Async-LP

Hu Jinqiu

October 5, 2018

Contents

1 Background and Motivation 3

2 Different Forms of Linear Programming 3
- 2.1 Standard Form . . . 3
- 2.2 Dual Form . . . 3
- 2.3 Self-dual Embedding Form . . . 4

3 First-order Method 5
- 3.1 Douglas-Rachford Splitting . . . 5
- 3.2 Primal-Dual Splitting . . . 8
- 3.3 Gradient Projection . . . 9
- 3.4 FISTA . . . 9

4 Asynchronously Parallel 9
- 4.1 Coordinate Friendly and ARock . . . 10
- 4.2 Douglas-Rachford Splitting . . . 10
- 4.3 Primal-Dual Splitting . . . 11
- 4.4 Gradient Projection . . . 13
- 4.5 FISTA . . . 13

5 Convergence 14
- 5.1 Douglas-Rachford Splitting . . . 14
- 5.2 Primal-Dual Splitting . . . 15
- 5.3 Gradient Projection . . . 15
- 5.4 FISTA . . . 15

6 Numerical Results 16
- 6.1 Used Problems . . . 16
- 6.2 Stopping condition . . . 16
- 6.3 Stopping check . . . 17
- 6.4 Douglas-Rachford Splitting . . . 17
- 6.5 Primal-Dual Splitting . . . 18
  - 6.5.1 Precondition for Primal-Dual Splitting . . . 18
  - 6.5.2 Parameters and Results . . . 19
- 6.6 Other things may need to be mentioned . . . 19

7 Further Work 19

1 Background and Motivation

Linear programs are problems that the objective function, equality constraints and inequality constraints are all linear functions. This is a special case of convex optimization. Since linear programming was introduced, this has been widely used in different fields according to [6]. Many practical problems can be transformed into LP. The maximum network-flow problem and shortest path problem can be expressed as LP. Company's production manager may use LP to design production plan to archive maximum profit with limited raw materials [5]. Moreover, many problems in machine learning are LP. For example, the support vector machine with L1 norm regularization is a LP.

There are many algorithms to solve LP nowadays, such as simplex method and interior-point method. These methods are now widely used. However, with the development of machine learning and big data, we should deal with problem with more data and higher dimensions, while these methods may be not efficient enough. Thus we need to look for other methods or apply new techniques on existing ones.

A technique called ARock, gives a framework and convergence guarantees for asynchronous parallel computing [3]. In this project, we will use this framework on some first-order methods to solve LP.

2 Different Forms of Linear Programming

There are many forms of linear programming used in different problems. Here we focus on some ones.

2.1 Standard Form

The standard form of linear programming can be written as:

$$\min_x \ c^T x \quad s.t. \ Ax = b, \ x \succeq 0,$$

where $A \in \mathbb{R}^{p \times m}$, $x, c \in \mathbb{R}^m$ and $b \in \mathbb{R}^p$. Without loss of generality, we can assume that $\operatorname{rank} A = p < m$. We will use *primal problem* to name this form. This form can be written as

$$\min_x \ c^T x + 1_{Ax=b}(x) + 1_{x \succeq 0}(x).$$

2.2 Dual Form

The dual form of the standard form of linear programming is

$$\min_s \ b^T s \quad s.t. \ \ c + A^T s \succeq 0.$$

Obviously, the dual form of the dual form is the primal form. This form can be written as

$$\min_s \ b^T s + 1_{c + A^T s \succeq 0}(s)$$

2.3 Self-dual Embedding Form

For LP with standard form

$$\min \quad c^T x \quad s.t. \quad Ax = b, \; x \geq 0,$$

according to [7], the basic self-dual embedding form of this LP is

$$\begin{aligned} &\min 0 \\ &\begin{pmatrix} 0 & A & -b \\ -A^T & 0 & c \\ b^T & -c^T & 0 \end{pmatrix} \begin{pmatrix} y \\ x \\ \tau \end{pmatrix} = \begin{pmatrix} 0 \\ s \\ k \end{pmatrix} \\ &s.t. \; x, \tau, s, k \geq 0. \end{aligned}$$

This form uses more variables than others, but we could get more information from the result. Assume that $(y^*, x^*, \tau^*, s^*, k^*)$ is the strictly complementary solution, which satisfies

$$s^T x + k\tau = 0, \quad k + \tau > 0,$$

we can know whether the linear programming is feasible. In particular, if $\tau^* > 0$ and $k^* = 0$, $x^*/\tau^*, y^*/\tau^*, s^*/\tau^*$ are primal, dual optimal and Kuhn-Tucker variable for the original linear programming. When $\tau^* = 0$ and $k^* > 0$, if $b^T y > 0$, the primal problem is infeasible and if $c^T x < 0$, the dual problem is infeasible.

Since this form is a feasible problem which means we only need to find feasible point and any feasible point will be the optimal one, we can transform this problem as quadratic programming. That is

$$\begin{aligned} &\min \frac{1}{2}\|Ax - \tau b\|_2^2 + \frac{1}{2}\| - A^T y + \tau c - s\|_2^2 + \frac{1}{2}(-c^T x + b^T y - k)^2 \\ &s.t. \; x, \tau, s, k \geq 0. \end{aligned}$$

