Distributed Solution of Large-Scale Linear Systems via Accelerated Projection-Based Consensus

Navid Azizan-Ruhi [1], **Farshad Lahouti** [2], **Salman Avestimehr** [3], **Babak Hassibi** [2]

[1] Department of Computing and Mathematical Sciences, Caltech
[2] Department of Electrical Engineering, Caltech
[3] Department of Electrical Engineering, USC
{azizan,lahouti,hassibi}@caltech.edu, avestimehr@ee.usc.edu

Abstract

Solving a large-scale system of linear equations is a key step at the heart of many algorithms in machine learning, scientific computing, and beyond. When the problem dimension is large, computational and/or memory constraints make it desirable, or even necessary, to perform the task in a distributed fashion. In this paper, we consider a common scenario in which a taskmaster intends to solve a large-scale system of linear equations by distributing subsets of the equations among a number of computing machines/cores. We propose an accelerated distributed consensus algorithm, in which at each iteration every machine updates its solution by adding a scaled version of the projection of an error signal onto the nullspace of its system of equations, and where the taskmaster conducts an averaging over the solutions with momentum. The convergence behavior of the proposed algorithm is analyzed in detail and analytically shown to compare favorably with the convergence rate of alternative distributed methods, namely distributed gradient descent, distributed versions of Nesterov's accelerated gradient descent and heavy-ball method, the Cimmino method, and ADMM. On randomly chosen linear systems, as well as on real-world data sets, the proposed method offers significant speed-up relative to all the aforementioned methods. Finally, our analysis suggests a novel variation of the distributed heavy-ball method, which employs a particular distributed preconditioning, and which achieves the same theoretical convergence rate as the proposed consensus-based method.

1 Introduction

With the advent of big data, many analytical tasks of interest rely on computations on distributed machines. This is either due to the inherent complexity of the problem, in terms of computation and/or memory, or due to the nature of the large data sets themselves that may already be dispersed across machines. Most algorithms in machine learning and scientific computing have been designed to run on a single machine in their native forms, as a result of which in many cases their distributed solutions have yet to be devised.

Many sophisticated algorithms in machine learning and data analysis are composed of a number of basic computations (e.g., matrix algebra, optimizations, etc). For these computations to run efficiently in a distributed setting, we are required to address a number of technical questions: (1) how is the computation task to be divided among the machines? (2) what is the communication architecture between the machines and the taskmaster and what messages should be communicated? (3) how does the distributed implementation fair in terms of complexity (computational and memory requirements)? and (4) in cases where the distributed computation runs in an iterative fashion what is the convergence rate?

In this paper, we focus on solving a system of linear equations in a distributed fashion, which is one of the most fundamental problems in numerical computation, and lies at the heart of many algorithms in engineering and the sciences. In particular, we consider the setting in which a taskmaster intends to solve a large-scale system of equations with the help of a set of computing machines/cores (Figure 1).

This problem can in general be cast as an optimization problem with a cost function that is separable in the data (but not in the variables) [1]. Hence, there are general approaches to construct distributed algorithms for this problem, such as distributed versions of gradient descent and its variants (e.g. Nesterov's accelerated gradient [15], heavy-ball method [16], etc.), where each machine computes the partial gradient corresponding to a term in the cost and the taskmaster then aggregates the partial gradients by summing them, as well as the so-called Alternating Direction Method of Multipliers (ADMM) and its variants [3]. Among others, some recent approaches for Distributed Gradient Descent (DGD) have been presented and analyzed in [23], [17] and [21], and also coding techniques for robust DGD in the presence of failures and straggler machines have been studied in [11, 20]. ADMM has been widely used [7, 5, 22] for solving various convex optimization problems in a distributed way, and in particular for consensus optimization [8, 18, 13], which is the relevant one for the type of separation that we have here.

In addition to the optimization-based methods, there are a few distributed algorithms designed specifically for solving systems of linear equations. The most famous one of these is what is known as the block Cimmino method [6, 19, 2], which is a block row-projection method [4], and is in a way a distributed implementation of the Kaczmarz method [9]. Another algorithm has been recently proposed in [12, 14], where a consensus-based scheme is used to solve a system of linear equations over a network of autonomous agents. Our algorithm bears some resemblance to all of these methods, but as it will be explained in detail, it has much faster convergence than any of them.

Our main contribution is the design and analysis of a fast distributed consensus-based algorithm to solve large-scale systems of linear equations. More specifically, we develop a methodology in which the taskmaster assigns a subset of equations to each of the machines and invokes a distributed consensus-based algorithm to obtain the solution to the original problem in an iterative manner. At each iteration, each machine updates its solution by projecting an error signal onto the nullspace of its corresponding system of equations and taking a weighted step in that direction. The taskmaster then conducts a memory-augmented averaging on the new solutions provided by the machines. We prove that the algorithm has linear convergence, and we quantify its convergence rate and complexity. Compared to the Kaczmarz/Cimmino-type methods [6, 19, 2] and the consensus scheme proposed in [12, 14], our method is significantly accelerated, due to the momentum incorporated in both projection and averaging steps. For this reason, we refer to our method as Accelerated Projection-Based Consensus (APC). We provide a complete analysis of the convergence rate of APC (Section 3), as well as a detailed comparison with all the other distributed methods mentioned above (Section 4). Also by empirical evaluations over both randomly chosen linear systems and real-world data sets, we demonstrate that the proposed algorithm offers significant speed-ups relative to the other distributed methods (Section 5). Finally, as a further implication of our results, we propose a novel distributed preconditioning method (Section 6), which can be used to improve the convergence rate of distributed gradient-based methods and match that of the APC.

