A distributed block coordinate descent method for training l_1 regularized linear classifiers

Dhruv Mahajan
Microsoft Research
Bangalore, India
dhrumaha@microsoft.com

S. Sathiya Keerthi
CISL, Microsoft Corporation
Mountain View, CA
keerthi@microsoft.com

S. Sundararajan
Microsoft Research
Bangalore, India
ssrajan@microsoft.com

ABSTRACT

Distributed training of l_1 regularized classifiers has received great attention recently. Existing methods approach this problem by taking steps obtained from approximating the objective by a quadratic approximation that is decoupled at the individual variable level. These methods are designed for multicore and MPI platforms where communication costs are low. They are inefficient on systems such as Hadoop running on a cluster of commodity machines where communication costs are substantial. In this paper we design a distributed algorithm for l_1 regularization that is much better suited for such systems than existing algorithms. A careful cost analysis is used to support these points. The main idea of our algorithm is to do block optimization of many variables within each computing node; this increases the computational cost per step that is commensurate with the communication cost, and decreases the number of outer iterations, thus yielding a faster overall method. Distributed Gauss-Seidel and greedy schemes are used for choosing variables to update in each step. We establish global convergence theory for our algorithm, including Q-linear rate of convergence. Experiments on two benchmark problems show our method to be much faster than existing methods.

Categories and Subject Descriptors

I.5.2 [**Pattern Recognition**]: Design Methodology—*Classifier design and evaluation*

General Terms

Algorithms, Performance, Experimentation

1. INTRODUCTION

The design of sparse linear classifiers using l_1 regularization is an important problem that has received great attention in recent years. This is due to its value in scenarios where the number of features is large and the classifier representation needs to be kept compact. With big data becoming common nowadays, distributed storage of data over a cluster of commodity machines becomes necessary. Thus, fast training of l_1 regularized classifiers over distributed data is an important problem.

A number of algorithms have been recently proposed for parallel and distributed training of l_1 regularized classifiers. Most of these algorithms are designed for multicore and MPI platforms in which data communication costs are negligible. These platforms are usually equipped with only a small number of computing nodes. Distributed systems, e.g., Hadoop running on a cluster of commodity machines, are better for employing a large number of nodes and hence, for inexpensive handling of big data. However, in such systems, communication costs are high; current methods for l_1 regularization are not optimally designed for such systems. *In this paper we develop a distributed block coordinate descent (DBCD) method that is efficient on distributed platforms in which communication costs are high.*

Most methods (including the current ones and the one we propose) fit into a generic algorithm format that we describe in Section 2. This gives a clear view of existing methods and allows us to motivate the new method. We give full details of the DBCD method in Section 3. A fuller discussion of existing methods in relation to our method is given in Section 4. The analysis of computation and communication costs in Section 5 sheds further insight. Experiments comparing our method with several existing methods on a few large scale datasets are given in Section 6. These experiments strongly demonstrate the efficiency of one version of our method that chooses update variables greedily. We conclude the paper in Section 7. The appendix gives a proof of the convergence result stated in Section 3.

2. A GENERIC ALGORITHM

The generic algorithm format allows us to explain the roles of key elements of the methods and point out how new choices for various steps can lead to a better design. Before describing it, we first formulate the l_1 regularization problem.

Problem formulation. Let w be the weight vector with m variables, w^j, $j = 1, \ldots, m$,[1] and $x_i \in R^m$ denote the i-th example. A linear classifier produces the output $y^i = w^T x_i$. For binary class label $t^i \in \{1, -1\}$, the loss is given by $\ell(y^i; t^i)$. We will assume that $\ell \in \mathcal{C}^1$, the class of continuously differentiable functions, and that ℓ' is non-negative and Lipschitz continuous[2]. Loss functions such as least squares loss, logistic loss, SVM squared hinge loss and Huber loss satisfy these assumptions. The total loss function, $f : R^m \to R$ is $f(w) = \frac{1}{n}\sum_i \ell(y^i; t^i)$. Let u be the l_1 regularizer given by $u(w) = \lambda \sum_j |w^j|$, where $\lambda > 0$ is the regularization constant. Our aim is to solve the problem

$$\min_{w \in R^m} F(w) = f(w) + u(w). \tag{1}$$

[1] Vector components are denoted by superscripts. We also use superscript for iteration number, but the distinction will be clear from the context.

[2] A function h is Lipschitz continuous if there exists a (Lipschitz) constant $L \geq 0$ such that $\|h(a) - h(b)\| \leq L\|a - b\| \;\; \forall\, a, b$.