Denote $g(y, x, \tau, s, k) = \frac{1}{2}\|Ax - \tau b\|_2^2 + \frac{1}{2}\| - A^T y + \tau c - s\|_2^2 + \frac{1}{2}(-c^T x + b^T y - k)^2$ and $\tilde{x} = (y \; x \; \tau \; s \; k)^T$, we can have $g(\tilde{x}) = \frac{1}{2}\tilde{x}^T \tilde{A}\tilde{x}$, where the semi-definite symmetric matrix has

$$\tilde{A} = \begin{pmatrix} AA^T + bb^T & -bc^T & -Ac & A & -b \\ -cb^T & A^T A + cc^T & -A^T b & 0 & c \\ -c^T A^T & -b^T A & c^T c + b^T b & -c^T & 0 \\ A^T & 0 & -c & I & 0 \\ -b^T & c^T & 0 & 0 & 1 \end{pmatrix}.$$

Then the problem can be written as

$$\min_{\tilde{x}} \; g(\tilde{x}) \quad s.t. \; \tilde{x} \in \mathcal{C}, \tag{2.1}$$

where $\mathcal{C} = \mathbb{R}^p \times \mathbb{R}_+^{p+2m+2}$

3 First-order Method

We have lots of algorithms to solve the linear programming, such as simplex method, interior method and other methods which can deal with not only linear programming but also other convex optimization problem. Considering that we want to give some async-parallel algorithms, some first-order methods can be used, such as Douglas-Rachford Splitting and Primal-Dual Splitting, to solve LP.

Before we introduce some popular first-order methods, we need to give some notations which will be frequently used in the following sections.

Definition 3.1 ((Proximal operator))**.** *We define the proximal operator for convex function f is*

$$\mathrm{prox}_f(x) = \mathrm{argmin}_y\, f(y) + \frac{1}{2}\|y - x\|^2.$$

And for convenience, we denote this as default.

$$(x)^+ = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

3.1 Douglas-Rachford Splitting

Douglas-Rachford Splitting(DRS) can be applied to the problem which can be written as:

$$\min_x f(x) + g(x),$$

where f and g are closed convex functions. The iteration can be written as:

$$\begin{aligned} x^{(k+1)} &= \mathrm{prox}_{\lambda f}(y^{(k)}) \\ y^{(k+1)} &= y^{(k)} + \mathrm{prox}_{\lambda g}(2x^{(k+1)} - y^{(k)}) - x^{(k+1)}, \end{aligned}$$

Since the proximal operator has been used in the iteration, DRS is very useful when f and g have in-expansive prox-operator.

It seems that the f and g are not symmetric, but the iteration has an equivalent form,

$$\begin{aligned} u^{(k)} &= \mathrm{prox}_{\lambda g}(x^{(k-1)} + \omega^{(k-1)}) \\ x^{(k)} &= \mathrm{prox}_{\lambda f}(u^{(k)} - \omega^{(k-1)}) \\ \omega^{(k)} &= w^{(k-1)} + x^{(k)} - u^{(k)}, \end{aligned}$$

and from this we can find, actually, the function f and g are symmetric in the iteration.

To apply this algorithm to the standard form of linear programming, we should determine what are f and g. However, there are three items in the objective function. Thus, considering the symmetry, we have three choices and we will talk about them one by one.

Case 1

When $f(x) = c^T x + 1_{x\succeq 0}(x)$ and $g(x) = 1_{Ax=b}(x)$, we can compute the proximal operator of each function.

$$\begin{aligned}
\mathrm{prox}_{\lambda f}(x) &= \mathrm{argmin}_{u\succeq 0}\, c^T u + \frac{1}{2\lambda}\|x-u\|^2 \\
&= (\mathrm{argmin}_{u_i}\, c_i u_i + \frac{1}{2\lambda}(x_i - u_i)^2) \\
&= (x_i - \lambda c_i) = (x-\lambda c)^+ \\
\mathrm{prox}_{\lambda g}(x) &= \mathrm{argmin}_{u}\, 1_{Au=b}(u) + \frac{1}{2\lambda}\|x-u\|^2 = \mathrm{proj}_{\{x|Ax=b\}}(u) \\
&= \mathrm{argmin}_{u|Au=b}\, \|x-u\|^2.
\end{aligned}$$

For $\mathrm{prox}_{\lambda g}(x)$, the KKT condition is

$$\begin{aligned}
&(u-x) + A^T\eta = 0 \quad Au = b \\
\Rightarrow & 0 = A(u-x) + AA^T\eta = b - Ax + AA^T\eta \\
\Rightarrow & \eta = (AA^T)^{-1}(Ax-b) \\
\Rightarrow & u = x + A^T(AA^T)^{-1}(Ax-b).
\end{aligned}$$

Thus we have

$$\begin{aligned}
&\mathrm{prox}_{\lambda f}(x) = (x-\lambda c)^+ \\
&\mathrm{prox}_{\lambda g}(x) = x + A^T(AA^T)^{-1}(b-Ax),
\end{aligned}$$

and the iteration is

$$\begin{aligned}
x^{(k+1)} &= (y^{(k)} - \lambda c)^+ \\
y^{(k+1)} &= x^{(k+1)} + A^T(AA^T)^{-1}(b + A(-2x^{(k+1)} + y^k)).
\end{aligned}$$

Case 2

When $f(x) = c^T x + 1_{Ax=b}(x)$ and $g(x) = 1_{x\succeq 0}(x)$, we can compute the proximal operator of function g and f.

$$\begin{aligned}
&\mathrm{prox}_{\lambda g}(x) = x^+ \\
&\mathrm{prox}_{\lambda f}(x) = \mathrm{argmin}_{\{u|Au=b\}}\, c^T u + \frac{1}{2\lambda}\|x-u\|^2,
\end{aligned}$$

and the KKT condition

$$\begin{aligned}
&\frac{1}{\lambda}(u-x) + A^T\eta + c = 0 \quad Au = b \\
\Rightarrow & Ac + \frac{1}{\lambda}(b - Ax) + AA^T\eta = 0 \\
\Rightarrow & \eta = (AA^T)^{-1}(\frac{1}{\lambda}(Ax-b) - Ac) \\
\Rightarrow & u = x - \lambda c + A^T(AA^T)^{-1}(Ax - b - \lambda Ac)
\end{aligned}$$