2 The Setup

We consider the problem of solving a large-scale system of linear equations

$$Ax = b, \tag{1}$$

where $A \in \mathbb{R}^{N\times n}$, $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^N$. While we will generally take $N \geq n$, we will assume that the system has a unique solution.[2] For this reason, we will most often consider the square case ($N = n$).

As mentioned before, for large-scale problems (when $N, n \gg 1$), it is highly desirable, or even necessary, to solve the problem in a distributed fashion. Assuming we have m machines, the equations

[1] Solving a system of linear equations, $Ax = b$, can be set up as the optimization problem $\min_x \|Ax-b\|^2 = \min_x \sum_i \|(Ax)_i - b_i\|^2$.

[2] The case when $N > n$ and a solution does not exist, and where a least-squares solution may be sought, will be considered in future work.

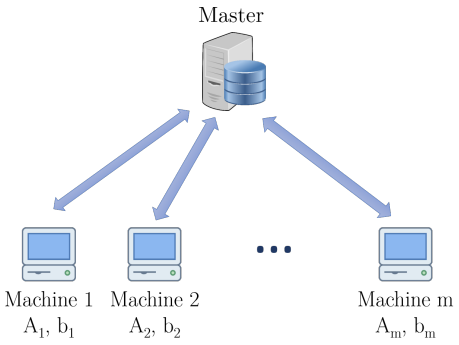

Figure 1: Schematic representation of the taskmaster and the m machines. Each machine i has only a subset of the equations, i.e. $[A_i, b_i]$.

can be partitioned so that each machine gets a disjoint subset of them. In other words, we can write (1) as

$$\begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_m \end{bmatrix} x = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix},$$

where each machine i receives $[A_i, b_i]$. It is worth mentioning that in some applications the data is already stored on different machines in a distributed way. For the sake of simplicity, we assume that m divides N, and that the equations are distributed evenly among the machines, so that each machine gets $p = \frac{N}{m}$ equations. Therefore $A_i \in \mathbb{R}^{p \times n}$ and $b_i \in \mathbb{R}^p$ for every $i = 1, \dots m$. It is helpful to think of p as being relatively small compared to n. In fact, each machine has a system of equations which is highly under-determined.

3 Accelerated Projection-Based Consensus

3.1 The Algorithm

Each machine i can certainly find a solution (among infinitely many) to its own highly under-determined system of equations $A_i x = b_i$, with simply $O(p^3)$ computations. We denote this initial solution by $x_i(0)$. Clearly adding any vector in the right nullspace of A_i to $x_i(0)$ will yield another viable solution. The challenge is to find vectors in the nullspaces of each of the A_i's in such a way that all the solutions for different machines coincide.

At each iteration t, the master provides the machines with an estimate of the solution, denoted by $\bar{x}(t)$. Each machine then updates its value $x_i(t)$ by projecting its difference from the estimate onto the nullspace, and taking a weighted step in that direction. Mathematically

$$x_i(t+1) = x_i(t) + \gamma P_i(\bar{x}(t) - x_i(t)),$$

where $P_i = I - A_i^T (A_i A_i^T)^{-1} A_i$ is the projection matrix onto the nullspace of A_i (It is easy to check that $A_i P_i = 0$ and $P_i^2 = P_i$).

Although this might have some resemblance to the block Cimmino method because of the projections, as it will be explained, the convergence time of the APC method is much better (by a square root) than that of the block Cimmino method. Moreover, it turns out that the block Cimmino method is in fact a special case of APC for $\gamma = 1$.

The update rule of $x_i(t+1)$ described above can be also thought of as the solution to an optimization problem with two terms, the distance from the global estimate $\bar{x}(t)$ and the distance from the previous solution $x_i(t)$. In other words, one can show that

$$\begin{aligned} x_i(t+1) = \underset{x_i}{\text{argmin}} \quad & \|x_i - \bar{x}(t)\|^2 + \frac{1-\gamma}{\gamma} \|x_i - x_i(t)\|^2 \\ \text{s.t.} \quad & A_i x_i = b_i \end{aligned}$$

Algorithm 1 APC: Accelerated Projection-Based Consensus (for solving $Ax = b$ distributedly)

Input: data $[A_i, b_i]$ for each machine $i = 1, \dots m$, parameters η, γ
Initialization: at each machine i find a solution $x_i(0)$ (among infinitely many) to $A_i x = b_i$.
for $t = 1$ **to** T **do**
  for each machine i **parallel do**
    $x_i(t+1) \leftarrow x_i(t) + \gamma P_i(\bar{x}(t) - x_i(t))$
  end for
  at the master: $\bar{x}(t+1) \leftarrow \frac{\eta}{m} \sum_{i=1}^{m} x_i(t+1) + (1-\eta)\bar{x}(t)$
end for

The second term in the objective is what distinguishes this method from the block Cimmino method. If one sets γ equal to 1 (which, as it will be shown in Section 4.5, is the reduction to the block Cimmino method), the second term completely disappears, and the update no longer depends on $x_i(t)$. This has a dramatic impact on the convergence rate, as it will be shown.