Let $g = \nabla f$. The optimality conditions for (1) are:

$$\forall j: \quad g^j + \lambda \operatorname{sign}(w^j) = 0 \text{ if } |w^j| > 0; \quad |g^j| \le \lambda \text{ if } \; w^j = 0. \tag{2}$$

Generic algorithm. Let there be n training examples and let X denote the $n \times m$ data matrix, whose i-th row is x_i^T. For problems with a large number of features, it is natural to randomly partition the columns of X and place the parts in P computing nodes. Let $\{B_p\}_{p=1}^P$ denote this partition of $\mathcal{M} = \{1, \ldots, m\}$. *We will assume that this feature partitioning is given and that all algorithms operate within that constraint.* The variables associated with a particular partition get placed in one node. Given a subset of variables S, let X_S be the submatrix of X containing the columns corresponding to S. For a vector $z \in R^m$, z_S will denote the vector containing the components of z corresponding to S.

Algorithm 1 gives the generic algorithm. Items such as B_p, S_p^t, w_{B_p}, $d_{B_p}^t$, X_{B_p} stay local in node p and do not need to be communicated. Step (d) can be carried out using an *All Reduce* operation [1] over the nodes and then y becomes available in all the nodes. The gradient sub-vector $g_{B_p}^t$ can then be computed locally as $g_{B_p}^t = X_{B_p}^T b$ where $b \in R^n$ is a vector with $\{\ell'(y_i)\}$ as its components.

Algorithm 1: A generic distributed algorithm

```
Choose $w^0$ and compute $y = Xw^0$;
for $t = 0, 1 \ldots$ do
    for $p = 1, \ldots, P$ do
        (a) Select a subset of variables, $S_p^t \subset B_p$;
        (b) Form $f_p^t(w_{B_p})$, an approximation of $f$ and
        solve (exactly or approximately):
            $\min f_p^t(w_{B_p}) \;\; \text{s.t.} \;\; w_j = w_j^t \; \forall\, j \notin B_p \setminus S_p^t \quad (3)$
        to get $\bar{w}_{B_p}^t$ and set direction: $d_{B_p}^t = \bar{w}_{B_p}^t - w_{B_p}^t$;
        (c) Choose $\alpha^t$ and update:
        $w_{B_p}^{t+1} \leftarrow w_{B_p}^t + \alpha^t d_{B_p}^t$;
    end
    (d) Update $y \leftarrow y + \alpha \sum_p X_{B_p} d_{B_p}^t$;
    (e) Terminate if optimality conditions hold;
end
```

Step (a) - variable sampling. Some choices are: **(a.1)** random selection [4, 18]; **(a.2)** random cyclic: over a set of consecutive iterations (t) all variables are touched once [2]; **(a.3)** greedy: always choose a set of variables that, in some sense violate (2) the most at the current iterate [16, 7]; and, **(a.4)** greedy selection using the Gauss-Southwell rule [24, 27].

Step (b) - function approximation. Most methods choose a quadratic approximation that is decoupled at the individual variable level:

$$f_p^t(w_{B_p}^t) = \sum_{j \in B_p} g^j(w^t)(w^j - (w^t)^j) + \frac{L^j}{2}(w^j - (w^t)^j)^2 \tag{4}$$

The main advantages of (4) are its simplicity and closed-form minimization. Choices for L^j that have been tried are: **(b.1)** L^j = a Lipschitz constant for g^j [4, 16]; **(b.2)** L^j = a large enough bound on the Lipschitz constant for g^j to suit the sampling in step (a) [18]; **(b.3)** adaptive adjustment of L^j [7]; and **(b.4)** $L^j = H_{jj}^t$, the j-th diagonal term of the Hessian at w^t [2].

Step (c) - step size. The choices are: **(c.1)** always fix $\alpha^t = 1$ [4, 18, 16]; **(c.2)** use stochastic approximation to choose $\{\alpha^t\}$ so that $\sum_t (\alpha^t)^2 < \infty$ and $\sum_t |\alpha^t| = \infty$ [7]; and **(c.3)** choose α by line search that is directly tied to the optimization of F in (1) [2].

To understand the role of the various choices better, let us focus on the use of (4) for f_p^t. Algorithm 1 can diverge due to one of the following reasons: (i) choosing too many variables ($|S_p^t|$ large) for parallel updating in step (a); (ii) choosing small values for the proximal coefficient L^j in step (b); and (iii) not controlling α^t to be sufficiently small in step (c). Different methods control against these by making suitable choices in the steps.