Thus we have

$$\begin{aligned}\mathrm{prox}_{\lambda f}(x) &= x - \lambda c + A^T(AA^T)^{-1}(Ax - b - \lambda Ac)\\ &= (I + A^T(AA^T)^{-1})(x - \lambda c) - A^T(AA^T)^{-1}b\\ \mathrm{prox}_{\lambda g}(x) &= x^+.\end{aligned}$$

Considering the symmetry of f and g, we can get

$$\begin{aligned}x^{(k+1)} &= (y^{(k)})^+\\ y^{(k+1)} &= y^{(k)} + (I + A^T(AA^T)^{-1})(2x^{(k+1)} - y^{(k)} - \lambda c) - x^{(k+1)}.\end{aligned}$$

After denoting $w = y + \lambda c$, we have

$$\begin{aligned}x^{(k+1)} &= (w^{(k)} - \lambda c)^+\\ w^{(k+1)} &= x^{(k+1)} + A^T(AA^T)^{-1}(b + A(-2x^{(k+1)} + w^k)),\end{aligned}$$

the same as that in Case 1. Thus this case is equivalent to Case 1.

Case 3

When $f(x) = 1_{x\succeq 0}(x) + 1_{Ax=b}(x)$ and $g(x) = c^Tx$, we can compute the proximal operator of function f. Then we should solve the problem

$$\begin{aligned}&\min_u \frac{1}{2\lambda}\|u - x\|\\ &s.t.\ Au = b\\ &\qquad u \succeq 0,\end{aligned}$$

which is a quadratic programming with both equality constraints and inequality constraints. Thus we should solve a quadratic programming partly or wholly during each iteration and the algorithm might be too difficult as a result. We don't take this case into further discussion.

With the discussion above, we choose the iteration in Case 1 to solve linear programming. After substituting $x^{(k+1)}$ into $y^{(k+1)}$, we can get

$$y^{(k+1)} = (y^{(k)} - \lambda c)^+ + A^T(AA^T)^{-1}(b + A(y^{(k)} - 2(y^{(k)} - \lambda c)^+)) \tag{3.1}$$

$$x^* = (y^* - \lambda c)^+. \tag{3.2}$$

For dual problem, if we set $f(s) = b^Ts$ and $g(s) = 1_{c+A^Ts\succeq 0}(s)$, we need to compute prox_g. However, prox_g is a quadratic programming with inequality constraint, which doesn't have closed form solution. To avoiding this problem, we can add relax variables to change the complicated inequality constraint into equality constraint with simple inequality constraint. We get $\min_{s,\tau} b^Ts + 1_{c+A^Ts-\tau=0}(s,\tau) + 1_{\tau\succeq 0}(\tau)$, which is the same as the standard form.

We can also apply this method to the self-dual embedding form of LP. With the form 2.1, we can set $f(\tilde{x}) = \frac{1}{2}\tilde{x}^T\tilde{A}\tilde{x}$ and $g(\tilde{x}) = 1_{\tilde{x}\in\mathcal{C}}(\tilde{x})$. Then we get

$$\begin{aligned}\mathrm{prox}_{\lambda f}(\tilde{x}) &= (I + \lambda\tilde{A})^{-1}\tilde{x}\\ \mathrm{prox}_{\lambda g}(\tilde{x}) &= \mathrm{proj}_{\mathcal{C}}(\tilde{x}),\end{aligned}$$

and the iteration is

$$\begin{aligned}\tilde{x}^{(k+1)} &= \mathrm{proj}_{\mathcal{C}}(\tilde{y}^{(k)})\\ \tilde{y}^{(k+1)} &= \tilde{y}^{(k)} + (I + \lambda\tilde{A})^{-1}(2\tilde{x}^{(k+1)} - \tilde{y}^{(k)}) - \tilde{x}^{(k+1)}.\end{aligned}$$

After substituting x, we have

$$\tilde{y}^{(k+1)} = \tilde{y}^{(k)} + (I + \lambda\tilde{A})^{-1}(2\,\mathrm{proj}_{\mathcal{C}}(\tilde{y}^{(k)}) - \tilde{y}^{(k)}) - \mathrm{proj}_{\mathcal{C}}(\tilde{y}^{(k)}).,\tag{3.3}$$

3.2 Primal-Dual Splitting

We can apply Primal-Dual Splitting(PD for short) to solve the problem with the form

$$\min_x f(x) + g(x) + h(Ax),$$

where f is differentiable convex function, g and h are closed convex functions and A, b, c, x is the same as those in the former section. Sometimes h is proximable but $h \circ A$ is non-proximable or diffcult to compute, so that we can use PD to separate h and A based on [2]. The iteration can be written as:

$$\begin{aligned}s^{(k+1)} &= \mathrm{prox}_{\gamma h^*}(s^{(k)} + \gamma Ax^{(k)})\\ x^{(k+1)} &= \mathrm{prox}_{\eta g}(x^{(k)} - \eta(\nabla f(x^{(k)}) + A^T(2s^{(k+1)} - s^{(k)}))).\end{aligned}$$

The LP problem can be written as

$$\min \quad c^T x + 1_{x\geq 0}(x) + 1_{\cdot=b}(Ax).$$

Obviously we can set $f(x) = c^T x$, $g(x) = 1_{x\geq 0}(x)$ and $h(x) = 1_{x=b}(x)$. Computing the dual and proximal of functions, we have

$$\begin{aligned}h^*(y) &= b^T y\\ \mathrm{prox}_{\gamma h^*}(x) &= x - \gamma b\\ \mathrm{prox}_{\eta g}(x) &= (x)^+.\end{aligned}$$