The master collects the updated values $x_i(t+1)$ to form a new estimate $\bar{x}(t+1)$. A plausible choice is to simply take the average, i.e.,

$$\bar{x}(t+1) = \frac{1}{m} \sum_{i=1}^{m} x_i(t+1).$$

This update works, and is what appears both in ADMM and in the consensus method of [12, 14]. But it turns out that it is extremely slow. Instead, we take an affine combination of the average and the previous estimate as

$$\bar{x}(t+1) = \frac{\eta}{m} \sum_{i=1}^{m} x_i(t+1) + (1-\eta)\bar{x}(t),$$

which introduces a one-step memory, and behaves like a momentum.

The resulting update rule is therefore

$$x_i(t+1) = x_i(t) + \gamma P_i(\bar{x}(t) - x_i(t)), \quad i = 1, \dots, m, \tag{2a}$$

$$\bar{x}(t+1) = \frac{\eta}{m} \sum_{i=1}^{m} x_i(t+1) + (1-\eta)\bar{x}(t), \tag{2b}$$

which leads to Algorithm 1.

3.2 Convergence Analysis

We analyze the convergence of the proposed algorithm and prove that it has linear convergence with no additional assumption imposed. We also derive the rate of convergence explicitly.

Let us define the matrix $X \in \mathbb{R}^{n \times n}$ by

$$X \triangleq \frac{1}{m} \sum_{i=1}^{m} A_i^T (A_i A_i^T)^{-1} A_i. \tag{3}$$

As it will become clear soon, the condition number of this matrix predicts the behavior of the algorithm. Note that since the eigenvalues of the projection matrix P_i are all 0 and 1, for every i, the eigenvalues of X are all between 0 and 1. Denoting the eigenvalues of X by μ_i, we have:

$$0 \leq \mu_{\min} \triangleq \mu_n \leq \cdots \leq \mu_1 \triangleq \mu_{\max} \leq 1. \tag{4}$$

Let us define quadratic polynomials $p_i(\lambda)$ characterized by γ and η as

$$p_i(\lambda; \gamma, \eta) \triangleq \lambda^2 + (-\eta\gamma(1-\mu_i) + \gamma - 1 + \eta - 1)\,\lambda + (\gamma - 1)(\eta - 1) \tag{5}$$

for $i = 1, \dots, n$. Further, define set S as the collection of pairs $\gamma \in [0, 2]$ and $\eta \in \mathbb{R}$ for which the largest magnitude solution of $p_i(\lambda) = 0$ among every i is less than 1. Mathematically

$$S = \{(\gamma, \eta) \in [0, 2] \times \mathbb{R} \mid \forall \lambda \in \mathbb{C} \text{ and } \forall i \text{ for which } p_i(\lambda; \gamma, \eta) = 0 \text{ we have } |\lambda| < 1\}. \tag{6}$$

The following result summarizes the convergence behavior of the proposed algorithm.

Theorem 1. *Algorithm 1 converges to the true solution as fast as ρ^t converges to 0, as $t \to \infty$, for some $\rho \in (0,1)$, if and only $(\gamma, \eta) \in S$. Furthermore, the optimal rate of convergence is*

$$\rho = \frac{\sqrt{\kappa(X)} - 1}{\sqrt{\kappa(X)} + 1} \approx 1 - \frac{2}{\sqrt{\kappa(X)}}, \tag{7}$$

where $\kappa(X) = \frac{\mu_{\max}}{\mu_{\min}}$ is the condition number of X, and the optimal parameters (γ^*, η^*) are the solutions to the following equations

$$\begin{cases} \mu_{\max}\eta\gamma = (1 + \sqrt{(\gamma-1)(\eta-1)})^2, \\ \mu_{\min}\eta\gamma = (1 - \sqrt{(\gamma-1)(\eta-1)})^2. \end{cases}$$

For proof see the supplementary material.

3.3 Computational Complexity and Numerical Stability

In addition to the convergence rate, or equivalently the number of iterations until convergence, one needs to consider the computational complexity per iteration.

At each iteration, since $P_i = I_n - A_i^T(A_iA_i^T)^{-1}A_i$, and A_i is $p \times n$, each machine has to do the following two matrix-vector multiplications: 1) $A_i(x_i(t) - \bar{x}(t))$, which takes pn scalar multiplications, and 2) $\left(A_i^T(A_iA_i^T)^{-1}\right)$ times the vector from the previous step, which takes another np operations. Thus the overall computational complexity of each iteration is $2pn$.

Finally, we should mention that the computation done at each machine during each iteration is essentially a projection, which has condition number one and is as numerically stable as a matrix vector multiplication can be.

4 Related Methods

4.1 Distributed Gradient Descent (DGD)

As mentioned earlier, (1) can also be viewed as an optimization problem of the form

$$\underset{x}{\text{minimize}} \ \|Ax - b\|^2,$$

and since the objective is separable in the data, i.e. $\|Ax - b\|^2 = \sum_{i=1}^m \|A_ix - b_i\|^2$, generic distributed optimization methods such as distributed gradient descent apply well to the problem.

