Parallel and Distributed Sparse Optimization

Zhimin Peng Ming Yan Wotao Yin

Department of Mathematics, University of California, Los Angeles
Emails: *zhimin.peng, yanm, wotaoyin@math.ucla.edu*

***Abstract*—This paper proposes parallel and distributed algorithms for solving very large-scale sparse optimization problems on computer clusters and clouds. Modern datasets usually have a large number of features or training samples, and they are usually stored in a distributed manner. Motivated by the need of solving sparse optimization problems with large datasets, we propose two approaches including (i) distributed implementations of prox-linear algorithms and (ii) GRock, a parallel greedy block coordinate descent method. Different separability properties of the objective terms in the problem enable different data distributed schemes along with their corresponding algorithm implementations. We also establish the convergence of GRock and explain why it often performs exceptionally well for sparse optimization. Numerical results on a computer cluster and Amazon EC2 demonstrate the efficiency and elasticity of our algorithms.**

***Keywords*—sparse optimization, ℓ_1 minimization, LASSO, parallel and distributed computing, GRock**

I. Introduction

Technological advances in data gathering have led to a rapid proliferation of big data in diverse areas such as the Internet, engineering, climate studies, cosmology and medicine. In order for this massive amount of data to make sense, new computational approaches are being introduced to let scientists and engineers analyze their data in a parallel and distributed manner. Among these approaches, structured solutions, and in particular sparse solutions, have grown enormously important. They play important roles in statistics (LASSO [1] and sparse logistic regression), machine learning (sparse SVM and PCA [2], [3]), image processing (total variation [4]), seismic data processing, and more recently in compressive sensing [5], [6], speech and text processing, bioinformatics and many others. Many modern applications of sparse optimization involve very large-scale data, so large that they can no longer be processed on a single workstation running single-threaded codes. Moving to parallel/distributed/cloud computing becomes a viable option. This paper introduces two novel approaches for large-scale sparse optimization problems that leverage the vast amounts of computing resources available.

Our two approaches are motivated by two structures present in sparse optimization: separable objective functions, and data "near-orthogonality", which we shall describe in the next section. In the first approach, we parallelize the existing prox-linear algorithms such as ISTA, FPC [7], and FISTA [8] by taking advantages of the separability of the terms in the objective. The second approach, GRock, is developed based on greedy block coordinate update. At each iteration, it selects a few blocks of variables and then a few variables in each selected block, both by greedy means, to update in parallel. Although greedy selections are seemingly awkward for parallel and distributed computation, we argue the exact opposite for a broad subclass of sparse optimization.

The proposed approaches have significant advantages over the existing ones. A well-known approach is based on the alternating direction method of multipliers (ADMM), also known as an operator splitting scheme. For many convex problems including those in sparse optimization, it gives rise to their parallel and distributed algorithms [9]. However, the approach in [9] does not scale well; given a fixed amount of data, distributing the data and ADMM computation to more nodes do not reduce its running time, because its number of iterations increases with the number of distributed data blocks. The time saved due to a smaller block size is offset by the increased number of iterations. In contrast, the introduced approaches do not increase the iterations so will run faster as the data are distributed to more nodes.

Another existing approach is parallel coordinate descent. For instance, Bradley, Kyrola, Bickson and Guestrin [10] developed the Shotgun algorithm; Scherrer, Halappanavar, Tewari and Haglin [11] proposed a generic framework for expressing parallel coordinate descent algorithms; recent work [12] iteratively and randomly selects multiple blocks of variables to update in parallel. Under certain orthogonality conditions, our proposed greedy approach needs much fewer iterations while having nearly the same per-iteration cost.

The source codes of the parallel and distributed implementations of our approaches are accessible from our personal websites. In this paper, we compare the scalability of different approaches and present results of solving problems with 170GB of data on Amazon EC2 in merely 1.7 minutes to reach a relative error of 10^{-5}.

II. Problem formulation and data distribution

We study a convex optimization formulation underlying many sparse optimization problems

$$\min_{\mathbf{x}\in\mathbb{R}^n} \; F(\mathbf{x}) = \lambda \cdot R(\mathbf{x}) + L(\mathbf{A}\mathbf{x}, \mathbf{b}), \tag{1}$$

where $R(\mathbf{x})$ is a (possibly nonsmooth) regularizer that imposes a certain structure to the solution and $L(\mathbf{A}\mathbf{x}, \mathbf{b})$ is a (typically smooth) data fidelity or loss function.

A function $f(\mathbf{x})$ is *block separable* if it can be written as $f(\mathbf{x}) = \sum_{s=1}^{S} f_s(\mathbf{x}_s)$, where $\mathbf{x}_s$ is the sth block of $\mathbf{x}$. It is *partially block separable* if it can be written as $f(\mathbf{x}) = \sum_{s=1}^{S} f_s(\mathbf{x})$.