Thus the iteration is

$$\begin{aligned}s^{(k+1)} =& s^{(k)} + \gamma Ax^{(k)} - \gamma b\\ x^{(k+1)} =& (x^{(k)} - \eta(c + A^T(2s^{(k+1)} - s^{(k)})))^+.\end{aligned}$$

If we substitute s^{k+1}, the update of x will only use $x^{(k)}$ and $s^{(k)}$. The iteration is

$$\begin{aligned} s^{k+1} =& s^k + \gamma A x^k - \gamma b && (3.4)\\ x^{k+1} =& (x^k - \eta(c + A^T(s^k + 2\gamma A x^k - 2\gamma b)))^+. && (3.5) \end{aligned}$$

For dual problem, using primal-dual splitting might not bring more results than the standard form. Primal-dual splitting updates the primal variables and dual variables together in each iteration. Then primal-dual splitting for dual problem is the same as that for primal problem, because the dual variables of dual problem are the primal variables of primal problem.

3.3 Gradient Projection

Gradient projection can solve the problem with the form $\min_{x\in\mathcal{C}} g(x)$, where $\mathcal{C}$ is convex set and g is differentiable convex function. The iteration is

$$x^{(k+1)} = \text{proj}_{\mathcal{C}}(x - \eta\nabla g(x)).$$

For LP problem with self-embedding form, the iteration is

$$\tilde{x}^{(k+1)} = \text{proj}_{\mathcal{C}}((I - \eta\tilde{A})\tilde{x}^{(k)}). \tag{3.6}$$

3.4 FISTA

We can change the step size in gradient projection from constant to that rely on the number iterations to speed up the algorithm. There is a famous accelerated gradient projection named FISTA. The iteration is

$$\begin{aligned} y =& x^{k-1} + \frac{k-2}{k+1}(x^{k-1} - x^{k-2})\\ x^k =& \text{prox}_{t_k h}(y - t_k\nabla g(y)). \end{aligned}$$

For LP problem with self-embedding form, we set $g(\tilde{x}) = \frac{1}{2}\tilde{x}^T\tilde{A}\tilde{x}$ and $h(\tilde{x}) = 1_{\tilde{x}\in\mathcal{C}}(\tilde{x})$. Then the iteration is

$$\begin{aligned} y =& \tilde{x}^{k-1} + \frac{k-2}{k+1}(\tilde{x}^{k-1} - \tilde{x}^{k-2}) && (3.7)\\ \tilde{x}^k =& \text{proj}_{\mathcal{C}}(y - t_k\tilde{A}(y)). && (3.8) \end{aligned}$$

4 Asynchronously Parallel

In the following parts, we will disscuss how to transform each algorithm for asynchronously paralleling and whether they are parallel efficient. Firstly, we introduce some conceptions in [2] to describe parallel efficiency and a framework in [3] for async-parallel.

4.1 Coordinate Friendly and ARock

Definition 4.1 (number of operations)**.** *We let $\mathfrak{M}[a \to b]$ denote the number of basic operations that it takes to compute the quantity b from the input a.*

Definition 4.2 (Coordinate friendly operator)**.** *We say that an operator $\mathcal{T} : \mathbb{H} \to \mathbb{H}$ is coordinate friendly(CF) operator if, we can find a $\mathcal{M}(x)$ and $\forall i, x$ and $x^+ = (x_1, \cdots (\mathcal{T}x)_i, \cdots, x_m))$, the following holds,*

$$\mathfrak{M}[\{x, \mathcal{M}(x)\} \to \{x, \mathcal{M}(x^+)\}] = \mathcal{O}(\frac{1}{m}\mathfrak{M}[x \to \mathcal{T}x]),$$

where $\mathcal{M}(x)$ is some quantity maintained in the memory to facilitate each coordinate or block update and refreshed to $\mathcal{M}(x^+)$.

For coordinate friendly operator, updating part of variables means that it only take part of the total time. In the other word, when we double the number of threads, the time for each iteration could be halved. Thus we can believe the algorithm will speed up the algorithm when the operator of the algorithm is coordinate friendly.

ARock is an algorithmic framework to find a fixed-point to a nonexpansive operator, that is to find x^* satisfying $x^* = \mathcal{T}x^*$. The framework is

Algorithm 1 Arock

```
Require: $x^0 \in \mathcal{H}$, $k_max$, a distribution $(p_1, \cdots, p_m) > 0$ with sum equal to 1;
1: global counter $k \leftarrow 0$;
2: while $k \le k_{max}$, each thread asynchronously and continuously do
3:     select an index $i_k \in \{1, \cdots, m\}$ with Prob$(i_k = i) = p_i$;
4:     perform an update to $x_{i_k} \leftarrow x_{i_k} + \eta_k((I - \mathcal{T})x)_{i_k}$;
5:     $k \leftarrow k + 1$;
6: end while
```

4.2 Douglas-Rachford Splitting

If we compute $A^T(AA^T)^{-1}b$ and $A^T(AA^T)^{-1}A$ at first, the cost of 3.1 satisfies $\mathfrak{M}[y^k \to y^{k+1}] = \mathcal{O}(m^2)$. The cost for updating one coordinate is $\mathfrak{M}[y \to y^+] = \mathcal{O}(m)$. However, if we don't pre-compute this matrix and vector, we need to solve a linear equation each time. The costs of updating whole variables and updating one coordinate are both $\mathcal{O}(m^3)$, which has the same order as solving linear system. Hence, we should pre-compute $A^T(AA^T)^{-1}b$ and $A^T(AA^T)^{-1}A$ before the iteration and then the algorithm is CF.