The regular or full gradient descent has the update rule $x(t+1) = x(t) - \alpha A^T(Ax(t) - b)$, where $\alpha > 0$ is the step size or learning rate. The distributed version of gradient descent is one in which each machine i has only a subset of the equations $[A_i, b_i]$, and computes its own part of the gradient, which is $A_i^T(A_ix(t) - b_i)$. The updates are then collectively done as:

$$x(t+1) = x(t) - \alpha \sum_{i=1}^m A_i^T(A_ix(t) - b_i). \tag{8}$$

One can show that this also has linear convergence, and the rate of convergence is

$$\rho = \frac{\kappa(A^TA) - 1}{\kappa(A^TA) + 1} \approx 1 - \frac{2}{\kappa(A^TA)}. \tag{9}$$

We should mention that since each machine needs to compute $A_i^T(A_ix(t) - b_i)$ at each iteration t, the computational complexity per iteration is $2pn$, which is identical to that of APC.

4.2 Distributed Nesterov's Accelerated Gradient Descent (D-NAG)

A popular variant of gradient descent is Nesterov's accelerated gradient descent [15], which has a memory term, and works as follows:

$$y(t+1) = x(t) - \alpha \sum_{i=1}^m A_i^T(A_ix(t) - b_i), \tag{10a}$$

$$x(t+1) = (1+\beta)y(t+1) - \beta y(t). \tag{10b}$$

Table 1: A summary of the convergence rates of different methods. The smaller the convergence rate is, the faster is the method. DGD: Distributed Gradient Descent, D-NAG: Distributed Nesterov's Accelerated Gradient Descent, D-HBM Distributed Heavy-Ball Method, Consensus: Standard Projection-Based Consensus, B-Cimmino: Block Cimmino Method, APC: Accelerated Projection-based Consensus.

DGD	D-NAG	D-HBM	Consensus	B-Cimmino	APC (proposed)
$1-\frac{2}{\kappa(A^TA)}$	$1-\frac{2}{\sqrt{3\kappa(A^TA)+1}}$	$1-\frac{2}{\sqrt{\kappa(A^TA)}}$	$1-\mu_{\min}(X)$	$1-\frac{2}{\kappa(X)}$	$1-\frac{2}{\sqrt{\kappa(X)}}$

One can show [10] that the optimal convergence rate of this method is

$$\rho = 1 - \frac{2}{\sqrt{3\kappa(A^TA)+1}}, \tag{11}$$

which is improved over the regular distributed gradient descent.

4.3 Distributed Heavy-Ball Method (D-HBM)

The heavy-ball method [16], otherwise known as the gradient descent with momentum, is another accelerated variant of gradient descent as follows:

$$z(t+1) = \beta z(t) + \sum_{i=1}^{m} A_i^T(A_i x(t) - b_i), \tag{12a}$$

$$x(t+1) = x(t) - \alpha z(t+1). \tag{12b}$$

It can be shown [10] that the optimal rate of convergence of this method is

$$\rho = \frac{\sqrt{\kappa(A^TA)}-1}{\sqrt{\kappa(A^TA)}+1} \approx 1 - \frac{2}{\sqrt{\kappa(A^TA)}}, \tag{13}$$

which is further improved over DGD and D-NAG, and is similar to, but not the same as, APC. The difference is that the condition number of $A^TA = \sum_{i=1}^m A_i^TA_i$ is replaced with the condition number of $X = \sum_{i=1}^m A_i^T\left(A_iA_i^T\right)^{-1}A_i$ in APC. Given its structure as the sum of projection matrices, one may speculate that X has a much better condition number than A^TA. Indeed, our experiments with random, as well as real, data sets suggest that this is the case and that the condition number of X is often significantly better.

4.4 Alternating Direction Method of Multipliers (ADMM)

Alternating Direction Method of Multipliers (more specifically, consensus ADMM [18, 3]), is another generic method for solving optimization problems with separable cost function $f(x) = \sum_{i=1}^m f_i(x)$ distributedly, by defining additional local variables. Each machine i holds local variables $x_i(t) \in \mathbb{R}^n$ and $y_i(t) \in \mathbb{R}^n$, and the master's value is $\bar{x}(t) \in \mathbb{R}^n$, for any time t. For $f_i(x) = \frac{1}{2}\|A_ix - b_i\|^2$, the update rule of ADMM simplifies to

$$x_i(t+1) = (A_i^TA_i + \xi I_n)^{-1}(A_i^Tb_i - y_i(t) + \xi\bar{x}(t)), \quad i = 1,\dots,m \tag{14a}$$

$$\bar{x}(t+1) = \frac{1}{m}\sum_{i=1}^{m} x_i(t+1) \tag{14b}$$

$$y_i(t+1) = y_i(t) + \xi(x_i(t+1) - \bar{x}(t+1)), \quad i = 1,\dots,m \tag{14c}$$

It turns out that this method is very slow (and often unstable) in its native form for the application in hand. One can check that when system (1) has a solution all the y_i variables converge to zero in steady state. Therefore setting y_i's to zero can speed up the convergence significantly. We use this modified version in Section 5, to compare with.