For the commonly used regularizers R such as the ℓ_1 norm, $\ell_{2,1}$ norm, Huber function, and elastic net regularizer, we can write $R(\mathbf{x}) = \sum_{i=1}^{N} r(\mathbf{x}_i)$. Many of the common examples

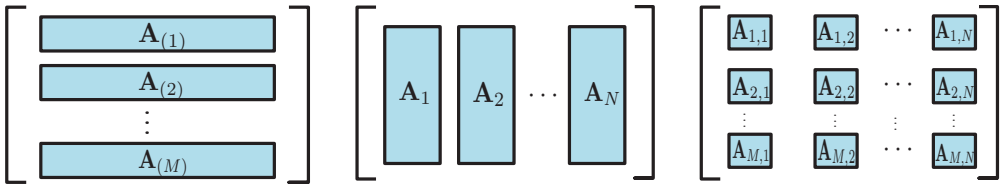


Fig. 1: data distribution scenarios.

of loss functions such as the square, logistic, and hinge loss functions satisfy $L(\mathbf{Ax}, \mathbf{b}) = \sum_{j=1}^{M} L(\mathbf{A}_{(j)}\mathbf{x}, \mathbf{b}_j)$ where $A_{(j)}$ is the jth row block of $\mathbf{A}$. Between R and L, we will need one of them to be separable as such.

Matrix $\mathbf{A}$ must be stored in a distributed manner when it is too large to store centrally, or when it is collected in a distributed manner. Hence, solving (1) will involve computing $\mathbf{Ax}$ and $\mathbf{A}^T\mathbf{y}$ (multiple times with different $\mathbf{x}$ and $\mathbf{y}$) in a distributed manner. We discuss three scenarios (shown in Fig. 1):

a. row block distribution: $\mathbf{A}$ is partitioned into row-blocks $\mathbf{A}_{(1)}, \mathbf{A}_{(2)}, \ldots, \mathbf{A}_{(M)}$. Stacking them forms $\mathbf{A}$.

b. column block distribution: $\mathbf{A} = [\mathbf{A}_1 \ \mathbf{A}_2 \ \cdots \ \mathbf{A}_N]$.

c. general block distribution: $\mathbf{A}$ is partitioned into MN sub-blocks. The (i, j)th block is $\mathbf{A}_{i,j}$.

Computing

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{A}_{(1)}\mathbf{x} \\ \mathbf{A}_{(2)}\mathbf{x} \\ \cdots \\ \mathbf{A}_{(M)}\mathbf{x} \end{bmatrix} \text{ and } \mathbf{A}^T\mathbf{y} = \begin{bmatrix} \mathbf{A}_1^T\mathbf{y} \\ \mathbf{A}_2^T\mathbf{y} \\ \cdots \\ \mathbf{A}_N^T\mathbf{y} \end{bmatrix},$$

in scenarios a and b, respectively, require $\mathbf{x}$ and $\mathbf{y}$ to be *broadcasted* to (or synchronized across) all the nodes. Then, independent of one another, every node computes $\mathbf{A}_{(i)}\mathbf{x}$ and $\mathbf{A}_j^T\mathbf{y}$.

On the other hand, computing

$$\mathbf{A}^T\mathbf{y} = \sum_{i=1}^{M} \mathbf{A}_{(i)}^T\mathbf{y}_i \text{ and } \mathbf{Ax} = \sum_{i=1}^{N} \mathbf{A}_j\mathbf{x}_j,$$

in scenarios a and b, respectively, take two steps each: assuming every node keeps its corresponding $\mathbf{x}_i$ and $\mathbf{y}_j$, each $\mathbf{A}_{(i)}^T\mathbf{y}_i$ and $\mathbf{A}_j\mathbf{x}_j$ are computed independently in parallel; then the summation is put together through a *reduce* operation across all the nodes.

In summary, in scenario a, computing $\mathbf{Ax}$ is divided into the parallel jobs of computing $\mathbf{A}_{(i)}\mathbf{x}$, which are preceded by *broadcasting* $\mathbf{x}$ whereas computing $\mathbf{A}^T\mathbf{y}$ is done by parallelly computing $\mathbf{A}_{(i)}^T\mathbf{y}_i$. followed by a *reduce* operation. Scenario b has the exact opposite.

In scenario c, computing either $\mathbf{Ax}$ or $\mathbf{A}^T\mathbf{y}$ requires a mixed use of both broadcasting and reduce operations. For example, it takes three steps to compute

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{A}_{(1)}\mathbf{x} \\ \mathbf{A}_{(2)}\mathbf{x} \\ \cdots \\ \mathbf{A}_{(M)}\mathbf{x} \end{bmatrix} = \sum_{j=1}^{N} \begin{bmatrix} \mathbf{A}_{1,j}\mathbf{x}_j \\ \mathbf{A}_{2,j}\mathbf{x}_j \\ \cdots \\ \mathbf{A}_{M,j}\mathbf{x}_j \end{bmatrix}$$

In step 1, for every $j = 1, 2, \ldots, N$ in parallel, $\mathbf{x}_j$ is *broadcasted* to nodes $(1, j), (2, j), \ldots, (M, j)$; in step 2, $\mathbf{A}_{i,j}\mathbf{x}_j$ for all i, j are computed in parallel; finally, for every $i = 1, 2, \ldots, M$ in parallel, a *reduce* operation is applied to nodes $(i, 1), (i, 2), \ldots, (i, N)$ to return $\mathbf{A}_{(i)}\mathbf{x} = \sum_{j=1}^{N} \mathbf{A}_{i,j}\mathbf{x}_j$ to all these nodes for further use. A similar process can compute $\mathbf{A}^T\mathbf{y}$.

In all scenarios, the blocks of $\mathbf{A}$ can have different sizes (although, in scenario c, the blocks should be aligned; otherwise, extra broadcasting and reduce operations will be incurred). In fact, assigning larger blocks to faster (or less busy) nodes will improve the overall efficiency.