The pseudocode of async-parallel updating of DRS is as 2.

The same as the DRS for standard form, if we pre-compute $(I+\lambda\tilde{A})^{-1}$, the DRS for self-dual embedding form is also coordinate friendly. As the former 2, we can write the pseudocode as 3.

Algorithm 2 Douglas-Rachford Splitting for linear programming

```
Require: $c, A, b$
Ensure: optimal x
 1: Initialize shared $y \leftarrow 0$, global counter $k \leftarrow 0$;
 2: Initialize a distribution $(p_1, \cdots, p_m) > 0$ with sum $= 1$;
 3: Pre-compute and cache $\tilde{A} = A^T(AA^T)^{-1}A$, $\tilde{b} = A^T(AA^T)^{-1}b$;
 4: Spawn a set of parallel threads;
 5: Initialize $0 < \eta < \eta_{max}$;
 6: while $k \le k_{max}$, each thread asynchronously and continuously do
 7:     select an index $i_k \in \{1, \cdots, m\}$;
 8:     $y' \leftarrow y$ ($y'$ is a local copy of $y$);
 9:     compute $x \leftarrow (y' - \lambda c)^+$;
10:     compute $\Delta y_{i_k} \leftarrow \frac{\eta}{mp_{i_k}}(\tilde{A}_{i_k,:}(y' - 2x) + \tilde{b}_{i_k} + x_{i_k} - y'_{i_k})$;
11:     update $y_{i_k} \leftarrow y_{i_k} + \Delta y_{i_k}$;
12:     $k \leftarrow k + 1$;
13: end while
14: return $(y - \lambda c)^+$;
```

Algorithm 3 DRS for self-dual embedding form

```
Require: $c, A, b$
Ensure: optimal x
 1: Initialize shared $y \leftarrow (1, \cdots, 1)$, global counter $k \leftarrow 0$;
 2: Initialize a distribution $(p_1, \cdots, p_{m+2p+2})$;
 3: Pre-compute and cache $\tilde{A}$ and $(I + \lambda\tilde{A})^{-1}$;
 4: Initialize $0 < \eta < \eta_{max}$
 5: Spawn a set of parallel thread;
 6: while $k \le k_{max}$, each thread asynchronously and continuously do
 7:     select an index $i_k \in \{1, \cdots, m + 2p + 2\}$;
 8:     $y' \leftarrow y$;
 9:     update $y_{i_k} \leftarrow y_{i_k} + \eta((I + \lambda\tilde{A})^{-1}_{i_k,:}(2\,\mathrm{proj}_{\mathcal{C}}(y') - y') - \mathrm{proj}_{\mathcal{C}}(y')_{i_k})$;
10:     $k \leftarrow k + 1$;
11: end while
12: return $y_{p+1:p+m}$;
```

4.3 Primal-Dual Splitting

We can update s and x in Jacobi-style, so that there will be no over-lapping blocks. To let the update be coordinate friendly, we need to cache Ax during the iteration. The whole update will cost $\mathfrak{M}[\{(s\ x)^{(k)}\} \to \{(s\ x)^{(k+1)}\}] = \mathcal{O}(mp)$ each iteration, while updating one coordinate of s is $\mathfrak{M}[\{(s\ x), Ax\} \to \{(s^+\ x), Ax\}] = \mathcal{O}(1)$ and updating one coordiante of x is $\mathfrak{M}[\{(s\ x), Ax\} \to \{(s\ x^+), Ax^+\}] = \mathcal{O}(p)$. Thus it is coordinate friendly.

The pseudocode of this is 4.

However, if we update them in this way, the s^{k+1} will be computed twice

Algorithm 4 Primal-Dual Splitting for linear programming

```
Require: $c, A, b$
Ensure: optimal x
1: Initialize shared $s \leftarrow 0$, $x \leftarrow 0$, $Ax \leftarrow 0$, global counter $k \leftarrow 0$;
2: Initialize a distribution $(0_1, \cdots, q_{m+p})$ with sum $= 1$;
3: Spawn a set of parallel thread;
4: while $k \le k_{max}$, $\forall$thread asynchronously and continuously do
5:     select an index $i_k \in \{1, \cdots, m+p\}$;
6:     if $k \le p$ then
7:         $s_{i_k} \leftarrow s_{i_k} + \frac{\delta_k}{(m+p)q_{i_k}}\gamma(Ax)_{i_k} - \frac{\delta_k}{(m+p)p_{i_k}}\gamma b_{i_k}$;
8:     else
9:         $s' \leftarrow s$;$Ax' \leftarrow Ax$;
10:        $y \leftarrow x_{i_k-p} - \eta(c_{i_k-p} + A^T_{i_k-p,:}(s' + 2\gamma(Ax') - 2\gamma b))$;
11:        $\Delta x_{i_k-p} = \frac{\delta_k}{(m+p)q_{i_k}} y^+ - x_{i_k-p}$;
12:        $x_{i_k-p} \leftarrow x_{i_k-p} + \Delta x_{i_k-p}$; $Ax \leftarrow Ax + \Delta x_{i_k-p} A_{:,i_k-p}$;
13:    end if
14:    $k \leftarrow k+1$;
15: end while
16: return x
```

in each iteration , which may be a kind of waste. Considering that A^T is sparse in most applications, we can try the overlapping-block coordinate updates mentioned in [2].

The pseudocode is as 5.

