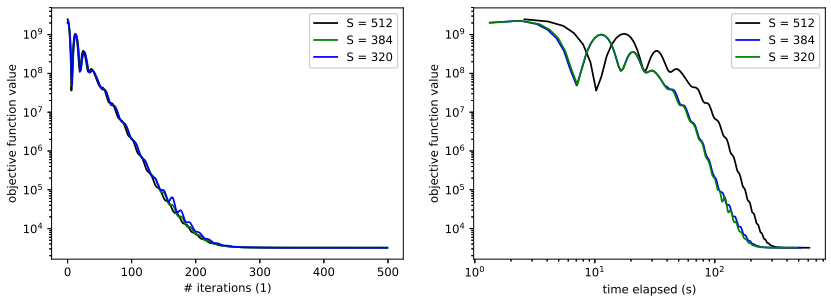


Figure 6: Objective function value vs. # iterations and time elapsed for different S, N = 512

We next use $N = 256$ cores, each core dealing with a local $A_i$ matrix with $m_i = 2,400$ samples and $n = 20,000$ features. In this case, $\Delta S = 3$, and $k_{cut} = 80$. As shown in Figure 7, the hybrid ADMM algorithm performs no better than the synchronous ADMM algorithm. When $S = 256$ (the synchronous case), it takes 274 seconds to converge, meanwhile it takes 277 seconds when $S = 160$ initially. In this case, the communication time is 58 seconds, whereas the total run time is 274 seconds when $S = 256$ (the synchronous case), as shown in Table 1. The communication takes a smaller fraction of the total run time. Therefore, the advantage of the hybrid ADMM algorithm is negligible in this problem.

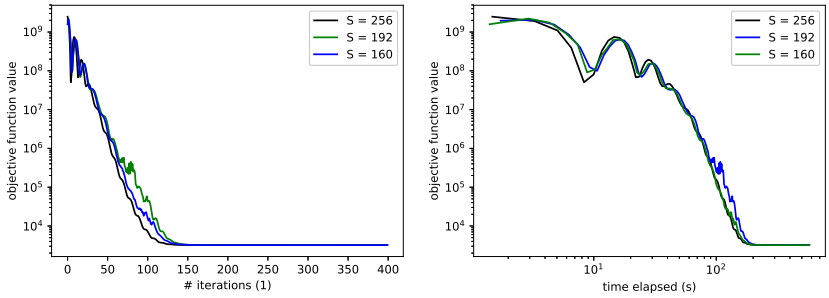


Figure 7: Objective function value vs. # iterations and time elapsed for different S, N = 256

We finally use use $N = 128$ cores, each core dealing with a local $A_i$ matrix with $m_i = 4,800$ samples and $n = 20,000$ features. In this case, $\Delta S = 3$, and $k_{cut} = 40$. As shown in Figure 8, the hybrid ADMM algorithm performs worse than the synchronous ADMM algorithm. When $S = 128$ (the synchronous case), it takes 660 seconds to converge, meanwhile it takes 711 seconds when $S = 96$ initially. In this case, the communication time is only 13 seconds, almost negligible compared to the the total run time of 660 seconds till convergence when $S = 128$ (the synchronous case), as shown in Table 1. We should also avoid using $N = 128$ cores, because the $x$-minimization steps now takes much longer time than $N = 256$ and $N = 512$.

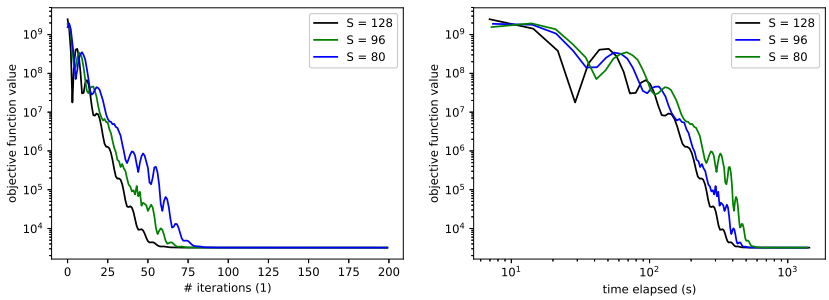


Figure 8: Objective function value vs. # iterations and time elapsed for different S, N = 128

| # cores | $t_{comm}(seconds)$ | $t_{total}(seconds)$ |
|---|---|---|
| 512 | 378 | 463 |
| 256 | 58 | 274 |
| 128 | 13 | 660 |

Table 1: The communication time and the total run time till convergence for the synchronous ADMM algorithm with different N

The results in this section show that the hybrid ADMM algorithms is helpful only when communication is the bottleneck. Among all the different cases in section 4.2, the best strategy is using the synchronous ADMM algorithm with $N = 256$ cores. This conclusion doesn't imply that the hybrid ADMM algorithm is useless. In cloud computing when the latency is a problem, the hybrid ADMM algorithm would converge faster than the synchronous ADMM algorithm, as in the $N = 512$ case.

# 5 Conclusions

In this project, we investigate the synchronous and asynchronous ADMM algorithms in solving huge scale LASSO problems.

The strong scaling results for the synchronous ADMM algorithm reveal that increasing the number of cores doesn't always reduce the total run time in solving a LASSO problem of a fixed size. There is an optimal number of cores with which the total run time is minimal. Multiple factors including the computation time of each iteration, the complexity of the communication structure, as well as the rank-to-rank variability in computation time affect the choice of the number of cores .

The asynchronous ADMM algorithm proposed by[2] doesn't converge in the computer experiments performed due to the invalidity of Assumption 1: at any master iteration $k$, updates from the $N$ workers have the same probability of arriving at the master. A hybrid ADMM algorithm that blends the convergence guarantee of the synchronous ADMM algorithm and the fast speed of the asynchronous ADMM algorithms is proposed. The computer experiments results show that this hybrid ADMM algorithm converges in all cases.

Different number of cores are used to solve the same huge LASSO problem using the synchronous and hybrid ADMM algorithms. As the number of cores used changes, the fraction of communication time changes. The hybrid ADMM algorithm is faster only when communication is the bottleneck, i.e. when the fraction of communication time is large. In computing facilities such as cloud where the latency is an issue, the hybrid ADMM algorithms could be advantageous.

When $N = 512$, the communication time is 378 seconds, whereas the total run time is only 463 seconds till convergence for the synchronous ADMM algorithm. The reason for the sudden jump of the communication time from $N = 256$ to $N = 512$ is worth exploring. It may be purely due to the fact that the communication structure becomes more complex as there are more cores. It could also be caused by the increased rank-to-rank variability in local computation time (recall synchronous ADMM is limited by the slowest rank), although the computation effort in each rank is almost the same for the ADMM algorithm in solving LASSO problems. There may be a solution to solve this problem faster using more than $N = 256$ cores.

## References


[1] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

[2] Ruiliang Zhang and James T. Kwok. Asynchronous distributed admm for consensus optimization. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages II–1701–II–1709. JMLR.org, 2014.

[3] E. Wei and A. Ozdaglar. On the o(1/k) convergence of asynchronous distributed alternating direction method of multipliers. In *2013 IEEE Global Conference on Signal and Information Processing*, pages 551–554, Dec 2013.

[4] http://mpi4py.scipy.org/docs/.

[5] http://www.nersc.gov/users/computational-systems/cori/.