We should also note that the computational complexity of ADMM is $O(pn)$ per iteration (the inverse is computed using matrix inversion lemma), which is again the same as that of gradient-type methods and APC.

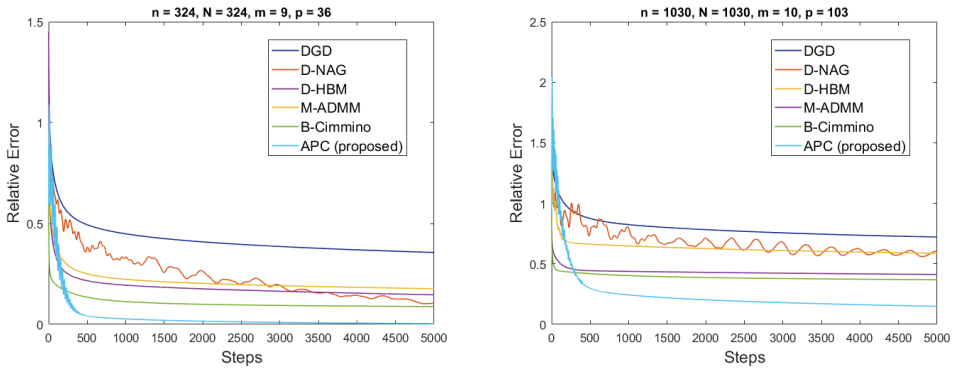

Figure 2: The decay of the error for different distributed algorithms, on two real problems from Matrix Market [1] (QC324: Model of H_2^+ in an Electromagnetic Field, and ORSIRR 1: Oil reservoir simulation).

4.5 Block Cimmino Method

The Block Cimmino method [6, 19, 2], which is a parallel method specifically for solving linear systems of equations, is perhaps the closest algorithm in spirit to APC. It is, in a way, a distributed implementation of the so-called Kaczmarz method [9]. The convergence of the Cimmino method is slower by an order in comparison with APC (its convergence time is the square of that of APC), and it turns out that APC includes this method as a special case when $\gamma = 1$.

The block Cimmino method is the following:

$$r_i(t) = A_i^+(b_i - A_i\bar{x}(t)), \quad i = 1, \dots, m \tag{15a}$$

$$\bar{x}(t+1) = \bar{x}(t) + \nu \sum_{i=1}^{m} r_i(t), \tag{15b}$$

where $A_i^+ = A_i^T(A_iA_i^T)^{-1}$ is the pseudoinverse of A_i.

Proposition 2. *The APC method (Algorithm 1) includes the block Cimmino method as a special case for* $\gamma = 1$.

The proof is provided in the supplementary material.

It is not hard to show that optimal rate of convergence of the Cimmino method is

$$\rho = \frac{\kappa(X) - 1}{\kappa(X) + 1} \approx 1 - \frac{2}{\kappa(X)}, \tag{16}$$

which is by an order worse than that of APC ($1 - \frac{2}{\sqrt{\kappa(X)}}$).

A summary of the optimal convergence rates of all the methods is provided in Table 1

5 Experimental Results

In this section, we evaluate the proposed method (APC) by comparing it with the other distributed methods discussed throughout the paper, namely DGD, D-NAG, D-HBM, ADMM, and block Cimmino methods. We use randomly-generated problems as well as real-world ones form the National Institute of Standards and Technology repository, *Matrix Market* [1].

Fig. 2 shows the relative error (the distance from the true solution, divided by the true solution, in ℓ_2 norm) for all the discussed methods, on two examples from the repository, namely QC324: Model of H_2^+ in an Electromagnetic Field, and ORSIRR 1: Oil Reservoir Simulation. To make the comparison between different methods fair, we have tuned the parameters in all of them to their optimal values. Also as mentioned before, all the algorithms have the same per-iteration complexity. As one can see, APC outperforms the other methods significantly.

Table 2: A comparison between the optimal convergence time T ($= \frac{1}{-\log \rho}$) of different methods on real and synthetic examples. Boldface values show the smallest convergence time.

	DGD	D-NAG	D-HBM	M-ADMM	B-CIMMINO	APC
QC324 (324×324)	1.22×10^7	4.28×10^3	2.47×10^3	1.07×10^7	3.10×10^5	$\mathbf{3.93 \times 10^2}$
ORSIRR 1 (1030×1030)	2.98×10^9	6.68×10^4	3.86×10^4	2.08×10^8	2.69×10^7	$\mathbf{3.67 \times 10^3}$
ASH608 (608×188)	5.67×10^0	2.43×10^0	1.64×10^0	1.28×10^1	4.98×10^0	$\mathbf{1.53 \times 10^0}$
STANDARD GAUSSIAN (500×500)	1.76×10^7	5.14×10^3	2.97×10^3	1.20×10^6	1.46×10^7	$\mathbf{2.70 \times 10^3}$
NONZERO-MEAN GAUSSIAN (500×500)	2.22×10^{10}	1.82×10^5	1.05×10^5	8.62×10^8	9.29×10^8	$\mathbf{2.16 \times 10^4}$
STANDARD TALL GAUSSIAN (1000×500)	1.58×10^1	4.37×10^0	2.78×10^0	4.49×10^1	1.13×10^1	$\mathbf{2.34 \times 10^0}$

Table 2 shows the convergence time $T = \frac{1}{-\log \rho} \approx \frac{1}{1-\rho}$ of different distributed algorithms on a number of synthetic and real-world problems with different sizes. It can be seen that APC has a much faster convergence, often by orders of magnitude. As expected from the analysis, the APC's closest competitor is the distributed heavy-ball method.