Algorithm 5 PD overlapping-block

```
Require: $c, A, b$
Ensure: optimal x
1: Initialize shared $s \leftarrow 0$, $x \leftarrow 0$, $Ax \leftarrow 0$, global counter $k \leftarrow 0$;
2: Initialize a distribution $(0_1, \cdots, q_m)$ with sum $= 1$;
3: Initialize $\rho_{ij}$ with $\sum_i \rho_{ij} = 1\ \forall j$;
4: Spawn a set of parallel thread;
5: while $k \le k_{max}$, $\forall$thread asynchronously and continuously do
6:     select an index $i_k \in \{1, \cdots, m\}$;
7:     $s' \leftarrow s$, $Ax' \leftarrow Ax$,$x'_{i_k} \leftarrow x_{i_k}$;
8:     $\tilde{s}_j \leftarrow s'_j + \gamma(Ax')_j - \gamma b_j \quad j \in \mathbb{J}(i_k)$;
9:     $y_{i_k} = (x'_{i_k} - \eta(c_{i_k} + \sum_{j \in \mathbb{J}(i_k)} A^T_{i_k,j}(2\tilde{s}_j - s'_j)))^+$
10:    $\Delta x_{i_k} = \frac{\delta_k}{mq_{i_k}}(y_{i_k} - x'_{i_k})$;
11:    update $x_{i_k} \leftarrow x_{i+_k} + \Delta x_{i_k}$; $Ax \leftarrow Ax + \Delta x_{i_k} A_{:,i_k}$; $s_j \leftarrow s_j + \rho_{ij}(\tilde{s}_j -$
       $s_j) \quad j \in \mathbb{J}(i)$;
12:    $k \leftarrow k+1$
13: end while
14: return x
```

4.4 Gradient Projection

The cost of the whole update is $\mathcal{O}((m+2p)^2)$, while update one coordinate is $\mathcal{O}(m+2p)$. Thus this is coordinate friendly.

To update $\tilde{x}$ asynchronously, we can write the pseudocode as 6.

Algorithm 6 Gradient Projection for self-dual embedding form

```
Require: $c, A, b$
Ensure: optimal x
 1: Initialize shared $z \leftarrow (1, \cdots, 1)$, global counter $k \leftarrow 0$;
 2: Initialize a distribution $(p_1, \cdots, p_{m+2p+2})$;
 3: Pre-compute and cache $\tilde{A}$;
 4: Spawn a set of parallel thread;
 5: Pre-compute $\rho(\tilde{A})$, by power method;
 6: $\lambda \leftarrow 1/\rho(\tilde{A})$;
 7: while $k \le k_{max}$, each thread asynchronously and continuously do
 8:     select an index $i_k \in \{1, \cdots, m+2p+2\}$;
 9:     $z' \leftarrow z$;
10:     update $z_{i_k} \leftarrow (z_{i_k} - \lambda \tilde{A}_{i_k,:} z')^+$;
11:     $k \leftarrow k+1$;
12: end while
13: return $z_{p+1:p+m}$;
```

4.5 FISTA

After substituting y in 3.7 and 3.8, the iteration can be written as

$$\begin{pmatrix} \tilde{x}^k \\ \tilde{x}^{k-1} \end{pmatrix} = \begin{pmatrix} \mathrm{proj}_{\mathcal{C}} & \\ & I \end{pmatrix} \begin{pmatrix} \frac{2k-1}{k+1}(I - t_k\tilde{A}) & -\frac{k-2}{k+1}(I - t_k\tilde{A}) \\ I & \end{pmatrix} \begin{pmatrix} \tilde{x}^{k-1} \\ \tilde{x}^{k-2} \end{pmatrix}. \quad (4.1)$$

Obviously this is coordinate friendly when we precompute $\tilde{A}$. One of the possible algorithm is as 7.

Algorithm 7 Async-FISTA

```
Require: $c, A, b$
Ensure: $\tilde{x}$
 1: compute $\tilde{A}$
 2: compute $L$ by power method
 3: Initialize $x0 \leftarrow 0$; $x1 \leftarrow 0$;
 4: while $k < k_{max}$ do
 5:     select $i_k \in \{1, \cdots, 2(m+2p+2)\}$
 6:     if $i_k \leq m+2p+2$ then
 7:         $x1_{i_k} \leftarrow (\frac{2k-1}{k+1}(x1_{i_k} - t_k\tilde{A}_{i_k,:}x1) - \frac{k-2}{k+1}(x0_{i_k} - t_k\tilde{A}_{i_k,:}x0))^+$
 8:     end if
 9:     if $i_k > m+2p+2$ then
10:         $x0_{i_k-m-2p-2} \leftarrow x1_{i_k-m-2p-2}$
11:     end if
12:     $k \leftarrow k+1$
13: end while
14: return x0
```

5 Convergence

In [3], the authors provide theorems for the convergence of ARock. Here is the convergence theorem.

Definition 5.1. *An operator $T : \mathcal{H} \to \mathcal{H}$ is $c-$Lipschitz, satisfies $\|Tx - Ty\| \leq c\|x - y\|$, $\forall x, y \in' mathcal H$. If $c \leq 1$, T is nonexpansive.*

Theorem 5.2. *Let (x^k) be the sequence generated by ARock with $\eta_k \in [\eta_{min}, \frac{cmp_{min}}{2\tau\sqrt{p_{min}}+1}]$ for any $\eta_{min} > 0$ and $0 < c < 1$, then the sequence weakly and almost surely converges to an $\operatorname{Fix} T$ -valued random variable.*

With the theorem above, what we need to do is to check whether the operator in each algorithm is nonexpansive or not.

