Convergence Rate Analysis of Parallel Block Coordinate Descent Method for Composite Function Minimization

College of Mathematics

Student:Yutong Dai Adviser:Yang Weng

[**Abstract**] In this paper, we propose a synchronous parallel block coordinate descent algorithm for minimizing a composite function, which consists of a smooth convex function plus a nonsmooth but separable convex function. Our method can include some existing synchronous parallel algorithms as special cases. For the sake of using our method to solve high dimension problems, we further develop a randomized variant. We have proved that both proposed parallel algorithms are endowed with sub-linear convergence rate for general convex composite function. The experiments on solving the regularized logistic regression with ℓ_1 norm problem show that the proposed parallel algorithms are quite efficient to implement.

[**Key Words**] Optimization; Block Coordinate Descent; Synchronous Parallel Computation; Logistic Regression with ℓ_1 regularization

目录

1 Introduction

Consider the problem of minimizing a nonsmooth convex composite function $f(x)$ of the form,

$$\begin{aligned} &\min f(x) = g(x) + h(x) \triangleq g(x_1, ..., x_K) + \lambda \sum_{i=1}^{K} h_i(x_i) \\ &\text{subject to } x_i \in X_i \end{aligned} \tag{1.1}$$

In (1.1), $(x_1, ..., x_K)$ denotes a pre-defined partition of the optimization variable x, where $x_i \in X_i \subset \mathbb{R}^{n_i}$ and $\sum_{i=1}^{K} n_i = n$; $X = X_1 \times ... \times X_K$ is the feasible set for x. $g(x)$ is continuously differentiable and block-wise convex, i.e., $g(x_1, ..., x_K)$ is convex with respect to $x_i, i = 1, ..., K$ (once $g(x)$ is convex with respect to $x \in X$, $g(x)$ is block-wise convex.); $h(x)$ is convex and separable, not necessarily differentiable.

The Problem(1.1) indeed covers a variety of application problems in the many areas, to name a few, machine learning and statistics, e.g. [13, 19, 15], and has been intensively studied. A family of well-known algorithms are the Block Coordinate Descent(BCD) and its variants([21, 7, 18, 22] and references therein). They are iterative methods and at each iteration new iterate is derived from either approximately or exactly minimizing the chosen variables while keeping the rest fixed. Assumptions made on $g(x), h(x)$ and different block-selection rules give rise to different convergence results. We briefly summarize related works, to our best knowledge, as follows.

1.1 Related Work

There are two mainstreams for solving Problem(1.1). One is to do exact minimization at each iteration while the other is to do approximate minimization and both of them prefer to employ the Gauss-Seidel update strategy. To be specific, when updating each block, the Gauss-Seidel style fixes the rest blocks to be the most up- to-date solution. Also, there's a less frequently employed Jacobi update strategy. When updating each coordinate, the Jacobi style fixes the rest blocks to be the solution obtained from previous cycle. As for block selection rules, cyclic rules, maximum block improvement rule and random rules are commonly used[7, 21].

For exact minimization, the local linear convergence rate was established under some assumptions by Luo and Tseng[9]. Tseng[20] proved the convergence for nondifferentiable functions under certain conditions. However, doing exact minimization at each iteration may be theoretically intractable or numerically infeasible, so much more recent works focus on inexact minimization. When $h(x) := 0$, (block) coordinate descent combined with different step-size selection rules is commonly used. For example, global linear(sub-linear) convergence rate for strongly (general) convex optimization was established by Beck and Tetruashvili[2]. When the nondifferentiable part $h(x)$ exists, one popular approximation method is to use a $\tilde{g}(x)$ to upper bound the smooth part $g(x)$ while keep the nonsmooth convex term $h(x)$. Minimizing $\tilde{g}(x) + h(x)$ is often

referred as proximal (block) coordinate descent in literature (e.g. [16]). More general and unified treatments for inexact minimization can be found in [7, 21]. Hong et al. suggested minimizing upper bound function at each iteration for chosen blocks and sub-linear convergence rate was established for nonsmooth separable convex function. Tseng and Yun instead used quadratic approximation function combined with Armijo rule and established linear convergence rate for nonsmooth separable convex function[21]. Usually, under some mild assumptions and conditions, combined with either deterministic or random block selection rules, sub-linear and linear convergence rate was established for general convex functions and strongly convex functions respectively in inexact minimization strategy.

Many numerical experiments in above mentioned works reveal that BCD-type algorithms enjoy satisfactory performances in practice, for example, [5, 11, 22, 17]. The underlying reasons for popularity of BCD-type algorithms includes following reasons. On the one hand, it breaks high dimension optimization problems into lower ones and first order information is used and thus the cost for both computation and storage at each iteration can be significantly reduced. On the other hand, it scales well with problem dimension, for example, in [18], Shai Shalev-Shwartz and Ambuj Tewari have proved that the required iteration to achieve (expected) ϵ accuracy grows linearly with the number of variables as well as the number of training samples.

1.2 Motivation

Current existing sequential BCD-type algorithms, which optimize one block on one processor at one time, may not fulfill our desire to get faster since Nesterov[12] has proved that you can get at most the $O(\frac{1}{k^2})$ convergence rate when only using gradients and function evaluations. Meanwhile, in [11], Nesterov claimed that, when the number of variables is huge, his proposed accelerated method may not be that efficient to implement. At the same time, with the significant progress made in collecting and storing data, there are imperious demands for scalable optimization algorithms. Many efficient algorithms, e.g. [21], scale poorly with the growth of the problem's dimension.

Though a better convergece rate for BCD-type algorithms can not be obtained, thankfully, the availability of high performance multi-core computing platforms seems to be our last hope to be faster. In addition, there have been tremendous improvements in software, which help researchers to utilize computers with multicores more efficiently. With the accesses of abundant computation resources and powerful software, we can optimize multi-blocks simutaneously to save the runtime. Therefore, we turn to work on parallel versions of BCD-type algorithms.

As mentioned by Wright([22]), there are two commonly employed parallel strategies. One is synchronous parallelism([3, 15, 17]) and the other is asynchronous parallelism([8] and references therein). Here, we only focus on the synchronous parallelism.

1.3 Contributions

Motivated by [7, 1, 4], we propose a Parallel Successive Upper approximation Minimization (PSUM) algorithm and its randomized variant(RPSUM). The contributions of this work are as follows:

1. Most sequential BCD-type algorithm combine Gauss-Seidel update strategy with either deterministic or random block selection rule, e.g.[7], few convergence analysis regarding Jacobi update exists. However, to parallelize BCD-type algorithm in a synchronous setting, one must employ Jacobi update strategy. We have proved $O(\frac{1}{k})$ convergence rate using PSUM to solve convex problems.

2. Unlike existing algorithms in literature, e.g. [3, 17], our algorithms use general approximation function, which treat their algorithms as special cases. Meanwhile, minimizing the approximation function actually can potentially be much easier than directly tackle with the original function. Moreover, in [3], when features are highly correlated, it fails to parallelize. But, our algorithm can also deal with this problem and accomplish full parallelism, that is, we can simultaneously update up to K blocks while the variables are highly correlated.

3. We find the idea behind our propose PSUM algorithm is similar with the algorithm in [15]. But our work is different in three aspects. First, PSUM is a parameter free method, that is, we do not need choose step-size to guarantee the convergence of our algorithm. On the contrary, the algorithm in [15] needs fine tuning step-size . Second, assumptions made on approximation function are less strict than those made in [15]. Third, the proof technique is also different, as we fully exploit the Jacobi update strategy.

4. When confronted with high dimension problem, optimizing with respect all blocks at the same time seems to be impractical, as we usually can't have enough processors. So we devised RPSUM, which simultaneously optimizes randomly chosen p blocks, coinciding with the maximal processors you have. Also, many literatures(e.g. [22]) can account for the reason for choosing blocks at random since they have reported that randomized block selection rule can outperform deterministic ones.

1.4 Outline

The rest of this paper is organized as follows. Section2 describes our proposed PSUM and RPSUM algorithms and necessary assumptions. In Section3, convergence analysis and convergence rate is provided here. In Section4, we implement our algorithms to solve the logistic regression problem with ℓ_1 regularization and also numerical results are reported here. We make extra remarks and draw conclusions in Section5.

2 The PSUM and RPSUM Algorithms

2.1 Preliminaries

For simplicity, we need to introduce some auxiliary variables.