III. Approach I: distributed implementations of prox-linear algorithms

When $R(\mathbf{x})$ and $L(\mathbf{Ax}, \mathbf{b})$ are block separable, it is straightforward to develop distributed implementations of prox-linear algorithms such as ISTA, FPC, and FISTA. In short, prox-linear algorithms are the iterations of gradient descent and proximal operations. The former requires $\nabla_x L(\mathbf{Ax}^k, \mathbf{b}) = \mathbf{A}^T\nabla L(\mathbf{Ax}^k, \mathbf{b})$, which can be obtained based on the distributed computing of $\mathbf{Ax}$ and $\mathbf{A}^T\mathbf{y}$ together with the separability of $L(\mathbf{Ax}, \mathbf{b})$. The latter is often straightforward and can take advantage of the separability of $R(\mathbf{x})$.

Specifically, the basic prox-linear iteration applied to (1) is

$$\mathbf{x}^{k+1} \leftarrow \arg\min_{\mathbf{x}} \lambda \cdot R(\mathbf{x}) + \langle \mathbf{x}, \mathbf{A}^T\nabla L(\mathbf{Ax}^k, \mathbf{b})\rangle + \frac{1}{2\delta_k}\|\mathbf{x} - \mathbf{x}^k\|_2^2,$$

which is given by

$$\mathbf{x}^{k+1} = \mathbf{prox}_{\lambda R}(\mathbf{x}^k - \delta_k\mathbf{A}^T\nabla L(\mathbf{Ax}^k, \mathbf{b})), \qquad (2)$$

where δ_k is the step size and the proximal operator $\mathbf{prox}_{\lambda R}(\mathbf{t})$ is given by $\mathbf{prox}_{\lambda R}(\mathbf{t}) = \arg\min_{\mathbf{x}} \lambda \cdot R(\mathbf{x}) + \frac{1}{2}\|\mathbf{x} - \mathbf{t}\|_2^2$.

In scenario a, every node i keeps $\mathbf{A}_{(i)}$, $\mathbf{b}_i$, and current $\mathbf{x}^k$ entirely. It computes $\mathbf{A}_{(i)}\mathbf{x}^k$ and then $\nabla L_i(\mathbf{A}_{(i)}\mathbf{x}^k, \mathbf{b}_i)$. Once such computation is finished on all nodes, a *reduce* operation computes $\mathbf{A}^T\nabla L(\mathbf{Ax}^k, \mathbf{b}) = \sum_{i=1}^{M} \mathbf{A}_{(i)}^T\nabla L_i(\mathbf{A}_{(i)}\mathbf{x}^k; \mathbf{b}_i)$ and returns the sum to every node. Every node then independently finishes (2), and all obtain the identical $\mathbf{x}^{k+1}$.

In scenario b, every node j keeps $\mathbf{A}_j$, entire $\mathbf{b}$, but just $\mathbf{x}_j^k$ out of current $\mathbf{x}^k$. Every node computes $\mathbf{A}_j\mathbf{x}_j^k$ and awaits the *reduce* operation to return $\mathbf{Ax}^k = \sum_{j=1}^{N} \mathbf{A}_j\mathbf{x}_j^k$. Then, every node j independently computes $\mathbf{A}_j^T\nabla L(\mathbf{Ax}^k, \mathbf{b})$ and further, based on the separability of R, obtains $\mathbf{x}_j^{k+1} = \mathbf{prox}_{\lambda R_j}(\mathbf{x}_j^k - \delta_k\mathbf{A}_j^T\nabla L(\mathbf{Ax}^k, \mathbf{b}))$.

Scenarios a and b require the separability of $L(\mathbf{Ax}, \mathbf{b})$ and $R(\mathbf{x})$, respectively, not both at the same time. Scenario c will

Algorithm 1 P-FISTA: **scenario-b** distributed LASSO

1: node j keeps $\mathbf{A}_j$, $\mathbf{b}$, initializes $\mathbf{x}_j^0 = \mathbf{x}_j^1 = 0$;
2: **for** $k = 1, 2, \ldots, K$ **do**
3: $\quad \bar{\mathbf{x}}_j \leftarrow \mathbf{x}_j^k + \frac{k-2}{k+1}\left(\mathbf{x}_j^k - \mathbf{x}_j^{k-1}\right)$;
4: $\quad \mathbf{w} \leftarrow \sum_{j=1}^N \mathbf{A}_j \bar{\mathbf{x}}_j$ by MPI_Allreduce;
5: $\quad \mathbf{y} \leftarrow \nabla L(\mathbf{w}; \mathbf{b})$;
6: $\quad \mathbf{g}_j \leftarrow \mathbf{A}_j^T \mathbf{y}$;
7: $\quad \mathbf{x}_j^{k+1} \leftarrow \mathbf{prox}_{\lambda\|\cdot\|_1}(\bar{\mathbf{x}}_j - \delta_k \mathbf{g}_j)$;
8: **end for**

require both at the same time, and we leave the details to the reader.

If δ_k is fixed, it is often set inversely proportional to the Lipschitz constant of $\nabla_x L(\mathbf{Ax}, \mathbf{b})$, which can be estimated by a distributed implementation of the power method based on operations $\mathbf{Ax}$ and $\mathbf{A}^T\mathbf{y}$. Some algorithms determine δ_k by the Barzilai-Borwein method followed by back-track line search, both of which require extra computation but nonetheless boil down to inner products involving $\nabla L(\mathbf{Ax}^k, \mathbf{b})$.