5.1 Douglas-Rachford Splitting

We can define the operator in 3.1 as

$$\begin{aligned} Ty &= (y - \lambda c)^+ + A^T(AA^T)^{-1}(b + A(y - 2(y - \lambda c)^+) \\ &= (I - A^T(AA^T)^{-1}A)(y - \lambda c)^+ + A^T(AA^T)^{-1}A(y - \lambda c) + const. \end{aligned}$$

Then we get

$$\begin{aligned} Ty_1 - Ty_2 =& (I - A^T(AA^T)^{-1}A)((y_1 - \lambda c)^+ - (y_2 - \lambda c)^+) \\ &+ A^T(AA^T)^{-1}A(y_1 - \lambda c - y_2 + \lambda c). \end{aligned}$$

Without loss of generality, we can set $c = 0$. Denote $A = P(D\ 0)Q$ is the SVD decomposition. We get

$$RHS = Q\,\mathrm{diag}(0, I_{m-p})Q^T(y_1^+ - y_2^+) + Q\,\mathrm{diag}(I_p, 0)Q^T(y_1^- - y_2^-),$$

where $x = x^+ + x^-$ is the positive part and negative part of x. Since $\forall x, y \in \mathbb{R}$, $(x-y)^2 \geq (x^+ - y^+)^2 + (x^- - y^-)^2$, we have

$$\begin{aligned}\|Ty_1 - Ty_2\|^2 \leq & \|Q\,\mathrm{diag}(0, I_{m-p})Q^T(y_1^+ - y_2^+)\|^2 + \|Q\,\mathrm{diag}(I_p, 0)Q^T(y_1^- - y_2^-)\|^2 \\ \leq & \|(y_1^+ - y_2^+)\|^2 + \|(y_1^- - y_2^-)\|^2 \\ \leq & \|y_1 - y_2\|^2,\end{aligned}$$

Thus the operator is nonexpansive and the algorithms converges.

5.2 Primal-Dual Splitting

We can define the operator in 3.4 and 3.5 as

$$T\begin{pmatrix} s^{k+1} \\ x^{k+1} \end{pmatrix} = \begin{pmatrix} I & \\ & (\cdot)^+ \end{pmatrix}$$
$$\left(\begin{pmatrix} I & \gamma A \\ -\eta A^T & I - 2\eta\gamma A^T A \end{pmatrix}\begin{pmatrix} s^k \\ x^k \end{pmatrix} + \begin{pmatrix} -\gamma b \\ -\eta c + 2\eta\gamma A^T b \end{pmatrix}\right).$$

Based on [2], this operator is nonexpansive in space $(\mathbb{R}^{m+p}, <\cdot>_M)$, where

$$M = \begin{pmatrix} \frac{1}{\eta} I & -A \\ -A^T & \frac{1}{\gamma} I \end{pmatrix},$$

and the iteration converges.

5.3 Gradient Projection

We can define the operator in 3.6 as

$$Tx = ((I - \lambda\tilde{A})\tilde{x})^+.$$

When $1/\lambda$ is greater than the radius of $\tilde{A}$, the linear transformation $I - \lambda\tilde{A}$ is nonexpansive. Since the projection operator is nonexpansive, the operator T is nonexpansive and the algorithm converges.

5.4 FISTA

The linear transformation in 4.1 is not nonexpansive, so that this algorithm could not be proved by ARock. The proof will be left as further work.

6 Numerical Results

6.1 Used Problems

Considering that DRS need to compute matrix inverse, we need to use both dense problem and sparse problem.

One of the problems is random dense problem. We generate the data used in LP randomly. Each element of A is i.i.d. uniformly distributed random variables valued in $[-0.5, 0.5]$ or $[0, 1.0]$. At first we choose $[0, 1.0]$, but the condition number of matrix is so large that our algorithms, especially the algorithm 4, don't perform as expected, but when we choose $[-0.5, 0.5]$ as the range, the condition numbers are around 5 for different size of problem. We generate a variable *temp* which is uniformly distributed random variables valued in $[0, 1]^m$. The variable $b = A*temp+\omega$, where ω is uniformly distributed random variables valued in $[-0.5, 0.5]^m$, and variable c is uniformly distributed random variables valued in $[0, 1]^m$.

The other problem we used is transportation problem. We have m factories and n shops. Factory i can send s_i goods to shops and shop j demands d_j. Assume that the cost of sending x_{ij} goods from factory i to shop j is $c_{ij}x_{ij}$. The math form of transportation problem is

$$\begin{aligned} \min_x \quad & \sum_i \sum_j c_{ij}x_{ij} \\ s.t. \quad & \sum_j x_{ij} \leq s_i,\ i = 1, \cdots, m \\ & \sum_i x_{ij} \geq d_j,\ j = 1, \cdots, n \\ & x_{ij} \geq 0. \end{aligned}$$

For convenience, we can assume $\forall i_1, i_2\ s_{i_1} = s_{i_2}$, $\forall j_1, j_2\ d_{j_1} = d_{j_2}$ and $\sum s_i = \sum d_j$.

6.2 Stopping condition

Denote x as primal variable, s as dual variable and z as Kuhn-Tucker variable.

When we increase dimensions in problem, the norm of b and c increase relatively. But this may cause some numerical difficulties for our algorithms, if we use $\|Ax - b\|_2 \leq \epsilon$ as primal-feasible condition and $\|(c + A^T s)^{-1}\|_2 \leq \epsilon$ as dual-feasible condition, because the number of iterations is sensitive with the stopping conditions. So we should look for other ones.

We use those two conditions the same as Mosek, [1]. That is

$$\begin{aligned} \|Ax - b\|_\infty &\leq \epsilon(1 + \|b\|_\infty) \\ \|A^T s - c + z\|_\infty &\leq \epsilon(1 + \|c\|_\infty). \end{aligned}$$

The condition we used for the dual gap is

$$|c^T x - b^T s| \leq \epsilon(1 + |c^T x| + |b^T s|).$$

6.3 Stopping check

We use three ways to pause the iteration to check whether the algorithm has reached the stopping condition.