1. $x_{-i} = (x_1, ..., x_{i-1}, x_{i+1}, ..., x_K) \quad i = 1, 2, ..., K$

$$\nabla\big(\sum_{i=1}^{K} g(., x_{-i})\ \big)\big|_x = \nabla g(x) \tag{2.1}$$

$$\nabla\big(\sum_{i=1}^{K} g(x_i, .)\ \big)\big|_x = (K-1)\nabla g(x) \tag{2.2}$$

2. $\xi_i^k \in \partial(\, h_i(.)|_{x_i^k})$, where $\partial(\, h_i(.)|_{x_i^k})$ represents a sub-gradient of $h_i(.)$ at x_i^k

The notation $\nabla\big(\sum_{i=1}^{K} g(., x_{-i})\ \big)\big|_x$ seems to be obscure at glance. If we define a permutation matrix U partitioned as $U = [U_1, ..., U_K]$, where $U_i \in \mathbb{R}^{n \times n_i}$ such that

$$x = \sum_{i=1}^{K} U_i x_i \quad x_i = U_i^T x, \quad i = 1, 2, ..., n.$$

Then $\nabla\big(\sum_{i=1}^{K} g(., x_{-i})\ \big)\big|_x$ is equivalent to $\sum_{i=1}^{K} U_i \nabla_i g(x)$, where $\nabla_i g(x)$ is the partial gradient of g with respect to x_i.

2.2 Main Assumptions

Suppose $f(x)$ is a closed proper convex function in $\mathbb{R}^n$. Let $dom\ f$ denote the effective domain of $f(x)$ and let $int(dom\ f)$ denote the interior of $dom\ f$. We assume $dom\ f \cap X \neq \emptyset$. Furthermore, we make assumptions on $f(x)$ regarding Problem (1.1).

Assumption A

Global minimum of Problem(1.1) can be attained on a set $\mathbb{S}$. There's a finite R such that the level set for $f(.)$ defined by x^0 is bounded, that is,

$$\max_{x \in \mathbb{S}} \max_{x} \{||x - x^*|| : f(x) < f(x^0)\} \leq R$$

Next we assume our chosen approximation function $u_i(, ; x)$ satisfy:

Assumption B

1. $u_i(x_i;x) = g(x), \quad \forall x \in X, \forall i \in \{1,2,...,K\}$;
2. $u_i(v_i;x) \geq g(v_i, x_{-i}), \quad \forall x \in X, x_i \in X_i, \forall i \in \{1,2,...,K\}$;
3. $\nabla u_i(x_i;x) = \nabla_i g(x),, \quad \forall x \in X, \forall i \in \{1,2,...,K\}$;
4. $u_i(.;x)$ has Lipschitz continuous gradient with respect to the second argument, that is, for any $x, y \in X$

$$||\nabla u_i(.;x) - \nabla u_i(.;y)|| \leq G_i||x-y||$$

Define $G_{max} = \max_i G_i$

We follow [7], also without causing any confusion, to call the $u_i(.;x)$'s that satisfy Assumption B as a *upper-approximation* function.

2.3 The PSUM Algorithm

The PSUM algorithm is formally described as follows. We parallelize BCD algorithms using Jacobi update strategy, which is clearly indicated in the Algorithm 1 ,step(6). First, we choose a feasible starting point $x^0 \in X$ and give a constant c, which is important for convergence and we shall see soon the way to set it. Then on each iteration, PSUM updates K blocks simultaneously. Until some conditions are met, the algorithm is terminated. The intuition behind the key update rule is three-folds:

1. Use $u_i(z_i;x)$ to upperly approximate the smooth part $g(x)$.
2. Keep the nonsmooth regularized term $h_i(z_i)$ unchanged.
3. Use a quadratic term $c||z_i - x_i||^2$ to guarantee convergence.

Algorithm 1 PSUM for Problem(1.1)

1: **Initialization**
2: $k \leftarrow 0$
3: Consider $x_i^0 \in X^i$, for all $i = 1,2,...,K$
4: **while** not converged **do**
5: **In parallel on** i**th processor**$(i = 1,2...,K)$
6: $x_i^{k+1} \leftarrow argmin_{z_i \in X_i} u_i(z_i;x^k) + \lambda h_i(z_i) + c||z_i - x_i^k||^2$
7: $k \leftarrow k+1$
8: **end while**

When n is large, but K is relative small, then this algorithm can achieve full parallelism, since you may afford K processors.

2.4 The RPSUM Algorithm

However, when K is sufficiently large, for example, if we employ coordinate descent, then the K equals to the number of variables, which is particularly huge in genome domain. Under these circumstances, we may fail to achieve full parallelism, simply owing to the lack of adequate processors. Therefore, we propose the RPSUM with slight modification in the block selection rule. As the PSUM, we first choose a feasible starting point $x^0 \in X$ and give a constant c. Then on the iteration $t+1$, Algorithm 2 chooses

$p < K$ coordinates independently and uniformly at random from $\{1, 2, ..., K\}$; these form a multi-set S_{t+1}. [1] Blocks indexed by S_{t+1} are updated in parallel while the rest blocks are held fixed. Until some conditions are met, the algorithm is terminated.

Algorithm 2 RPSUM for Problem(1.1)

Initialization
$t \leftarrow 0$
Consider $x_i^0 \in X^i$, for all $i = 1, 2, ..., K$
while not converged **do**
  In parallel on each of p **processors**
    Choose $i \in \{1, 2, ..., K\}$ uniformly at random
    $x_i^{t+1} \leftarrow argmin_{z_i \in X_i} u_i(z_i; x^t) + \lambda h_i(z_i) + c||z_i - x_i^t||^2$
  For those coordinates(js) that are not chosen, $x_j^{t+1} \leftarrow x_j^t$
  $t \leftarrow t + 1$
end while

The RPSUM seems to be less efficient, as in iteration $t + 1$, $K - p$ blocks remain unchanged. But it's a trade-off since we might not be able to utilize K processors when K is incredibly huge.

3 Convergence Analysis for PSUM and RPSUM

We provide detailed convergence analysis for the PSUM in this section and the proof is quite similar for RPSUM, hence only some differences will be explicitly addressed.

3.1 Convergence Analysis for PSUM

Proposition 3.1. Suppose *Assumption B* holds, then for all $x, y, z \in X$, with $x = (x_1, ..., x_K), y = (y_1, ..., y_K), z = (z_1, ..., z_K)$,

$$||\nabla(\sum_{i=1}^{K} u_i(.; x)\ )\big|_z - \nabla(\sum_{i=1}^{K} u_i(.; y)\ )\big|_z|| \leq \sqrt{K} G_{max} ||x - y|| \tag{3.1}$$

[1] For the sake of intelligibility, we use t instead of k to number the iteration.

Proof.

$$\begin{aligned}||\nabla(\sum_{i=1}^{K}u_i(.;x)\ )\big|_z-\nabla(\sum_{i=1}^{K}u_i(.;y)\ )\big|_z||&=||\sum_{i=1}^{K}\left(\nabla u_i(.;x)-\nabla u_i(.;y)\ )\big|_z\right||\\&=||\begin{pmatrix}\nabla u_1(z_1;x)-\nabla u_1(z_1;y)\\ \vdots\\ \nabla u_K(z_K;x)-\nabla u_K(z_K;y)\end{pmatrix}||\\&=\sqrt{\sum_{i=1}^{K}||\nabla u_i(z_i;x)-\nabla u_i(z_i;y)||^2}\\&\leq\sqrt{K}G_{max}||x-y||\end{aligned}\tag{3.2}$$

The last inequality holds due to the Assumption B.4 □

Proposition 3.2. Suppose *Assumption B* holds then the sequence $\{x^k\}$, generated by Algorithm 1, satisfies,

$$(x^{k+1}-x^k)\nabla g(x^{k+1})\leq(\sqrt{K}G_{max}-2c)||x^{k+1}-x^k||^2+\lambda(h(x^k)-h(x^{k+1}))\tag{3.3}$$

Proof. Consider the problem,

$$\tilde{x}^{k+1}\leftarrow\underset{z\in X}{argmin}\sum_{i=1}^{K}\{u_i(z_i;x^k)+\lambda h_i(z_i)+c||z_i-x_i^k||^2\}\tag{3.4}$$

Due to the separable structure of (3.4), we have $\tilde{x}^{k+1}=x^{k+1}$. By the optimality of x^{k+1}, we derive,

$$(x^k-x^{k+1})^T\nabla(\sum_{i=1}^{K}\{u_i(z_i;x^k)+\lambda h_i(z_i)+c||z_i-x_i^k||^2\}\ )\big|_{z=x^{k+1}}\geq 0\tag{3.5}$$