A. Example: distributed LASSO

The LASSO model is

$$\underset{\mathbf{x}}{\text{minimize}}\, \lambda\|\mathbf{x}\|_1 + \frac{1}{2}\|\mathbf{Ax} - \mathbf{b}\|_2^2.$$

The ℓ_1-proximal operator $\mathbf{prox}_{\lambda\|\cdot\|_1}(\mathbf{t})$ is known as shrinkage or soft-thresholding, which is component-wise separable and can be computed in closed form as $\text{shrink}(x_j, \lambda) = \text{sign}(x_j)\max(|x_j|, 0)$. Algorithm 1 presents our distributed implementation for scenario b, which is based on FISTA [8], a Nesterov-type acceleration of (2), and the SPMD (single program, multiple data) technique. When Algorithm 1 runs on distributed nodes, steps 3, 5, 6 and 7 are executed independently on every node, and the *MPI_Allreduce* operation in step 4 forms the sum and sends it to all the nodes.

B. Example: distributed sparse logistic regression

Sparse logistic regression solves

$$\min_{\mathbf{w},c}\ \lambda\|\mathbf{w}\|_1 + \frac{1}{m}\sum_{i=1}^m \log\left(1 + \exp(-b_i(\mathbf{w}^T\mathbf{a}_i + c))\right). \quad (3)$$

For given data $\mathbf{a}_i$ and b_i. Form matrix

$$[\mathbf{C}\ \mathbf{b}] = \begin{bmatrix} b_1\mathbf{a}_1^T & b_1 \\ b_2\mathbf{a}_2^T & b_2 \\ \cdots & \\ b_m\mathbf{a}_m^T & b_m \end{bmatrix} = \begin{bmatrix} \mathbf{C}_{(1)} & \mathbf{b}_{(1)} \\ \mathbf{C}_{(2)} & \mathbf{b}_{(2)} \\ \cdots & \\ \mathbf{C}_{(M)} & \mathbf{b}_{(M)} \end{bmatrix},$$

where each $\mathbf{C}_{(i)}$ and $b_{(i)}$ are blocks of rows of $\mathbf{C}$ and $\mathbf{b}$, respectively, and $L(\mathbf{t}) = \frac{1}{m}\sum_{i=1}^m \log(1 + \exp(-t_i))$. Then, (3) can be rewritten as

$$\underset{\mathbf{w},c}{\text{minimize}}\ \lambda\|\mathbf{w}\|_1 + L(\mathbf{Cw} + \mathbf{b}c), \quad (4)$$

whose prox-linear iteration is

$$\mathbf{w}^{k+1} = \mathbf{prox}_{\lambda\|\cdot\|_1}(\mathbf{w}^k - \delta_k\mathbf{C}^T\nabla L(\mathbf{Cw}^k + \mathbf{b}c^k)),$$
$$c^{k+1} = c^k - \delta_k\mathbf{b}^T\nabla L(\mathbf{Cw}^k + \mathbf{b}c^k).$$

Algorithm 2 diSLR: **scenario-a** distributed sparse logistic reg.

1: node i keeps $\mathbf{C}_{(i)}$, $\mathbf{b}_{(i)}$, sets $\mathbf{w}^0 = \mathbf{w}^1 = 0$, $c^0 = c^1 = 0$;
2: **for** $k = 1, 2, \ldots, K$ **do**
3: $\quad \bar{\mathbf{w}} \leftarrow \mathbf{w}^k + \frac{k-2}{k+1}\left(\mathbf{w}^k - \mathbf{w}^{k-1}\right)$;
4: $\quad \bar{c} \leftarrow c^k + \frac{k-2}{k+1}\left(c^k - c^{k-1}\right)$;
5: $\quad \mathbf{y}_{(i)} \leftarrow \nabla L_i(\mathbf{C}_{(i)}\bar{\mathbf{w}} + \mathbf{b}_{(i)}\bar{c})$;
6: $\quad \mathbf{g}_\mathbf{w} \leftarrow \sum_{i=1}^M \mathbf{C}_{(i)}^T\mathbf{y}_{(i)}$ by MPI_Allreduce;
7: $\quad g_c \leftarrow \sum_{i=1}^M \mathbf{b}_{(i)}^T\mathbf{y}_{(i)}$ by MPI_Allreduce;
8: $\quad \mathbf{w}^{k+1} \leftarrow \mathbf{prox}_{\lambda\|\cdot\|_1}(\bar{\mathbf{w}} - \delta_k\mathbf{g}_\mathbf{w})$;
9: $\quad c^{k+1} \leftarrow \bar{c} - \delta_k g_c$;
10: **end for**

Algorithm 2 describes our implementation for scenario a. Steps 3–5, 8, and 9 run independently on every node, and steps 6 and 7 call MPI_Allreduce to form the sums and send them to all the nodes.

IV. Approach II: Parallel Greedy Coordinate Descent Method

The coordinate descent method (CD) is one of the first schemes applied to nonlinear optimization. At each iteration of this method, the objective $F(\mathbf{x})$ is minimized with respect to a chosen coordinate (or block of coordinates) while the others are fixed. Its main advantage is the simplicity of each update.