6 A Distributed Preconditioning to Improve Gradient-Based Methods

The noticeable similarity between the optimal convergence rate of APC ($\frac{\sqrt{\kappa(X)}-1}{\sqrt{\kappa(X)}+1}$) and that of D-HBM ($\frac{\sqrt{\kappa(A^TA)}-1}{\sqrt{\kappa(A^TA)}+1}$) suggests that there might be a connection between the two. It turns out that there is, and we propose a distributed preconditioning for D-HBM, which makes it achieve the same convergence rate as APC. The algorithm is as follows.

Prior to starting the iterative process, each machine i can premultiply its own set of equations $A_i x = b_i$ by $(A_i A_i^T)^{-1/2}$, which can be done in parallel (locally) with $O(p^2 n)$ operations. This transforms the global system of equations $Ax = b$ to a new one $Cx = d$, where

$$C = \begin{bmatrix} (A_1 A_1^T)^{-1/2} A_1 \\ \vdots \\ (A_m A_m^T)^{-1/2} A_m \end{bmatrix}, \quad \text{and} \quad d = \begin{bmatrix} (A_1 A_1^T)^{-1/2} b_1 \\ \vdots \\ (A_m A_m^T)^{-1/2} b_m \end{bmatrix}.$$

The new system can then be solved using distributed heavy-ball method, which will achieve the same rate of convergence as APC, i.e. $\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$ where $\kappa = \kappa(C^TC) = \kappa(X)$.

7 Conclusion

We considered the problem of solving a large-scale system of linear equations by a taskmaster with the help of a number of computing machines/cores, in a distributed way. We proposed an accelerated projection-based consensus algorithm for this problem, and fully analyzed its convergence rate. Analytical and experimental comparisons with the other known distributed methods confirm significantly faster convergence of the proposed scheme. Finally, our analysis suggested a novel distributed preconditioning for improving the convergence of the distributed heavy-ball method to achieve the same theoretical performance as the proposed consensus-based method.

References

[1] Matrix market. `http://math.nist.gov/MatrixMarket/`. Accessed: May 2017.

[2] Mario Arioli, Iain Duff, Joseph Noailles, and Daniel Ruiz. A block projection method for sparse matrices. *SIAM Journal on Scientific and Statistical Computing*, 13(1):47–70, 1992.

[3] 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.

[4] Randall Bramley and Ahmed Sameh. Row projection methods for large nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(1):168–193, 1992.

[5] Wei Deng and Wotao Yin. On the global and linear convergence of the generalized alternating direction method of multipliers. *Journal of Scientific Computing*, 2012.

[6] Iain S Duff, Ronan Guivarch, Daniel Ruiz, and Mohamed Zenadi. The augmented block Cimmino distributed method. *SIAM Journal on Scientific Computing*, 37(3):A1248–A1269, 2015.

[7] Bingsheng He and Xiaoming Yuan. On the $O(1/n)$ convergence rate of the Douglas-Rachford alternating direction method. *SIAM Journal on Numerical Analysis*, 2012.

[8] P. M. Aguiar J. F. Mota, J. M. Xavier and M. Puschel. D-ADMM: A communication-efficient distributed algorithm for separable optimization. *IEEE Transactions on Signal Processing*, 2013.

[9] Stefan Kaczmarz. Angenäherte auflösung von systemen linearer gleichungen. *Bulletin International de l'Academie Polonaise des Sciences et des Lettres*, 35:355–357, 1937.

[10] Laurent Lessard, Benjamin Recht, and Andrew Packard. Analysis and design of optimization algorithms via integral quadratic constraints. *SIAM Journal on Optimization*, 26(1):57–95, 2016.

[11] Songze Li, Qian Yu, Mohammad Ali Maddah-Ali, and Amir Salman Avestimehr. A unified coding framework for distributed computing with straggling servers. In *Proceedings of the 2016 IEEE Workshop on Network Coding and Applications (NetCod)*, Dec. 2016.

[12] Ji Liu, Shaoshuai Mou, and A Stephen Morse. An asynchronous distributed algorithm for solving a linear algebraic equation. In *Decision and Control (CDC), 2013 IEEE 52nd Annual Conference on*, pages 5409–5414. IEEE, 2013.

[13] Layla Majzoubi and Farshad Lahouti. Analysis of distributed ADMM algorithm for consensus optimization in presence of error. In *Proceedings of the 2016 IEEE Confs. Audio, Speech and Signal Processing (ICASSP)*, Mar. 2016.

[14] Shaoshuai Mou, Ji Liu, and A Stephen Morse. A distributed algorithm for solving a linear algebraic equation. *IEEE Transactions on Automatic Control*, 60(11):2863–2878, 2015.