Simple operations on (3.5) gives rise to

$$0\leq(x^k-x^{k+1})^T\left[\nabla(\sum_{i=1}^{K}u_i(z_i;x^k)\ )\big|_{z=x^{k+1}}\right]+\lambda(x^k-x^{k+1})^T\xi^{k+1}-2c||x^{k+1}-x^k||^2\tag{3.6}$$

$$\leq(x^k-x^{k+1})^T\left[\nabla(\sum_{i=1}^{K}u_i(z_i;x^k)\ )\big|_{z=x^{k+1}}\right]+\lambda[h(x^k)-h(x^{k+1})]-2c||x^{k+1}-x^k||^2\tag{3.7}$$

where $\xi^{k+1}\in\partial h(x^{k+1})$. The last inequality holds naturally by the definition of sub-gradient

$$h(x^k) - h(x^{k+1}) \geq (x^k - x^{k+1})^T \xi^{k+1}$$

Adding $(x^{k+1} - x^k)^T \big[\nabla\big(\sum_{i=1}^K u_i(z_i; x^{k+1})\big)\big|_{z=x^{k+1}}\big]$ on both sides of (3.7)

$$(x^{k+1} - x^k)^T \big[\nabla\big(\sum_{i=1}^K u_i(z_i; x^{k+1})\big)\big|_{z=x^{k+1}}\big] \leq -2c||x^{k+1} - x^k||^2 + \lambda[h(x^k) - h(x^{k+1})] - (x^{k+1} - x^k)^T \big[\nabla\big(\sum_{i=1}^K u_i(z_i; x^k)\big)\big|_{z=x^{k+1}} - u_i(z_i; x^{k+1})\big)\big|_{z=x^{k+1}}\big] \tag{3.8}$$

Plugging (3.1) into (3.8) and using Cauchy-Schwarz inequality, we have

$$\begin{aligned}(x^{k+1} - x^k)^T \nabla g(x^{k+1}) &= (x^{k+1} - x^k)^T \big[\nabla\big(\sum_{i=1}^K u_i(z_i; x^{k+1})\big)\big|_{z=x^{k+1}}\big] \\ &\leq (\sqrt{K}G_{max} - 2c)||x^{k+1} - x^k||^2 + \lambda[h(x^k) - h(x^{k+1})] \end{aligned} \tag{3.9}$$

□

Lemma 3.1 (Sufficient Descent)**.** Suppose *Assumption A and B* hold then the sequence $\{x^k\}$, generated by Algorithm 1, satisfies,

$$f(x^{k+1}) \leq f(x^k) - \gamma||x^{k+1} - x^k||^2 \tag{3.10}$$

where $K\gamma = [c - (K-1)(\sqrt{K}G_{max} - 2c)] > 0$ and $c > \frac{K-1}{2K-1}\sqrt{K}G_{max}$.

Proof. By the definition of x_i^{k+1} we derive

$$\sum_{i=1}^K u_i(x_i^{k+1}; x^k) + \lambda h(x^{k+1}) + c||x^{k+1} - x^k||^2 \leq \sum_{i=1}^K u_i(x_i^k; x^k) + \lambda h(x^k) \tag{3.11}$$

By the assumptions of $u_i(.; x)$, we have

* (a) $u_i(v_i; x) \geq g(v_i, x_{-i}) \qquad \forall v_i \in X_i, x \in X, i = 1, 2, ..., K$

* (b) $u_i(x_i; x) = g(x_i, x_{-i}) \qquad \forall x \in X, i = 1, 2, ..., K$

Hence, plugging (a),(b) into (3.11) and rearranging it, we obtain

$$\sum_{i=1}^{K} g(x_i^{k+1}, x_{-i}^k) + \lambda h(x^{k+1}) \leq \sum_{i=1}^{K} g(x_i^k, x_{-i}^k) + K\lambda h(x^k) - c||x^{k+1} - x^k||^2$$
$$= Kg(x^k) + \lambda h(x^k) - c||x^{k+1} - x^k||^2 \tag{3.12}$$

By convexity of $\sum_{i=1}^{K} g(x_i^{k+1}, .)$, we have

$$\sum_{i=1}^{K} g(x_i^{k+1}, x_{-i}^k) \geq \sum_{i=1}^{K} g(x_i^{k+1}, x_{-i}^{k+1}) + (x^k - x^{k+1})^T \nabla \big(\sum_{i=1}^{K} g(x_i^{k+1}, .) \big)\big|_{x^{k+1}}$$
$$= Kg(x^{k+1}) + (K-1)(x^k - x^{k+1})^T \nabla g(x^{k+1}) \tag{3.13}$$

Combing (3.9),(3.12),(3.13) produces

$$Kg(x^{k+1}) + \lambda h(x^{k+1}) \leq Kg(x^k) + \lambda h(x^k) - c||x^{k+1} - x^k||^2 + (K-1)(x^{k+1} - x^k)^T \nabla g(x^{k+1}) \tag{3.14}$$
$$\leq Kg(x^k) + \lambda h(x^k) +$$
$$(K-1)\lambda[h(x^k) - h(x^{k+1})] + \big[-c + (K-1)(\sqrt{K}G_{max} - 2c) \big] ||x^{k+1} - x^k||^2 \tag{3.15}$$

Rearranging the (3.15) gives

$$Kg(x^{k+1}) + K\lambda h(x^{k+1}) \leq Kg(x^k) + K\lambda h(x^k) + \big[-c + (K-1)(\sqrt{K}G_{max} - 2c) \big] ||x^{k+1} - x^k||^2$$

Choose $c > \frac{K-1}{2K-1}\sqrt{K}G_{max}$, we conclude (3.10).

□

Lemma 3.2 (Cost-to-Go Estimate)**.** Suppose *Assumption A and Assumption B* hold, then the sequence $\{x^k\}$, generated by Algorithm 1, satisfies

$$f(x^{k+1}) - f(x^*) \leq \theta ||x^{k+1} - x^k|| \tag{3.16}$$

where $\theta = (G_{max}K + 2c)R$

Proof.

$$
\begin{aligned}
&f(x^{k+1}) - f(x^*) + 2c(x^{k+1} - x^k)^T(x^{k+1} - x^*) \\
&= g(x^{k+1}) - g(x^*) + h(x^{k+1}) - h(x^*) + 2c(x^{k+1} - x^k)^T(x^{k+1} - x^*) \\
&\leq (x^{k+1} - x^*)^T \nabla g(x^{k+1}) + h(x^{k+1}) - h(x^*) + 2c(x^{k+1} - x^k)^T(x^{k+1} - x^*) \\
&= \sum_{i=1}^{K} (x_i^{k+1} - x_i^*)\big[\nabla_i g(x^{k+1}) - \nabla u_i(x_i^{k+1}; x^k)\big] + \sum_{i=1}^{K} (x_i^{k+1} - x_i^*)^T\big[\nabla u_i(x_i^{k+1}; x^k)\big] + \\
&\quad h(x^{k+1}) - h(x^*) + 2c(x^{k+1} - x^k)^T(x^{k+1} - x^*)
\end{aligned} \tag{3.17}
$$

Notice that $x_i^{k+1} = argmin_{z_i \in X_i}\, u_i(z_i; x^k) + \lambda h_i(z_i) + c||z_i - x_i^k||^2$. Using the optimality condition of this problem, we claim that there exists some $\xi_i^{k+1} \in \partial h_i(x_i^{k+1})$ such that

$$
\begin{aligned}
0 &\geq (x_i^{k+1} - x_i^*)[\nabla u_i(x_i^{k+1}; x^k) + \lambda \xi_i^{k+1} + 2c(x_i^{k+1} - x_i^k)] && (3.18)\\
&\geq (x_i^{k+1} - x_i^*)[\nabla u_i(x_i^{k+1}; x^k)] + \lambda[h_i(x_i^{k+1}) - h_i(x_i^*)] + 2c(x_i^{k+1} - x_i^k)(x_i^{k+1} - x_i^*) && (3.19)
\end{aligned}
$$

In view of (3.17) and (3.19), we derive

$$
\begin{aligned}
f(x^{k+1}) - f(x^*) + 2c(x^{k+1} - x^k)^T(x^{k+1} - x^*) &\leq \sum_{i=1}^{K} (x_i^{k+1} - x_i^*)\big[\nabla_i g(x^{k+1}) - \nabla u_i(x_i^{k+1}; x^k)\big] && (3.20)\\
&= \sum_{i=1}^{K} (x_i^{k+1} - x_i^*)\big[\nabla u_i(x_i^{k+1}; x^{k+1}) - \nabla u_i(x_i^{k+1}; x^k)\big] && (3.21)\\
&\leq \sum_{i=1}^{K} ||x_i^{k+1} - x_i^*||\ ||\nabla u_i(x_i^{k+1}; x^{k+1}) - \nabla u_i(x_i^{k+1}; x^k)|| && (3.22)\\
&\leq \sum_{i=1}^{K} G_i ||x^{k+1} - x^k||\ ||x_i^{k+1} - x_i^*|| && (3.23)\\
&\leq G_{max} RK ||x^{k+1} - x^k|| && (3.24)
\end{aligned}
$$

where the last inequality holds due to the Assumption A. Rearranging (3.24), we obtain

$$
f(x^{k+1}) - f(x^*) \leq G_{max} RK ||x^{k+1} - x^k|| + 2c(x^k - x^{k+1})^T(x^{k+1} - x^*) \leq (G_{max} K + 2c) R ||x^{k+1} - x^k||. \tag{3.25}
$$

□

Theorem 3.1. Suppose *Assumption A and B* hold then the sequence $\{x^k\}$, generated by Algorithm 1, satisfies,

$$\Delta_k = f(x^k) - f(x^*) \leq \frac{\alpha}{\sigma}\frac{1}{k}, \forall k \geq 1 \tag{3.26}$$

where $\sigma = \gamma/\theta^2$ and $\alpha = max\{f(x^1) - f(x^*), 4\sigma - 2, 2\}$.