It is important to decide which coordinate(s) to update at each iteration. The most common rule cycles through all of the coordinates in a Gauss-Seidel fashion; we call it *Cyclic CD*. In *Random CD*, the coordinate(s) are selected at random, and its analysis is based on the expected objective value. *Greedy CD*, on the other hand, chooses the coordinate(s) with the best merit value(s), for example, a coordinate of the gradient vector with the maximal absolute value. There are also rules that mix the above three, which we call *Mixed CD*. For example, ref. [12] partitions the coordinates into B blocks and randomly selects P blocks, within each of which a coordinate is selected for update in a greedy manner.

We argue that an often-have property of sparse optimization favors Greedy CD than other rules. In fact, under certain orthogonality conditions (such as the RIP and incoherence conditions), certain greedy rules are guaranteed to select the coordinates corresponding to nonzero entries in the final solution. Below we present GRock, a greedy block coordinate descent method for solving (1) that is friendly with parallel computing. To this end, we assume $R(\mathbf{x})$ is separable and $R(\mathbf{x}) = \sum_{i=1}^n r(\mathbf{x}_i)$, $L(\mathbf{Ax}, \mathbf{b})$ is convex, and $\mathbf{A}$ has columns with unit 2-norm (to simplify our analysis). We let $\beta > 0$ be such that

$$L(\mathbf{A}(\mathbf{x} + \mathbf{d}), \mathbf{b}) \le L(\mathbf{Ax}, \mathbf{b}) + \mathbf{g}^T\mathbf{d} + \frac{\beta}{2}\mathbf{d}^T\mathbf{A}^T\mathbf{Ad}, \quad (5)$$

where $\mathbf{g} = \mathbf{A}^T\nabla L(\mathbf{Ax}, \mathbf{b})$. We have $\beta = 1$ for the square loss function and $\beta = \frac{1}{4}$ for the logistic loss. We define the potential of each coordinate i by

$$d_i = \arg\min_d\ \lambda \cdot r(x_i + d) + g_i d + \frac{\beta}{2}d^2, \quad (6)$$

which is given in closed form as $d_i = \mathbf{prox}_{\frac{\lambda}{\beta}r}(x_i - \frac{1}{\beta}g_i) - x_i$, where g_i is the ith entry of $\mathbf{g}$. Let $\mathbf{d} = [d_1; d_2; \ldots; d_n]$.

As we will later consider scenario b and use multiple computing nodes, we divide the coordinates into N blocks. For each block j, let

$$m_j = \max\{|d| : d \text{ is an element of } \mathbf{d}_j\} \tag{7}$$

and s_j be such that $m_j = d_{s_j}$, i.e., coordinate s_j achieves the maximum. At each iteration, out of the N blocks, the set $\mathcal{P}$ of the P blocks with largest m_j are selected, and their maximizing entries s_j, for $j \in \mathcal{P}$, are updated. In short, the best coordinate of each of the best P blocks are updated. The algorithm GRock is summarized in Algorithm 3.

Algorithm 3 GRock for scenario b

1: Initialize $\mathbf{x} = 0 \in \mathbb{R}^n$
2: **while** not converged **do**
3: $\quad \mathbf{d}_j \leftarrow$ (6) for each block j;
4: $\quad m_j, s_j \leftarrow$ (7) for each block j;
5: $\quad \mathcal{P} \leftarrow$ the indices of the P blocks with largest m_j;
6: $\quad x_{s_j} \leftarrow x_{s_j} + d_{s_j}$, for each $j \in \mathcal{P}$.
7: **end while**

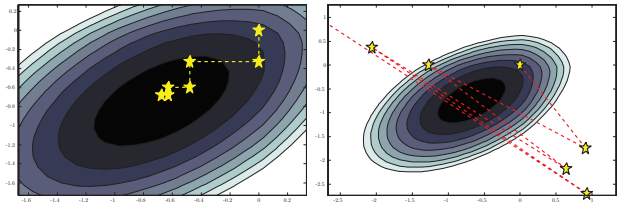


Fig. 2: demonstration for GRock

Without additional conditions, GRock is guaranteed to converge only for $P = 1$. In Figure 2, the left plot demonstrates $P = 1$, in which case the objective descents at each iteration and convergence is ensured. The right plot demonstrates $P = 3$ (we project the contour to the first two coordinates), in which case the objective increases and the points diverge over the iterations. Due to the slanted shape of the contour, if two components are updated together, the updates must work together in order to decrease the objective, whereas each update (6) ignores the other and does not have the big picture. If the contours are (nearly) aligned with the coordinates, then $P = 3$ would work fine and take few iterations than $P = 1$.

To ensure convergence, similar to the analysis in [12], we define a block spectral radius as

$$\rho_P = \max_{\mathbf{M} \in \mathcal{M}} \rho(\mathbf{M}), \tag{8}$$

where $\mathcal{M}$ is the set of all $P \times P$ submatrices that we can obtain from $\mathbf{A}^T\mathbf{A}$ corresponding to selecting exactly one column from each of the P blocks and $\rho(\mathbf{M})$ is the spectral radius of $\mathbf{M}$. Apparently, ρ_P is monotonically increasing with respect to P, thus $1 \le \rho_P \le \rho(\mathbf{A}^T\mathbf{A})$. The intuition for ρ_P is that if the columns of $\mathbf{A}$ from different blocks are nearly orthogonal to one another then $\mathbf{M} \in \mathcal{M}$ will be close to identity matrix and hence ρ_P is small. In fact, Lemma IV.1 shows that $\rho_P < 2$ induces monotonically decreasing of the objective function values.