[15] Yurii Nesterov. A method of solving a convex programming problem with convergence rate $O(1/k^2)$. In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

[16] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[17] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[18] Wei Shi, Qing Ling, Kun Yuan, Gang Wu, and Wotao Yin. On the linear convergence of the ADMM in decentralized consensus optimization. *IEEE Trans. Signal Processing*, 62(7):1750–1761, 2014.

[19] Fridrich Sloboda. A projection method of the Cimmino type for linear algebraic systems. *Parallel computing*, 17(4-5):435–442, 1991.

[20] Rashish Tandon, Qi Lei, Alexandros G Dimakis, and Nikos Karampatziakis. Gradient coding. *arXiv preprint arXiv:1612.03301*, 2016.

[21] Kun Yuan, Qing Ling, and Wotao Yin. On the convergence of decentralized gradient descent. *SIAM Journal on Optimization*, 26(3):1835–1854, 2016.

[22] Ruiliang Zhang and James T Kwok. Asynchronous distributed ADMM for consensus optimization. In *ICML*, pages 1701–1709, 2014.

[23] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603, 2010.

Supplementary Material

Proof of Theorem 1. Take x^* to be the unique solution of $Ax = b$. Define the errors with respect to x^* as $e_i(t) = x_i(t) - x^*$ for all $i = 1 \dots m$, and $\bar{e}(t) = \bar{x}(t) - x^*$. We can write Eq. (2a) as

$$e_i(t+1) = e_i(t) + \gamma P_i(\bar{e}(t) - e_i(t)).$$

Since both x^* and $x_i(t)$ are solutions to $A_i x = b_i$, their difference, which is $e_i(t)$, is in the nullspace of A_i, and therefore it remains the same under projection onto the nullspace. Hence $P_i e_i(t) = e_i(t)$, and we have

$$e_i(t+1) = (1-\gamma)e_i(t) + \gamma P_i \bar{e}(t), \tag{17}$$

for all $i = 1 \dots, m$. Similarly, the recursion (2b) can be expressed as

$$\bar{e}(t+1) = \frac{\eta}{m} \sum_{i=1}^{m} e_i(t+1) + (1-\eta)\bar{e}(t),$$

which using (17) becomes

$$\begin{aligned} \bar{e}(t+1) &= \frac{\eta}{m} \sum_{i=1}^{m} ((1-\gamma)e_i(t) + \gamma P_i \bar{e}(t)) + (1-\eta)\bar{e}(t) \\ &= \frac{\eta(1-\gamma)}{m} \sum_{i=1}^{m} e_i(t) + \left(\frac{\eta\gamma}{m} \sum_{i=1}^{m} P_i + (1-\eta) I_n \right) \bar{e}(t). \end{aligned} \tag{18}$$

All the m vectors e_i along with the average $\bar{e}$ can be stacked up together as a vector $e(t)^T = [e_1(t)^T, e_2(t)^T, \dots, e_m(t)^T, \bar{e}(t)^T] \in \mathbb{R}^{(m+1)n}$. The update rule can then be expressed as:

$$\begin{bmatrix} e_1(t+1) \\ \vdots \\ e_m(t+1) \\ \bar{e}(t+1) \end{bmatrix} = \begin{bmatrix} (1-\gamma) I_{mn} & \gamma \begin{bmatrix} P_1 \\ \vdots \\ P_m \end{bmatrix} \\ \frac{\eta(1-\gamma)}{m} [I_n \dots I_n] & M \end{bmatrix} \begin{bmatrix} e_1(t) \\ \vdots \\ e_m(t) \\ \bar{e}(t) \end{bmatrix}, \tag{19}$$

where $M = \frac{\eta\gamma}{m} \sum_{i=1}^{m} P_i + (1-\eta) I_n$.

The convergence of the algorithm is determined by the largest magnitude eigenvalue of the $(m+1)n \times (m+1)n$ block matrix in (19). The eigenvalues λ_i of this matrix are the solutions to the following characteristic equation:

$$\det \begin{bmatrix} (1-\gamma-\lambda) I_{mn} & \gamma \begin{bmatrix} P_1 \\ \vdots \\ P_m \end{bmatrix} \\ \frac{\eta(1-\gamma)}{m} [I_n \dots I_n] & \frac{\eta\gamma}{m} \sum_{i=1}^{m} P_i + (1-\eta-\lambda) I_n \end{bmatrix} = 0,$$

which using the Schur complement can be written as follows.

$$\begin{aligned} 0 &= (1-\gamma-\lambda)^{mn} \det \left(\frac{\eta\gamma}{m} \sum_{i=1}^{m} P_i + (1-\eta-\lambda) I_n - \frac{\eta(1-\gamma)\gamma}{(1-\gamma-\lambda)m} \sum_{i=1}^{m} P_i \right) \\ &= (1-\gamma-\lambda)^{mn} \det \left(\frac{\eta\gamma}{m} (1 - \frac{1-\gamma}{1-\gamma-\lambda}) \sum_{i=1}^{m} P_i + (1-\eta-\lambda) I_n \right) \\ &= (1-\gamma-\lambda)^{mn} \det \left(\frac{-\eta\gamma\lambda}{(1-\gamma-\lambda)m} \sum_{i=1}^{m} P_i + (1-\eta-\lambda) I_n \right) \\ &= (1-\gamma-\lambda)^{(m-1)n} \det \left(-\eta\gamma\lambda \frac{\sum_{i=1}^{m} P_i}{m} + (1-\gamma-\lambda)(1-\eta-\lambda) I_n \right). \end{aligned}$$