Proof. From Lemma3.1 and Lemma3.2, we have

$$\Delta_k - \Delta_{k+1} \geq \gamma||x^{k+1} - x^k||^2 \geq \frac{\gamma}{\theta^2}\Delta_{k+1}^2 = \sigma\Delta_{k+1}^2$$

or equivalently,

$$\sigma\Delta_{k+1}^2 + \Delta_{k+1} \leq \Delta_k, \forall k \geq 1 \tag{3.27}$$

From (3.27) and $\Delta_1 \leq \alpha$, we derive

$$\Delta_2 = \frac{-1 + \sqrt{1 + 4\sigma\Delta_1}}{2\sigma} < \frac{-1 + \sqrt{1 + 4\sigma\alpha}}{2\sigma} = \frac{2\alpha}{1 + \sqrt{1 + 4\sigma\alpha}} \leq \frac{2\alpha}{1 + |4\sigma - 1|}$$

where the last inequality holds as $\alpha \geq 4\sigma - 2$. We assert that, at this point, $\Delta_2 \leq \frac{\alpha}{2\sigma}$, since

* When $4\sigma - 1 \geq 0$, then $\Delta_2 \leq \frac{\alpha}{2\sigma}$.

* When $4\sigma - 1 < 0$, then $\Delta_2 \leq \frac{2\alpha}{2-4\sigma} \leq \frac{2\alpha}{8\sigma-4\sigma} = \frac{\alpha}{2\sigma}$.

Now, we argue that if $\Delta_k \leq \frac{\alpha}{k\sigma}$, then the following holds

$$\Delta_{k+1} \leq \frac{\alpha}{(k+1)\sigma}.$$

It can be seen from the fact that

$$\Delta_{k+1} \leq \frac{-1 + \sqrt{1 + \frac{4\alpha}{k}}}{2\sigma} = \frac{2\alpha}{k\sigma(1 + \sqrt{1 + \frac{4\alpha}{k}})} \leq \frac{2\alpha}{\sigma(k + \sqrt{k^2 + 4k + 1}} = \frac{\alpha}{\sigma(k+1)} \tag{3.28}$$

where the last inequality holds due to the fact that $\alpha, k \geq 2$. Therefore, we conclude our proof.

□

Although techniques used to prove Theorem3.1 are quite similar to those in [7], it's noteworthy that the [7] employs the on Gauss-Seidel update strategy, so we cannot directly derive Theorem3.1 from it and need to find a new way instead. We achieve this by first establishing Lemma3.1 and Lemma3.2. The proofs of them fully exploit the Jacobi update structures, hence completely new. Based on Lemma3.1 and Lemma3.2, then Theorem3.1 can be easily derived.

3.2 Convergence Analysis for RPSUM

Before beginning our analysis, we need to introduce additional auxiliary variables for simplicity. Since the S_{t+1} is a random index set, there are actually $q = C_K^p$ possible values, and we denote them as $S_{t+1}^j, j = 1, ..., q$. By the construction of Algorithm 2, we know x^{t+1} depends on x^t through S_{t+1}. Therefore, there're q possible values of x^{t+1}. We explicitly formulate this relation discussed above through the set function

$$\phi(S_{t+1}^j; x^t) = w_j^{t+1}, j = 1, 2, ..., q$$

where w_j^{t+1} represents one possible value of x^{t+1}.

Lemma 3.3 (Expected Sufficient Descent)**.** Suppose *Assumption B* hold, then the sequence $\{x_t\}$, generated by Algorithm 2 , satisfies,

$$E[f(x^{t+1} - f(x^t))] \leq -\gamma_e E||x^{t+1} - x^t||^2 \tag{3.29}$$

where $K\gamma_e = p(\sqrt{K}G_{max} - 2c) > 0$ and $c > \frac{1}{2}\sqrt{K}G_{max}$ and $E[.]$ is taken with respect to random index sets $S_1, S_2, ..., S_t, S_{t+1}$.

Proof.

$$\begin{aligned}
E_{S_{t+1}}\big[f(x^{t+1}) - f(x^t)\big] &= E_{S_{t+1}}\big[\frac{1}{p}\sum_{i\in S_{t+1}}\{g(x_i^{t+1}, x_{-i}^{t+1}) - g(x_i^t, x_{-i}^t)\} + \lambda\sum_{i\in S_{t+1}}\{h_i(x_i^{t+1}) - h_i(x_i^t)\}\big] \\
&= \frac{1}{q}\sum_{j=1}^{q}\big[g(w_j^{t+1}) - g(x^t)\big] + \lambda\frac{p}{K}\big[h(\hat{x}^{t+1}) - h(x^t)\big] \\
&\overset{(1)}{\leq} \sum_{j=1}^{q}[(w_j^{t+1} - x^t)^T\nabla g(w_j^{t+1})\big] + \lambda\frac{p}{K}\big[h(\hat{x}^{t+1}) - h(x^t)\big] \\
&\overset{(2)}{=} \frac{p}{K}(\hat{x}^{t+1} - x^t)\nabla g(\hat{x}^{t+1}) + \lambda\frac{p}{K}\big[h(\hat{x}^{t+1}) - h(x^t)\big]
\end{aligned} \tag{3.30}$$

where $\hat{x}_i^{t+1}$ is generated by the Algorithm 1; i.e., $\hat{x}_i^{t+1} \leftarrow argmin_{z_i\in X_i}\, u_i(z_i; x^t) + \lambda h_i(z_i) + c||z_i - x_i^t||^2, i = 1, ..., K$ and x^t is generated by Algorithm 2. By convexity of $g(.)$, (1) holds. As for (2), notice the fact that those updated coordinate i, x_i^{t+1} will be counted for C_{K-1}^{p-1} times and $x_i^{t+1} = \hat{x}_i^{t+1}$.

Taking full expectation with respect to $S_1, ..., S_t$ on both sides of (3.30), we obtain

$$f(\tilde{x}^{t+1}) - f(\tilde{x}^t) \leq \frac{p}{K}(\tilde{x}^{t+1} - \tilde{x}^t)\nabla g(\tilde{x}^{t+1}) + \lambda\frac{p}{K}\left[h(\tilde{x}^{t+1}) - h(\tilde{x}^t)\right] \tag{3.31}$$

where $\tilde{x}_i^{t+1}$ is generated by the Algorithm 1; i.e., $\tilde{x}_i^{t+1} \leftarrow argmin_{z_i \in X_i}\, u_i(z_i; \tilde{x}^t) + \lambda h_i(z_i) + c||z_i - \tilde{x}_i^t||^2, i = 1, ..., K$.

Using the results from Lemma3.2, we have,

$$(\tilde{x}^{t+1} - \tilde{x}^t)\nabla g(\tilde{x}^{t+1}) \leq (\sqrt{K}G_{max} - 2c)||\tilde{x}^{t+1} - \tilde{x}^t||^2 + \lambda(\tilde{h}(x^t) - h(\tilde{x}^{t+1})) \tag{3.32}$$

Plugging (3.32) into (3.31) leads to

$$E(f(x^{t+1}) - f(x^t)) = f(\tilde{x}^{t+1}) - f(\tilde{x}^t) \leq \frac{p}{K}(\sqrt{K}G_{max} - 2c)||\tilde{x}^{t+1} - \tilde{x}^t||^2 = \frac{p}{K}(\sqrt{K}G_{max} - 2c)E||x^{t+1} - x^t||^2$$

□

Remark The technique used here are more straightforward than that used in Lemma3.10, but it also requires us to choose larger c than that of Lemma3.10 to guarantee the sequence $E[f(x^t)]$ are non-increasing.

To go on our proof, we need to modify the **Assumption A** as

Assumption C

Global minimum of Problem(1.1) can be attained on a set $\mathbb{E}$, where all of its elements are generated by Algorithm 2. There's a finite R_e such that the level set for $f(.)$ defined by x^0 is bounded, that is,

$$\max_{x \in \mathbb{E}} \max_{x} \left\{E||x - x^*||^2 : E[f(x)] < f(x^0)\right\} \leq R_e^2$$

where expectation is taken with respect to all of random variables. Now we give the counterpart of Lemma 3.2.