Lemma IV.1. *Assume $R(\mathbf{x})$ is convex, $L(\mathbf{Ax}, \mathbf{b})$ satisfies assumption (5), and $\mathbf{x}^k$ be the sequence generated by Alg. 3. If $\rho_P < 2$, then*

$$\mathcal{F}(\mathbf{x}^{k+1}) - \mathcal{F}(\mathbf{x}^k) \le \frac{\rho_P - 2}{2}\beta\|\mathbf{x}^{k+1} - \mathbf{x}^k\|^2.$$

The proof of this lemma and the theorem below is included in the supplementary material. Based on Lemma IV.1, we can show the following convergence result.

Theorem IV.2. *Let $R(\mathbf{x}) = \|\mathbf{x}\|_1$, and $\mathbf{x}^*$ be a solution to (1), and $\mathbf{x}^k$ be the sequence generated by Alg. 3. Assume $F(\mathbf{x})$ satisfies the assumptions in Lemma IV.1 and $\rho_P < 2$, then we have*

$$F(\mathbf{x}^k) - F(\mathbf{x}^*) \le \frac{2\left(C(2\rho_n + \sqrt{\frac{n}{P}}) + 2\frac{\lambda}{\beta}\frac{n-P}{\sqrt{P}}\right)^2}{(2-\rho_P)} \cdot \frac{1}{k}.$$

Note that as long as any two columns from two different blocks are linearly independent, we automatically have $\rho_2 < 2$. In another word, we can at least let $P = 2$ and ensure convergence.

A. Greed rocks for sparse optimization

In general optimization, CD methods update only a few coordinates each iteration and thus take more iterations than gradient or prox-linear methods. However, in sparse optimization, it is the opposite with Greedy CD since its coordinate selections often occur on those corresponding to nonzero entries in the solution. As such selection will keep most entries in $\mathbf{x}$ as zero throughout the iterations, the problem dimension is effectively reduced. On the other hand, prox-linear iterations typically have dense intermediate solutions at the beginning, and they can take fairly many iterations before the intermediate solutions become finally sparse.

To illustrate this, we compare GRock to the other three selection schemes such as Cyclic CD, Greedy CD [13] and Mixed CD (see [12] for details) applied to the LASSO problem. The convergence results of the different coordinate descent methods are shown in Fig. 3. For Mixed CD and GRock, we partition the matrix $\mathbf{A} \in \mathbb{R}^{512\times1024}$ by column into 64 blocks, and we select 8 blocks to update in each iteration. Fig. 3 shows that greedy selection is effective for the LASSO problem. In particular, Fig. 3 highlights the fastest convergence of our GRock algorithm for this example.

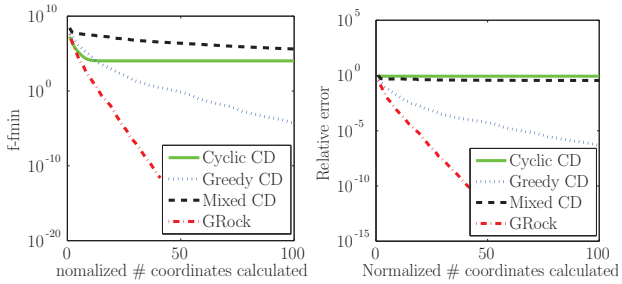


Fig. 3: a comparison of different coordinate descent methods

In addition, we test the GRock algorithm on the $mug025\_12\_12.mat$ dataset which is created by the authors in [10]. This data matrix $\mathbf{A}$ has the dimension of $[6205, 24820]$ with about 0.4% nonzero entries. Convergence results are shown in Fig. 4. The "optimal" solution $\mathbf{x}^*$ is calculated by FISTA with 400 iterations. The algorithm is terminated after 100 iterations or $\frac{\|\mathbf{x}^k-\mathbf{x}^*\|}{\|\mathbf{x}^*\|} \leq 10^{-11}$.

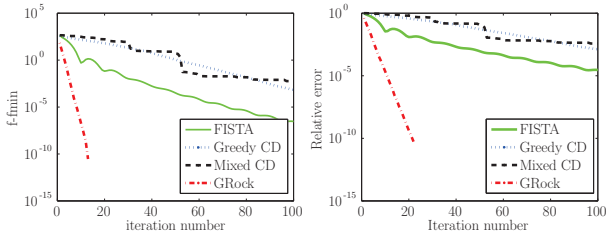


Fig. 4: a comparison of different coordinate descent methods and FISTA

B. Parallel GRock

GRock in Algorithm 3 can be easily parallelized as its steps 3 and 4 can be done in parallel over all the N blocks, step 5 realized by MPI, and step 6 parallelized over the P selected blocks. One of the major computation in GRock is $\mathbf{g} = \mathbf{A}^T\nabla L(\mathbf{A}\mathbf{x}, \mathbf{b})$, which is needed by (6) and thus step 2. Note that as only P coordinates of $\mathbf{x}$ are updated in each iteration, $\mathbf{A}\mathbf{x}$ can be cheaply updated instead of recomputed. Furthermore, in scenario b, similar to Approach I, the computation involving $\mathbf{A}^T$ is distributed to N computing nodes. Hence, the per-iteration complexity of GRock is lower than that of Approach I though still at the same order.