- We can pause the iteration when each thread runs n_{check} times. However, when the time cost of each iteration varies, this method will waste much time for fast thread to wait for other threads.
- Pause when global counter $iter \mod n_{check} = 0$. In experiment I find that many threads may change the global counter at the same time. Thus there is a possibility that many thread may not pause until they run another n_{check} times because the $iter$ has changed before deciding whether to stop.
- We give $thread_i$ a variable m_i to counter how many times the thread have been paused. The thread pause when $iter > (m_i + 1) * n_{check}$. This will avoid the disadvantages of former methods, but it may cost time to update m_i.

We choose the third way.

6.4 Douglas-Rachford Splitting

We test DRS on random dense problem first. We randomly generate a problem with 4000 variables and 2000 equality constraints. The number of blocks is the same as that of threads. We set the step size $\eta = 0.9$ and $\lambda = 1.0$ and cyclically choose the updating blocks. We set $n_{check} = 10 * n_{threads}$. The performance of different threads are in table 1. Next we test the performance of our algorithms among different scales of the size of problem. We set the step size $\eta = 0.5$ and $\lambda = 1.0$ and cyclically choose the updating blocks. We pause the iteration the same as before. The performance of different sizes of problems are in table 2.

Table 1: DRS on random problem of size 2000 times 4000

threads	1	2	4	8	16	32
time	136.66(sec)	73.6144	46.6535	35.3899	34.0718	38.9778
time without check	119.285	59.8424	35.7442	25.8883	24.6122	23.6369
iter	8552	19203	39805	81369	167857	327073

From table 1, we can find that when we double the number of threads, we nearly halve the time. But when number of threads is too large, the cost of parallel overhead may become larger so that the running time couldn't become small when we increase number of threads.

Table 2: DRS performance of different scales of problem

	100x200	200x400	400x800	800x1600	1600x3200	2000x4000
mosek	0.04(sec)	0.14	0.51	2.41	15.48	29
sedumi	0.09163	0.2198	0.6434	4.472	35.34	67.5
SDPT3	0.19	0.98	7.19	57.24		1518.79
DRS-8	0.111106	0.171871	1.90669	2.671313	20.30293	38.82731

Moreover, the cost of parallel overhead become significant when we double the threads from 8 to 16, no matter what is the size of problem. The time cost can be seen in table 3.

Table 3:

threads	100x200	400x800	800x1600	1600x3200	2000x4000
pre-compute	0.010528(sec)	0.04386	0.679083	2.40473	3.43741
4	0.084444	2.18612	3.18727	31.25	46.6535
8	0.100578	1.86283	1.99223	17.8982	35.3899
16	0.123647	1.50451	1.6561	18.9273	34.0718
DRS-32	0.263122	2.31558	1.34422	22.9678	38.9778

Then we test DRS on transportation problem. When there are 8 factories and 640 shops, we set step size is 0.9 and other parameters be the same as before. The result is in table 4. From this we can find that for sparse matrix, we need to use other methods to speed up the algorithm, at least using sparse structure to store the matrix.

Table 4:

mosek	sedumi	SDPT3	DRS precompute	DRS 8threads
0.03(sec)	0.33252	0.14	2.11423	36.6132

6.5 Primal-Dual Splitting

6.5.1 Precondition for Primal-Dual Splitting

We use the precondition technique in [4]. That is we can replace the parameter η and γ by diagonal matrix $T = \operatorname{diag}(\eta_1, \cdots, \eta_m)$ and $\Sigma = \operatorname{diag}(\gamma_1, \cdots, \gamma_p)$, where

$$\eta_i = \frac{1}{\sum_{j=1}^{p} |A_{ij}|^{\alpha}} \qquad \gamma_j = \frac{1}{\sum_{i=1}^{m} |A_{ij}|^{2-\alpha}}$$

6.5.2 Parameters and Results

We run algorithm 4 on random dense problem with 2000 variables and 1000 constraints. We separate primal and dual variables into 4 blocks each. The parameter for precondition is 1.0 and step size is 0.5. We use 4 threads to update and the maximum number of iteration of each thread is 1000000. But the algorithm stopped before reach the stopping condition.

6.6 Other things may need to be mentioned

We use Intel® Math Kernel Library(Intel®MKL) as the library.

7 Further Work

- We need give convergence proof for FISTA.
- Use more problems for test.

References

[1] Mosek. Mosek toolbox linear programming. https://docs.mosek.com/8.1/matlabfusion/solving-linear.html.

[2] Zhimin Peng, Tianyu Wu, Yangyang Xu, Ming Yan, and Wotao Yin. Coordinate friendly structures, algorithms and applications. *arXiv preprint arXiv:1601.00863*, 2016.

[3] Zhimin Peng, Yangyang Xu, Ming Yan, and Wotao Yin. Arock: an algorithmic framework for asynchronous parallel coordinate updates. *Mathematics*, 38(5), 2016.

[4] Thomas Pock and Antonin Chambolle. Diagonal preconditioning for first order primal-dual algorithms in convex optimization. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 1762–1769. IEEE, 2011.

[5] R J Vanderbei. Linear programming: Foundations and extensions. *Journal of the Operational Research Society*, 49(1):94–94, 1998.

[6] Wikipeida. Linear programming. https://en.wikipedia.org/wiki/Linear_programming.

[7] Xiaojie Xu, Pi-Fang Hung, and Yinyu Ye. A simplified homogeneous and self-dual linear programming algorithm and its implementation. *Annals of Operations Research*, 62(1):151–171, 1996.