Lemma 3.4 (Expected Cost-to-Go Estimate)**.** Suppose *Assumption B and Assumption C* hold, then

$$E[f(x^{k+1}] - f(x^*) \leq \theta_e\sqrt{E||x^{t+1} - x^t||^2} \tag{3.33}$$

where $\theta_e = (pG_{max} + 2c)R_e$

Proof. The proof is almost the same as Lemma3.2. We only address the different part here. It's easy to derive

$$f(x^{t+1}) - f(x^*) + 2c(x^{t+1} - x^t)^T(x^{t+1} - x^*) \leq \sum_{i \in S_{t+1}} G_i ||x^{t+1} - x^t||\ ||x_i^{t+1} - x_i^*||$$

Taking expectation with S_{t+1} on both sides and using Cauchy-Schwarz inequality, we derive

$$\begin{aligned}
&E_{S_{t+1}}\big[f(x^{t+1}) - f(x^*)\big] \leq E_{S_{t+1}}\big\{||x^{t+1} - x^t|| \sum_{i \in S_{t+1}} G_i ||x_i^{t+1} - x_i^*||\big\} + E_{S_{t+1}}\big\{2c(x^t - x^{t+1})^T(x^{t+1} - x^*)\big\} \\
&\leq G_{max}\sqrt{E_{S_{t+1}}||x^{t+1} - x^t||^2}\sqrt{E_{S_{t+1}}\{\sum_{i \in S_{t+1}} ||x_i^{t+1} - x_i^*||\}^2} + 2c\sqrt{E_{S_{t+1}}||x^{t+1} - x^t||^2}\sqrt{E_{S_{t+1}}||x^{t+1} - x^*||^2} \\
&\overset{(1)}{\leq} G_{max}\sqrt{E_{S_{t+1}}||x^{t+1} - x^t||^2}\sqrt{E_{S_{t+1}}\{p\sum_{i \in S_{t+1}} ||x_i^{t+1} - x_i^*||^2\}} + 2c\sqrt{E_{S_{t+1}}||x^{t+1} - x^t||^2}\sqrt{E_{S_{t+1}}||x^{t+1} - x^*||^2} \\
&= G_{max}p\sqrt{E_{S_{t+1}}||x^{t+1} - x^t||^2}\sqrt{E_i||x_i^{t+1} - x_i^*||^2} + 2c\sqrt{E_{S_{t+1}}||x^{t+1} - x^t||^2}\sqrt{E_{S_{t+1}}||x^{t+1} - x^*||^2} \\
&\overset{(2)}{\leq} (pG_{max} + 2c)R_e\sqrt{E_{S_{t+1}}||x^{t+1} - x^t||^2} = \theta_e\sqrt{E_{S_{t+1}}||x^{t+1} - x^t||^2}
\end{aligned} \tag{3.34}$$

The (1) inequality holds due to the fact that $(\sum_{i=1}^n a_i)^2 \leq n\sum_{i=1}^n a_i^2,\ a_i \in \mathbb{R}$; the (2) holds as the result of $E[f(x^{t+1}] \leq f(x^0)$(derived from Lemma3.3).

Taking full expectation on both sides of (3.34) leads to

$$E[f(x^{t+1})] - f(x^*) \leq E[\theta_e\sqrt{E_{S_{t+1}}||x^{t+1} - x^t||^2}] \leq \theta_e\sqrt{E||x^{t+1} - x^t||^2}$$

where in the last inequality, we use the fact that $E(l(x)) \leq l(E(x))$, if $l(x)$ is concave. Hence, we conclude our claim. □

Follow the results derived in Theorem 3.1, we can immediately assert

Theorem 3.2. Suppose *Assumption B and Assumption C* hold, then the sequence $\{x^t\}$, generated by Algorithm 2, satisfies,

$$\Delta_t = E[f(x^t) - f(x^*)] \leq \frac{\alpha_e}{\sigma_e}\frac{1}{t}, \forall t \geq 1 \tag{3.35}$$

where $\sigma_e = \gamma_e/\theta_e^2$ and $\alpha_e = max\{E[f(x^1)] - f(x^*), 4\sigma_e - 2, 2\}$.

4 Application

In this section, we use the proposed RPSUM to solve regularized logistic regression problem. The setup for logistic regression are as usual. Suppose we have N pairs of observation (x_i, y_i), where $x_i \in \mathbb{R}^p$ are predict variables and $y_i \in \mathbb{R}$ are binary response variables. Without loss of generality, we assume that x_{ij} are standardized: $\sum_{i=1}^{N} x_{ij} = 0, \frac{1}{N}\sum_{i=1}^{N} x_{ij}^2 = 1,$ for $j = 1, 2, ..., p.$ and response variable y_i take value in $\{0, 1\}$.

The logistic regression model represents the class-conditional probabilities for all pairs of (x_i, y_i) through a linear function of the predictors

$$\begin{aligned} P(y_i = 0|x_i) =& \frac{1}{1 + e^{-(\beta_0 + x_i^T \beta)}} \\ P(y_i = 1|x_i) =& \frac{1}{1 + e^{(\beta_0 + x_i^T \beta)}} \end{aligned} \tag{4.1}$$

where $\beta \in \mathbb{R}^p, \beta_0 \in \mathbb{R}$. Equivalently, (4.1) implies that

$$\frac{P(y_i = 0|x_i)}{P(y_i = 1|x_i)} = \beta_0 + x_i^T \beta \tag{4.2}$$

In order to find the proper β_0, β, a widely employed strategy is to maximize the log likelihood

$$\ell(\beta_0, \beta) = \sum_{i=1}^{N} [y_i(\beta_0 + x_i^T \beta) - ln(1 + e^{\beta_0 + x_i^T \beta})] \tag{4.3}$$

which is a concave function of parameter β_0, β.

The ℓ_1-regularized logistic regression is to maximize log-likelihood function with ℓ_1 penalty

$$\underset{(\beta_0, \beta) \in \mathbb{R}^{p+1}}{maximize} \quad \ell(\beta_0, \beta) - \lambda|\beta|_1 \tag{4.4}$$

Solving Problem (4.4) is equivalent to tackle with the following minimization problem

$$\underset{(\beta_0,\beta)\in\mathbb{R}^{p+1}}{minimize} \quad f(\beta_0,\beta) = -\frac{1}{2N}\ell(\beta_0,\beta) + \lambda|\beta|_1 \triangleq g(\beta_0,\beta) + h(\beta_0,\beta) \tag{4.5}$$

Since $\ell(\beta_0,\beta)$ is concave, $f(\beta_0,\beta)$ is convex, hence Algorithms1 and Algorithm2 can be employed.

4.1 RPSUM for Logistic Regression with ℓ_1 regularization

To solve Problem(4.5), we need first to partition the argument $(\beta_0,\beta)\in\mathbb{R}^{p+1}$ into $p+1$ coordinates(blocks), that is $(\beta_0,\beta) = (\beta_0,\beta_1,...,\beta_p)$ and then specify the form of $u_j(.;\beta_0,\beta)$ as

$$u_j(z_j;\beta_0,\beta) = g(\beta_0,\beta) + (z_j-\beta_j)\nabla_j g(\beta_0,\beta) + \frac{L}{2}(z_j-\beta_j)^2, j = 0,1,2,...,p \tag{4.6}$$

Since the largest eigenvalue of $\nabla^2 g(\beta_0,\beta_p)$, denoted as $\lambda_{max}(\nabla^2 g(\beta_0,\beta_p))$, is no bigger than $\frac{1}{4}$, therefore we set L in (4.6) to $\frac{1}{4}$. Next, we dive into the core of both Algorithms1 and Algorithm2. Due to the explicit form of Problem (4.5), we can solve the low dimension problem

$$\underset{z_j\in\mathbb{R}}{min}\ u_j(z_j;\beta_0,\beta) + \lambda|z_j| + c||z_i-\beta_j||^2 = (z_j-\beta_j)\nabla_j g(\beta_0,\beta) + (\frac{1}{8}+c)(z_j-\beta_j)^2 + \lambda|z_j| \quad j = 1,2,...p \tag{4.7}$$

precisely in a closed form since it's a lasso type problem. For $j=0$ case, the above (4.7) simply becomes quadratic optimization problem, which takes form

$$\underset{z_0\in\mathbb{R}}{min}(z_0-\beta_0)\nabla_0 g(\beta_0,\beta) + (\frac{1}{8}+c)(z_0-\beta_0)^2 \tag{4.8}$$

where $\nabla_0 g(\beta_0,\beta)$ is the partial gradient with respect to β_0. Solving (4.7) is equivalent to solve

$$\underset{z_j\in\mathbb{R}}{min}\ (\frac{1}{8}+c)z_j^2 + m_j z_j + \lambda|z_j| \quad j = 1,2,...,p$$

and thus

$$z_j = \begin{cases} \frac{-m_j-\lambda}{2(\frac{1}{8}+c)} & m_j < 0, |m_j| > \lambda \\ \frac{-m_j+\lambda}{2(\frac{1}{8}+c)} & m_j > 0, |m_j| > \lambda \\ 0 & \text{otherwise} \end{cases} \qquad j = 1, 2, ..., p$$

where $m_j = \nabla_j f(\beta_0, \beta) - 2(\frac{1}{8} + c)\beta_j$. As for (4.8), simple calculations imply that

$$z_0 = \frac{-m_0}{2(\frac{1}{8} + c)} .$$

Now we give a pseudocode for solving for Logistic Regression with ℓ_1 regularization to conclude this subsection.(See Algorithm3.)