From now on, for simplicity, whenever the identity matrix is of size n, we drop its subscript. Since $P_i = I - A_i^T(A_iA_i^T)^{-1}A_i$, we have $\frac{1}{m}\sum_{i=1}^m P_i = I - \frac{1}{m}\sum_{i=1}^m A_i^T(A_iA_i^T)^{-1}A_i = I - X$. There are $(m-1)n$ eigenvalues equal to $1-\gamma$, and the rest $2n$ eigenvalues are the solutions to

$$\begin{aligned} 0 &= \det\left(-\eta\gamma\lambda(I-X) + (1-\gamma-\lambda)(1-\eta-\lambda)I\right) \\ &= \det\left(\eta\gamma\lambda X + ((1-\gamma-\lambda)(1-\eta-\lambda) - \eta\gamma\lambda)\, I\right). \end{aligned}$$

Recall that the eigenvalues of X are denoted by $\mu_i,\ i = 1,\dots,n$. Therefore, the eigenvalues of $\eta\gamma\lambda X + ((1-\gamma-\lambda)(1-\eta-\lambda) - \eta\gamma\lambda)\, I$ are $\eta\gamma\lambda\mu_i + (1-\gamma-\lambda)(1-\eta-\lambda) - \eta\gamma\lambda,\ i = 1,\dots,n$. The above determinant can then be written as the product of the eigenvalues of the matrix inside it, as

$$0 = \prod_{i=1}^n \eta\gamma\lambda\mu_i + (1-\gamma-\lambda)(1-\eta-\lambda) - \eta\gamma\lambda.$$

Therefore, there are two eigenvalues $\lambda_{i,1}, \lambda_{i,2}$ as the solution to the quadratic equation

$$\lambda^2 + (-\eta\gamma(1-\mu_i) + \gamma - 1 + \eta - 1)\lambda + (\gamma-1)(\eta-1) = 0$$

for every $i = 1,\dots,n$, which will constitute the $2n$ eigenvalues. When all these eigenvalues, along with $1-\gamma$, are less than 1, the error converges to zero as ρ^t, with ρ being the largest magnitude eigenvalue (spectral radius).

The spectral radius is minimized when all these eigenvalues are complex and have magnitude $|\lambda_{i,1}| = |\lambda_{i,2}| = \sqrt{(\gamma-1)(\eta-1)} = \rho$. For that to happen, we should have

$$(\gamma + \eta - \eta\gamma(1-\mu_i) - 2)^2 \le 4(\gamma-1)(\eta-1), \quad \forall i,$$

or equivalently

$$-2\sqrt{(\gamma-1)(\eta-1)} \le \gamma + \eta - \eta\gamma(1-\mu_i) \le 2\sqrt{(\gamma-1)(\eta-1)}, \quad \forall i.$$

The expression in between is increasing in μ_i, and therefore for that to hold, it is enough for the lower bound to hold for the $\mu_{\min}$ and the upper bound to hold for $\mu_{\max}$, i.e.

$$\begin{cases} \gamma + \eta - \eta\gamma(1-\mu_{\max}) - 2 = 2\sqrt{(\gamma-1)(\eta-1)} \\ 2 + \eta\gamma(1-\mu_{\min}) - \gamma - \eta = 2\sqrt{(\gamma-1)(\eta-1)} \end{cases}.$$

This can be simplified to

$$\begin{cases} \mu_{\max}\eta\gamma = (1 + \sqrt{(\gamma-1)(\eta-1)})^2 = (1+\rho)^2 \\ \mu_{\min}\eta\gamma = (1 - \sqrt{(\gamma-1)(\eta-1)})^2 = (1-\rho)^2 \end{cases},$$

which implies $\kappa(X) = \frac{(1+\rho)^2}{(1-\rho)^2}$, and results in

$$\rho = \frac{\sqrt{\kappa(X)} - 1}{\sqrt{\kappa(X)} + 1}$$

□

Proof of Proposition 2. When $\gamma = 1$ in APC, Eq. (2a) becomes

$$\begin{aligned} x_i(t+1) &= x_i(t) - \left(I - A_i^T(A_iA_i^T)^{-1}A_i\right)(x_i(t) - \bar{x}(t)) \\ &= \bar{x}(t) + A_i^T(A_iA_i^T)^{-1}A_i(x_i - \bar{x}(t)) \\ &= \bar{x}(t) + A_i^T(A_iA_i^T)^{-1}(b_i - A_i\bar{x}(t)) \end{aligned}$$

This is no longer an "update" (it does not depend on $x_i(t)$), and it is immediate from (15a) that

$$r_i(t) = x_i(t+1) - \bar{x}(t).$$

Therefore, update (15b) can be expressed as

$$\begin{aligned} \bar{x}(t+1) &= \bar{x}(t) + \nu\sum_{i=1}^m (x_i(t+1) - \bar{x}(t)) \\ &= (1-m\nu)\bar{x}(t) + \nu\sum_{i=1}^m x_i(t+1), \end{aligned}$$

which is nothing but the same update as in (2b) with $\eta = m\nu$. □