When P is not given *a priori*, one can start with an aggressive (i.e., large) value and dynamically reduce it whenever the objective increases. Note that computing the objective in Approaches I and II involves rather minor extra computation and communication.

V. Complexity Analysis

We will compare the complexity of the three algorithms including parallel prox-linear method, parallel ADMM and GRock in terms of wall clock time. The time includes the computation part and communication part. The total time depends on the number of iterations taken.

Each iteration of distributed prox-linear algorithms takes $O(mn/N)$, dominated by the two matrix-vector multiplications, and $O(n \log N)$ on communication for calling MPI_Allreduce (assume butterfly communication [14]) to get $\sum_{i=1}^N \mathbf{A}_i\mathbf{x}_i$.

Each iteration of GRock takes $O(mn/N+Pm)$ on computing, breaking down to $O(mn/N)$ for one matrix-vector multiplication and $O(Pm)$ for updating the residual $\mathbf{A}\mathbf{x}-\mathbf{b}$ as only P coordinates of $\mathbf{x}$ are updated, and $O(n \log N + N \log N)$ on communication for calling MPI_Allreduce to get $\sum_{i=1}^N \mathbf{A}_i\mathbf{x}_i$ and the best P blocks.

Each iteration of distributed ADMM, which we will numerically compare below, takes $O(mn/N + 2m^2)$ on computing and $O(mN \log N)$ on communication for calling MPI_Allreduce. In addition, there is a computational cost of $O(nm^2/N + m^3)$ or $O(mn^2/N^2 + n^3/N^3)$ for certain matrix factorization during initialization (see section 4 of [9] for details).

The per-iteration costs of all algorithms are comparable.

VI. Numerical Results

Our tests use both a cluster at Rice University and Amazon EC2. The cluster at Rice consists of 170 Appro Greenblade E5530 nodes each with two quad-core 2.4GHz Xeon (Nahalem) CPUs. Each node has 12GB of memory shared by all cores on the node. The number of processes used is equal to that of the cores.

A. Which algorithm is more scalable?

We compare three distributed algorithms on LASSO. They include parallel ADMM applied to the Lagrange dual of LASSO (PD-ADMM), P-FISTA, and GRock. Two instances of LASSO in the test are described in table I. Observation vector

TABLE I: datasets tested on Rice cluster

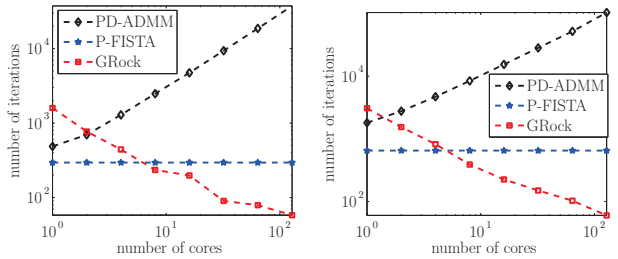
	$\mathbf{A}$ type	$\mathbf{A}$ size	λ	sparsity of $\mathbf{x}^*$
dataset I	Gaussian	1024×2048	0.1	100
dataset II	Gaussian	2048×4096	0.01	200

$\mathbf{b}$ is obtained by the method proposed in [8] such that the given sparse vector $\mathbf{x}^*$ is the optimal solution to the LASSO problem. The stopping criterion is set to $\frac{\|\mathbf{x}^k-\mathbf{x}^*\|}{\|\mathbf{x}^*\|} \leq 10^{-11}$. In P-FISTA, stepsize δ_k is set to $\frac{1}{\|\mathbf{A}\|_2^2}$. In GRock, we set $P = N$, the number of data blocks. The test tries $N = 1, 2, 4, 8, 16, 32, 64, 128$.

Fig. 5(a) and 5(b) show total number of iterations vs number of cores. As expected, the numbers of iterations of P-FISTA remain constant. PD-ADMM has its number of iterations linearly increasing with the number of processes, which is caused by the variable duplications in the ADMM formulation (see section 8 in [9]). On the other hand, GRock takes fewer iterations as increasing $N = P$ means more coordinates being updated at each iteration.

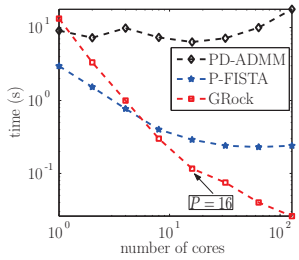
Fig. 5(c) and 5(d) show total time vs number of cores. PD-ADMM has its total computation time staying roughly constant due to a combined effect of more iterations and cheaper per-iteration cost. P-FISTA has its total time reducing linearly until the communication time begins to signify. GRock reduces its total time faster than P-FISTA, as a result of both cheaper per-iteration cost and fewer iterations. The results match the percentages of the communication part of total time, both of which are measured on a specific core, given in Fig. 5(e) and 5(f). The percentage is overall linearly correlated to the number of cores. Note that the matrix-factorization time during the initialization of PD-ADMM is counted in the total time.

Note that we do not have to choose $P = N$ for GRock as in this experiment. For some problems, we may need to choose $P < N$ to ensure the convergence of GRock.