Algorithm 3 RPSUMfor Logistic Regression with ℓ_1 regularization

Initialization
$t \leftarrow 0$
Set $\beta_0 = 0, \beta = 0_p$,where 0_p represent a p dimension vector whose elements are all 0.
while not converged **do**
    In parallel on q processors
        Choose $j \in \{0, 1, 2, ..., p\}$ uniformly at random.
        $\beta_j^{t+1} \leftarrow z_j^t$, where z_j^t are defined as above but use β_0^t, β^t where appropriate.
    For those coordinates(js) that are not chosen, $\beta_j^{t+1} \leftarrow \beta_j^t$
    $t \leftarrow t + 1$
end while

4.2 Experiments

In this section, we perform our proposed algorithms for Logistic Regression with ℓ_1 regularization on both real and simulated datasets. We make a few but important statements on the way our numerical experiments conducted.

- An Intel Xeon machine is used for experiments. All codes are written in Python and the multiprocessing library is used for parallel computation[14].

- We denote the lower bound $\frac{1}{2}\sqrt{K}G_{max}$ as $\bar{c}$ and to guarantee the theoretic convergence, we set $c = \bar{c}(1 + 0.01)$.

- We make slight twist to our algorithms to address the unavoidable communication overhead[10]. In one iteration, we update 40 scalar variables on each processor sequentially. We still refer it as a method that updates q (number of processors available) variables in parallel. To be more specific, if we have 4 threads available , then in each iteration, we will update 160 scalar variables, with 40 scalar variables processed by per processor. As for how to assign the 40 scalar variables for each processor, we employ two rules. One is cyclic rule that corresponds to PSUM while the other is random rules that represents the RPSUM. For example, if $p = 120$ and the number of available processors is 2, then for the cyclic rule and random rule, tasks are assigned as shown in Table 1.

Iteration	Cyclic Rule		Random Rule	
	Processor I	Processor II	Processor I	Processor II
1	update 1-40 coordinates	update 41-80 coordinates	randomly choose 40 coordinates	randomly choose 40 coordinates
2	update 81-120 coordinates	update 1-40 coordinates	randomly choose 40 coordinates	randomly choose 40 coordinates
3	...	...	...	...

Table 1: A simple illustration of the task scheduling.

- To make our results comparable, we set $q = 1$ in our program to represent the serial counterparts.

4.2.1 Real Data

We use Leukemia [6], which is gene-expression data with a binary response indicating type of leukemia—AML vs ALL, to conduct the numerical experiments. We used the preprocessed data of [5]. The results are given in Figure 1.

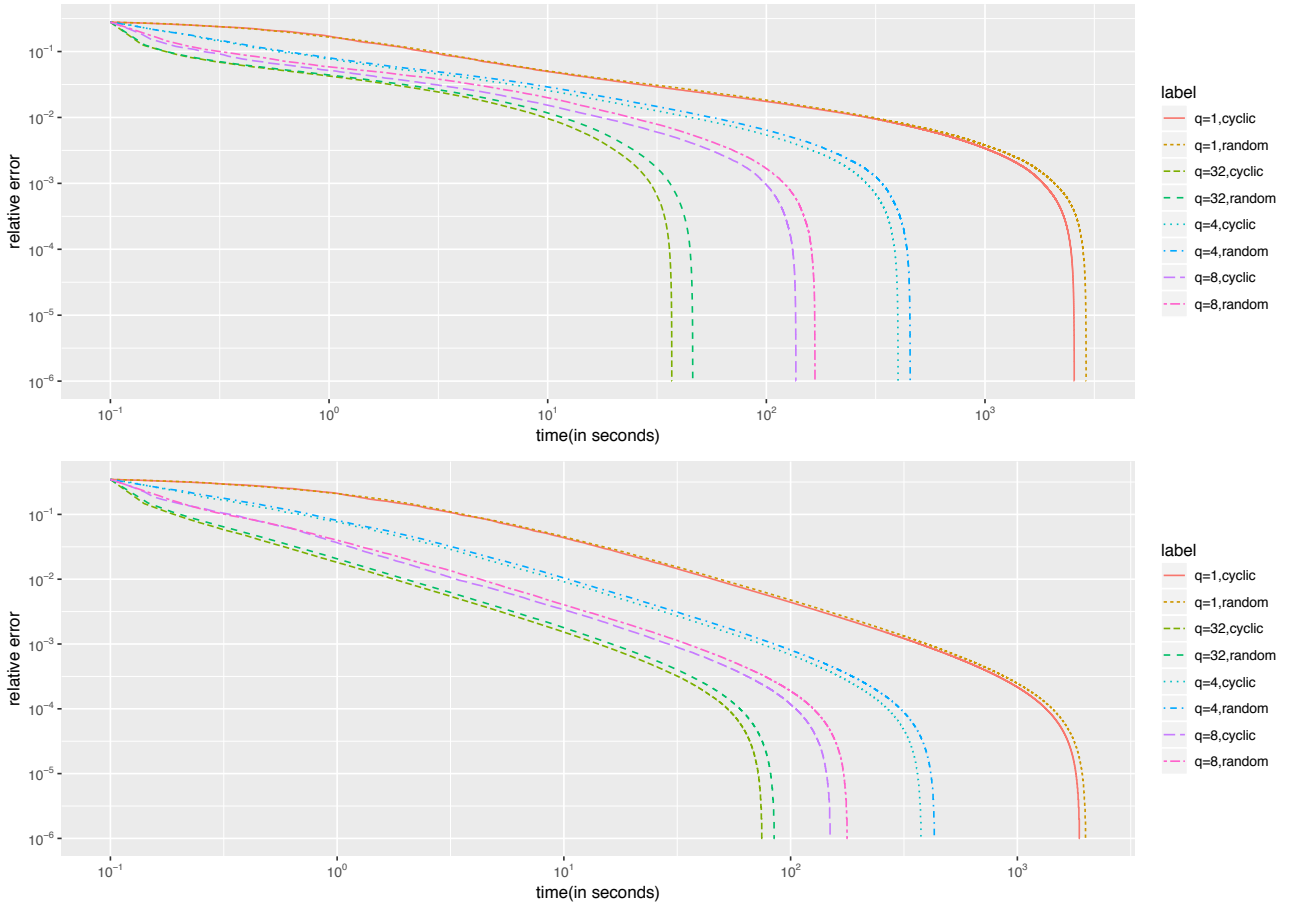

Figure 1: Performance comparison for RPSUM and PSUM with different degrees of regularization. The top figure represents the strong regularization situation with $\lambda = 10^{-2}$ while the bottom one correspond to the $\lambda = 10^{-6}$.

The y axis represents the relative error, which is defined by $f(x^{t+1}) - f^*$. It's quite clear that, with the increase with of available processors, we gain the near-linear speed up. Also, the number of data access for each method is near the same(see Table 2), so the comparison is really fair. It's also worth noting that the unpleasant phenomenon in [15] does not occur in this groups of experiments, which means we may assert that the more processors you have, more time you can save.

Methods	$\lambda = 10^{-2}$	$\lambda = 10^{-6}$
q=1,cyclic	2410880	1760880
q=1,random	2502640	1761320
q=4,cyclic	2401600	2254080
q=4,random	2399520	2253440
q=8,cyclic	2387200	2486720
q=8,random	2390400	2489920
q=32,cyclic	22627840	2728960
q=32,random	2659840	2731520

Table 2: Number of data access of Leukemia dataset with different q.

4.2.2 Simulated Data

We follow the simulation setup described in [5].

- First, generate the coefficient

$$\beta_j = (-1)^j e^{(\frac{2(j-1)}{20})}, j = 1, 2, ..., p.$$

- Second, generate the design matrix X of size $n \times p$.Then each row of X represents one observation, denoted as x_i.

- Third, generate the error vector ε of size $n \times 1$, with each element drawn from the Normal Distribution $N(0, 1)$. Together with X and β, we produce the working response Z,

$$Z = X\beta + \varepsilon.$$

- Finally, define $p_i = 1/(1 + exp(-Z[i]), i = 1, 2, ..., N$, where $Z[i]$ is the i-th element of Z. And use $Z[i]$ to generate the y_i with $\Pr(y_i = 1) = p_i$, $\Pr(y_i = 0) = 1 - p_i$.

We use the set up mentioned above to generate two datasets, one is of size $N = 1000, p = 10000$ the other is of size $N = 10000, p = 1000$.

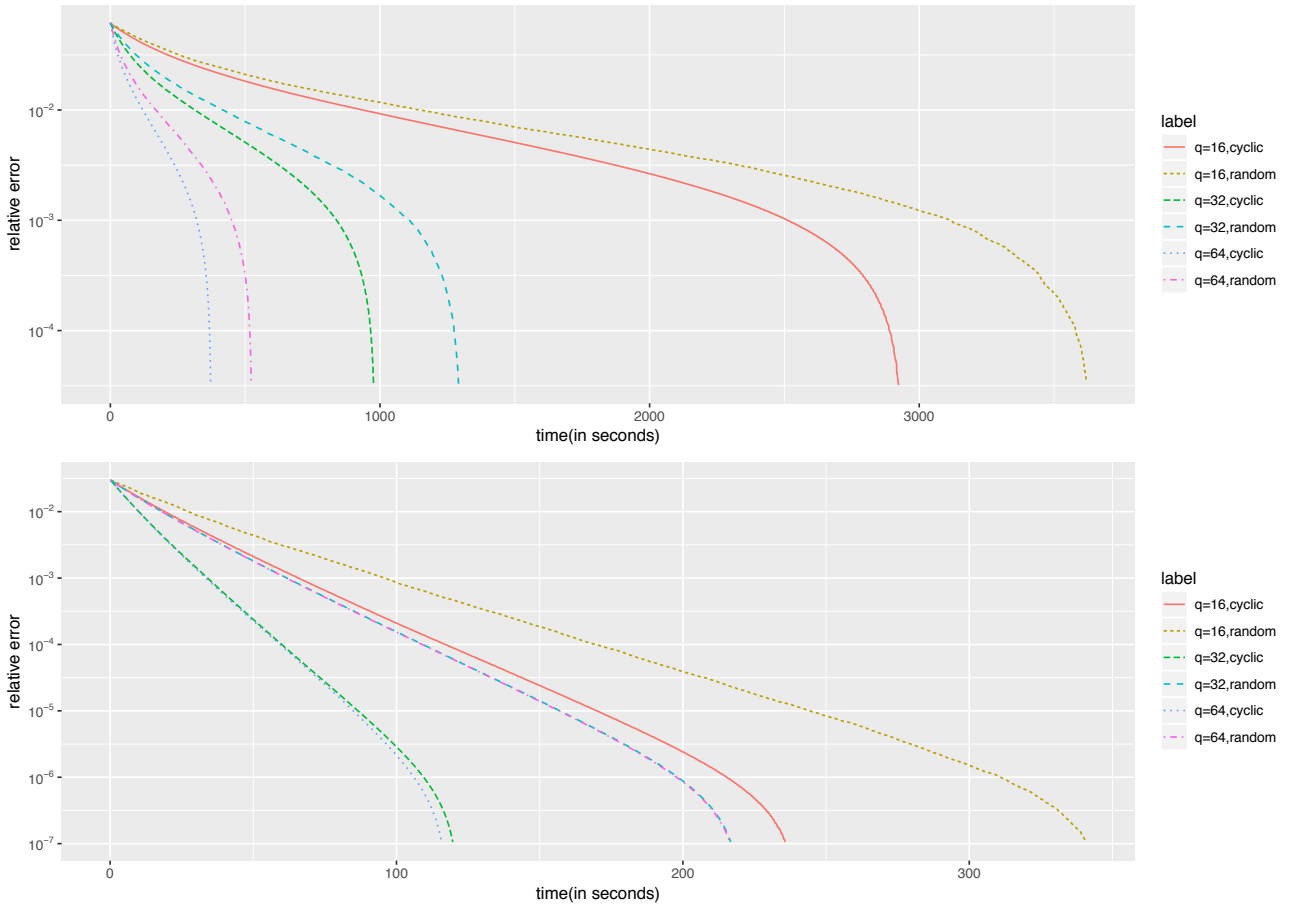

Figure 2: The above figure corresponds to the experiment results on the dataset N=1000,p=10000, while the bottom one reprsents the results of the dataset N=10000,p=1000.

We observe two interesting things here.

- Solving the $N > p$ problem is much easier than solving the $p > N$ problem.
- The increase in the number of processors does not guarantee the decrease of time to reach the same global optimal. In fact, after some points, the increase of processors will require more time. This is as the result of the communication overhead overwhelm the computation time. This observation coincides with results in [15]. It means that though in theory you can use as many as processor as you like, in fact, it does not guarantee the linear computation gain.

For the sake of saving space here, we omit the data on the number of data access for each method.

5 Discussions and Conclusions

In this section, we first make some remarks about the technique details in designing and implementing our proposed approach, and then finish this paper with concise conclusions.

Jacobi versus Gauss-Sediel. Many literatures indicate that methods with Jacobi update strategy is usually slower than those with Gauss-Seidel update strategy, e.g. [23]. Here, we also give a simple verification example below, see Figure 3 .

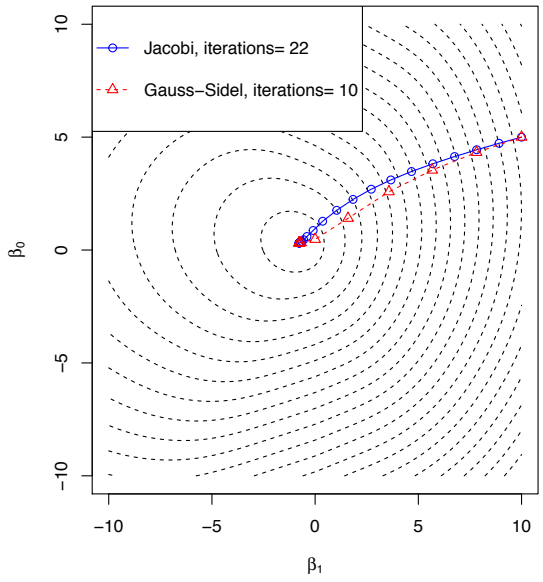

Figure 3: A simple comparison between Jacobi update strategy and Gauss-Seidel update strategy.

Clearly, the iterations of Jacobi style are twice as many as those of Gauss-Seidel's. And it seems that Jacobi strategy is much more conservative than the Gauss-Seidel strategy. Unfortunately, in the synchronous parallelism setting, the Gauss-Seidel strategy is not a qualified candidate. However, in the asynchronous parallelism setting, it's worth considering the Gauss-Seidel strategy. In [8], Liu et al. propose the "inconsistent" read model, which is really close to the real asynchronous computing. The only defect in their algorithm is that it has to first wait for all processors to finish updating assigned coordinate and make current iterate information available for all processors, then the individual coordinate can go on its work. For detailed discussion, see [8]. Hence, to overcome this shortcoming, we wonder if it is possible to achieve true asynchronous parallelism by cancelling the "synchronization" step in [8]. We shall dive into it and provide some theoretical analysis.

Smaller c. In Problem1.1, x_i is updated by solving the following problem

$$\underset{z_i \in X_i}{argmin}\ u_i(z_i; x^t) + \lambda h_i(z_i) + c||z_i - x_i^t||^2.$$

As indicated in the Theorem3.1 and Theorem3.2, there's a lower bound $\bar{c}$ on the parameter c to guarantee the theoretical convergence of our proposed PSUM and RPSUM algorithms. However, we observed in our numerical experiments that we can choose a c', which is a little bit smaller than the $\bar{c}$, to reach the same optimal solution with fewer iterations and less time.