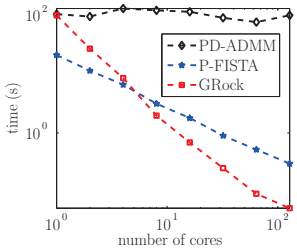


(a) Dataset I: cores vs iterations

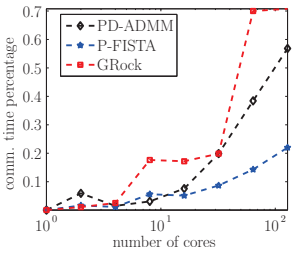
(b) Dataset II: cores vs iterations



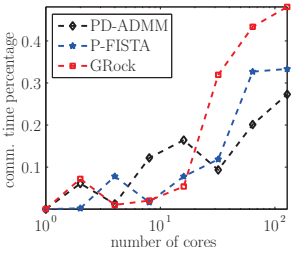
(c) Dataset I: cores vs time



(d) Dataset II: cores vs time



(e) Dataset I: cores vs comm. percentage



(f) Dataset II: cores vs comm. percentage

Fig. 5: LASSO on two datasets.

B. Large datasets

In this LASSO test, $\mathbf{A}$ is generated by randn$(100000, 200000)$ with 20 billion nonzero entries and having a size of 170GB. The solution $\mathbf{x}^*$ has 4000 nonzero entries, each sampled from $\mathcal{N}(0,1)$ independently. We set the maximum number of iterations to 2500 and the stopping criterion as $\frac{\|\mathbf{x}^k-\mathbf{x}^*\|}{\|\mathbf{x}^*\|} \leq 10^{-5}$. On Amazon EC2, we requested 20 high-memory quadruple extra-large instances giving us a total of 160 cores and 1.2TB of memory in total. Table II compares the performance of PD-ADMM P-FISTA and GRock. Note that the performance of PD-ADMM depends on a penalty parameter. We pick it as the best out of only a few trials as we cannot afford more trials.

TABLE II: large dataset time results

	PD-ADMM	P-FISTA	GRock
estimate stepsize (min.)	n/a	1.6	n/a
matrix factorization (min.)	51	n/a	n/a
iteration time (min.)	105	40	1.7
number of iterations	2500	2500	104
communication time	30.7	9.5	0.5
stopping relative error	1E-1	1E-3	1E-5
total time (min)	156	41.6	1.7
cost	**$85**	**$22.6**	**$0.93**

VII. Conclusion

In this paper, we have proposed two approaches including distributed implementation of prox-linear algorithms and GRock for solving large-scale sparse optimization problems. Our approaches are motivated by the two typical structures of sparse optimization problems, namely, separable objective functions and rough orthogonality in the data. Numerical results show that both approaches are more scalable than the popular distributed ADMM method, which is nonetheless more general and has applications beyond sparse optimization.

Acknowledgments

The work is supported by NSF grants DMS-0748839, and ECCS-1028790, and ARO/ARL MURI grant FA9550-10-1-0567. The authors thank Hui Zhang for his helpful discussions.

References

[1] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.

[2] J. Bi, K. Bennett, M. Embrechts, C. Breneman, and M. Song, "Dimensionality reduction via sparse support vector machines," *The Journal of Machine Learning Research*, vol. 3, pp. 1229–1243, 2003.

[3] H. Zou, T. Hastie, and R. Tibshirani, "Sparse principal component analysis," *Journal of computational and graphical statistics*, vol. 15, no. 2, pp. 265–286, 2006.

[4] L. I. Rudin, S. Osher, and E. Fatemi, "Nonlinear total variation based noise removal algorithms," *Physica D: Nonlinear Phenomena*, vol. 60, no. 1, pp. 259–268, 1992.

[5] E. J. Candès, "Compressive sampling," in *Proceedings oh the International Congress of Mathematicians: Madrid, August 22-30, 2006: invited lectures*, 2006, pp. 1433–1452.

[6] E. J. Candès, J. Romberg, and T. Tao, "Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information," *Information Theory, IEEE Transactions on*, vol. 52, no. 2, pp. 489–509, 2006.

[7] E. T. Hale, W. Yin, and Y. Zhang, "Fixed-point continuation for ℓ_1-minimization: Methodology and convergence," *SIAM Journal on Optimization*, vol. 19, no. 3, pp. 1107–1130, 2008.

[8] A. Beck and M. Teboulle, "A fast iterative shrinkage-thresholding algorithm for linear inverse problems," *SIAM Journal on Imaging Sciences*, vol. 2, no. 1, pp. 183–202, 2009.

[9] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011.

[10] J. K. Bradley, A. Kyrola, D. Bickson, C. Guestrin, and C. Guestrin, "Parallel coordinate descent for ℓ_1-regularized loss minimization." in *ICML*, 2011, pp. 321–328.

[11] C. Scherrer, M. Halappanavar, A. Tewari, and D. J. Haglin, "Scaling up coordinate descent algorithms for large ℓ_1 regularization problems," Pacific Northwest National Laboratory (PNNL), Richland, WA (US), Tech. Rep., 2012.

[12] C. Scherrer, A. Tewari, M. Halappanavar, and D. Haglin, "Feature clustering for accelerating parallel coordinate descent," in *NIPS*, P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., 2012, pp. 28–36.

[13] Y. Li and S. Osher, "Coordinate descent optimization for ℓ_1 minimization with application to compressed sensing; a greedy algorithm," *Inverse Problems and Imaging*, vol. 3, no. 3, pp. 487–503, 2009.

[14] P. Pacheco, *An introduction to parallel programming*. Morgan Kaufmann, 2011.