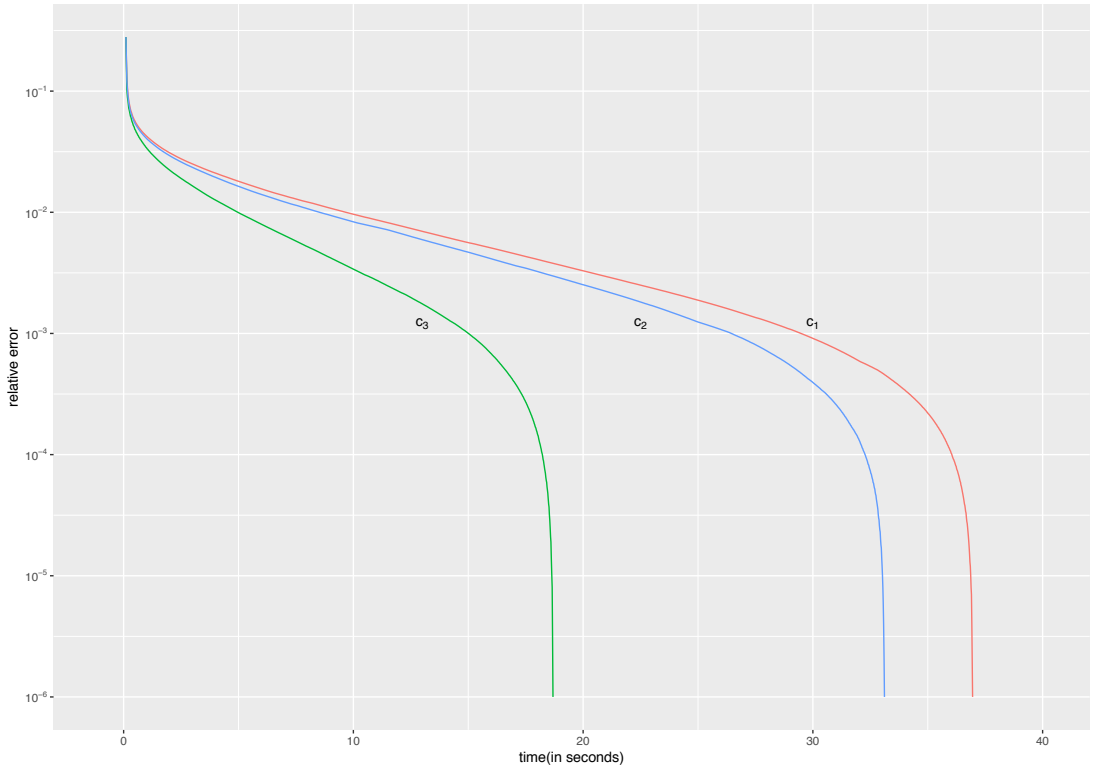

Figure 4: Different c results in different number of iterations to reach the same global optimal solution.

This phenomenon is as the results of the fact that the instructed function $u_i(z_i; x^t) + \lambda h_i(z_i) + c||z_i - x_i^t||^2$ is not a locally tight upper bound function of the $f(z)$. The smaller c' is, the upper approximating function approximates $f(z)$ better. To visualize this assertion, please see Figure 5.

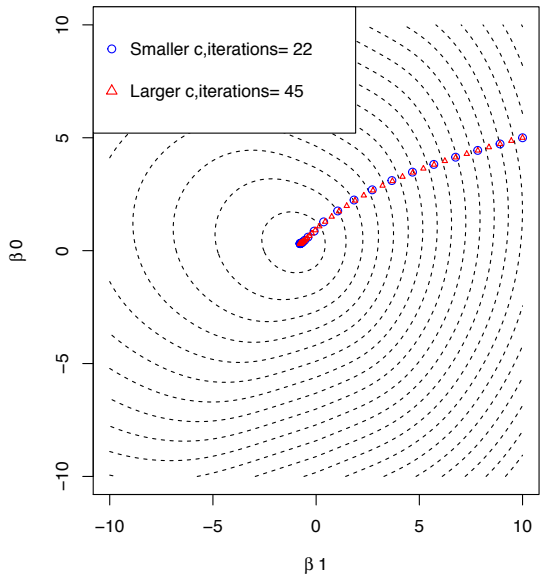

Figure 5: An illustration for the role c plays in our proposed algorithms.

It can be seen from this figure that, the smaller c' results an aggressive stride, hence fewer iterations while the larger c will produce a conservative step-size . It's also worth pointing out that c can not be too small, otherwise the PSUM and RPSUM algorithms will diverge.

Why does the randomization fail? It can be learned from the experiment results that the random block selection rule is inferior to the cyclic rule. One possible explanation is that no matter which coordinate you choose to optimize, you will obtain the relative the same decrease in object function. But randomly choose coordinates to update shall require extra time.

Conclusions. The fundamental problem in many application scenarios is how to minimize a nonsmooth convex composite function in an efficient manner. A family of popular solvers for this problem are serial block coordinate descent (BCD) but they are not efficient in many practical applications. Thanks to the development of multi-core parallel processing technology, it makes the speed boosting feasible by parallelizing these BCD-like algorithms. Towards this end, we propose parallel block coordinate descent algorithms called PSUM and RPSUM, which are endowed with sub-linear convergence rate for general convex composite function. Different from existing algorithms, the proposed PSUM and RPSUM algorithms employ general approximation function to optimize at each iteration. Therefore, some existing synchronous parallel algorithms can be treated as special cases of the PSUM. The experiments on solving the regularized logistic regression with ℓ_1 norm problem show that the proposed parallel algorithms are quite efficient to implement.

Appendix

Source codes are hosted on https://github.com/Rothdyt/Projects/tree/master/PSUM-source-code. The R code is used to build prototype of two algorithms and the python codes are used to implement all experiments. Due to the storage limitation of github, the simulated datasets are not provided here. Only the Leukemia dataset is uploaded.

References

[1] Goran Banjac, Kostas Margellos, and Paul J Goulart. "On the convergence of a regularized Jacobi algorithm for convex optimization". In: *arXiv* (2016).

[2] Amir Beck and Luba Tetruashvili. "On the Convergence of Block Coordinate Descent Type Methods". In: *Siam Journal on Optimization* 23.4 (2013), pp. 2037–2060.

[3] Joseph K Bradley et al. "Parallel Coordinate Descent for L1-Regularized Loss Minimization". In: *ICML* (2011).

[4] Luca Deori, Kostas Margellos, and Maria Prandini. "A regularized Jacobi algorithm for multi-agent convex optimization problems with separable constraint sets". In: *arXiv* (2016).

[5] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. "Regularization Paths for Generalized Linear Models via Coordinate Descent". In: *Journal of Statistical Software* 33.i01 (2010), p. 1.

[6] T. R. Golub et al. "Molecular Classification of Cancer: Class Discovery and Class Prediction by Gene Expression Monitoring". In: *Brain Research* 501.2 (1999), pp. 205–14.

[7] Mingyi Hong et al. "Iteration complexity analysis of block coordinate descent methods". In: *Mathematical Programming* (2013), pp. 1–30.

[8] Ji Liu and Stephen J. Wright. "Asynchronous Stochastic Coordinate Descent: Parallelism and Convergence Properties". In: *Siam Journal on Optimization* 25.1 (2015).

[9] Z. Q. Luo and P. Tseng. "On the convergence of the coordinate descent method for convex differentiable minimization". In: *Journal of Optimization Theory and Applications* 72.1 (1992), pp. 7–35.

[10] Norman Matloff. *Parallel Computing for Data Science: With Examples in R, C++ and CUDA*. CRC Press, 2015.

[11] Yu. Nesterov. "Efficiency of Coordinate Descent Methods on Huge-Scale Optimization Problems". In: *Siam Journal on Optimization* 22.2010002 (2010), pp. 341–362.

[12] Yurii Nesterov. "A method of solving a convex programming problem with convergence rate O (1/k2)". In: *Soviet Mathemtics Doklady* 27(2) (1983), pp. 372–376.

[13] Andrew Ng. "Efficient L1 Regularized Logistic Regression". In: *In AAAI-06* 1 (2006), pp. 1–9.

[14] Python. *Python 2.7.13 documentation*. Python Software Foundation. 2017. URL: https://docs.python.org/2/index.html.

[15] Meisam Razaviyayn et al. “Parallel Successive Convex Approximation for Nonsmooth Nonconvex Optimization”. In: *Advances in Neural Information Processing Systems* 2 (2014), pp. 1440–1448.

[16] Peter Richtárik and Martin Takáč. “Iteration complexity of randomized block-coordinate descent methods for minimizing a composite function”. In: *Mathematical Programming* 144.1 (2014), pp. 1–38.

[17] Peter Richtárik and Martin Takáč. “Parallel coordinate descent methods for big data optimization”. In: *Mathematical Programming* 156.1 (2016), pp. 433–484.

[18] Shai Shalev-Shwartz and Ambuj Tewari. “Stochastic Methods for ℓ_1 Regularized Loss Minimization”. In: *Journal of Machine Learning Research* 12.2 (2011), pp. 1865–1892.

[19] R Tibshirani. “Regression Selection and Shrinkage via the Lasso”. In: *Journal of Royal Statistical Scociety.* B(Methodological) (1996), pp. 267–288.

[20] P Tseng. “Convergence of a block coordinate descent method for nondifferentiable minimization”. In: *Journal of Optimization Theory and Applications* 109.3 (2001), pp. 475–494.

[21] Paul Tseng and Sangwoon Yun. “A coordinate gradient descent method for nonsmooth separable minimization”. In: *Mathematical Programming* 117.1 (2009), pp. 387–423.

[22] Stephen J Wright. “Coordinate descent algorithms”. In: *Mathematical Programming* 151.1 (2015), pp. 3–34.

[23] Yangyang Xu. “Hybrid Jacobian and Gauss-Seidel proximal block coordinate update methods for linearly constrained convex programming”. In: *arXiv* (2016).