The choice made for step (c) gives a nice delineation of methods. With **(c.1)**, one has to do a suitable mix of large enough L^j and small enough $|S_p^t|$. **(c.2)** is better since the proper control of $\{\alpha^t\} \to 0$ takes care of convergence; however, for good practical performance, L^j and α^t need to be carefully adapted, which is usually messy. Choice **(c.3)** is good in many ways: it leads to monotone decrease in F; it is good theoretically and practically; and, it allows both, small L^j as well as large $|S_p^t|$ without hindering convergence. Except for [2, 24, 27] **(c3)** has been unused in other methods because it is considered as 'not-in-line' with a proper parallel approach as it requires F computations for several α^t values within one t. But truly, the slightly increased computation and communication costs is amply made up by a reduction in the number of iterations to reach sufficient optimality. So we go with the choice **(c.3)** in our method.

The choice of (4) for f_p^t in step (b) is pretty much unanimously used in all previous works. While this is fine for communication friendly platforms such as multicore and MPI, it is not the right choice when communication costs are high. Such a setting permits more per-node computation time, and there is much to be gained by using a more complex f_p^t. We propose the use of a function f_p^t that couples the variables in S_p^t. We also advocate an approximate solution of (3) (e.g., a few rounds of coordinate descent within each node) in order to control the computation time.

Crucial gains are also possible via resorting to the greedy choices, **(a.3)** and **(a.4)** for choosing S_p^t. On the other hand, with methods based on **(c.1)**, one has to be careful in using **(a.3)**: apart from difficulties in establishing convergence, practical performance can also be bad, as we show in Section 6.

Contributions. Following are our main contributions.

1. We make careful choices for the three steps, leading to the development of a distributed block coordinate descent (DBCD) method that is very efficient on distributed platforms with high communication cost.

2. We establish convergence theory for our method using the results of [24, 27]. It is worth noting the following: (a) though [24, 27] cover algorithms using quadratic approximations for the total loss, we use a simple trick to apply them to general nonlinear approximations, thus bringing more power to their results; and (b) even those works use only (4) in their implementations.

3. We provide a cost analysis that brings out the computation and communication costs of Algorithm 1 clearly for different methods.

4. We give an experimental evaluation that shows the strong performance of DBCD against key current methods in scenarios where communication cost is significant.

3. DBCD METHOD

The DBCD method that we propose fits into the general format of Algorithm 1. It is actually a class of algorithms that allows various possibilities for steps (a), (b) and (c). Below we lay out these possibilities and establish convergence theory for our method.

3.1 Function approximation.

Let us begin with step (b). We stress the main point that, unlike previous methods, we allow f_p^t to be non-quadratic and also to be a joint function of the variables in w_{B_p}. We first describe a general set of properties that f_p^t must satisfy, and then discuss specific instantiations that satisfy these properties.

P1. $f_p^t \in \mathcal{C}^1$; $g_p^t = \nabla f_p^t$ is Lipschitz continuous, with the Lipschitz constant uniformly bounded over all t; f_p^t is strongly convex (uniformly in t), i.e., $\exists\ \mu > 0$ such that $f_p^t - \frac{\mu}{2}\|w_{B_p}\|^2$ is convex; and, f_p^t is gradient consistent with f at $w^t_{B_p}$, i.e., $g_p^t(w^t_{B_p}) = g_{B_p}(w^t)$.

Gradient consistency is essential because it is the property that connects f_p^t to f and ensures that a solution of (3) will make $d^t_{B_p}$ a descent direction for F at $w^t_{B_p}$, thus paving the way for a decrease in F at step (c). Strong convexity is a technical requirement that is needed for establishing sufficient decrease in F in each step of Algorithm 1. Lipschitz continuity is another technical condition that is needed for ensuring boundedness of various quantities. Let us now discuss some good ways of choosing f_p^t. *For all these instantiations, a proximal term is added to get the strong convexity required by* **P1**.

Proximal-Jacobi. We can follow the classical Jacobi method in choosing f_p^t to be the restriction of f to $w_{S_p^t}$, with the remaining variables fixed at their values in w^t. Let $\bar{B}_p$ denote the complement of B_p, i.e., the set of variables associated with nodes other than p. Thus we set

$$f_p^t(w_{B_p}) = f(w_{B_p}, w^t_{\bar{B}_p}) + \frac{\mu}{2}\|w_{B_p} - w^t_{B_p}\|^2 \tag{5}$$

where $\mu > 0$. It is worth pointing out that, since each node p keeps a copy of the classifier output vector y, the computation of f_p^t and g_p^t due to changes in w_{B_p} can be locally computed in node p. Thus the solution of (3) is local to node p and so step (b) can be executed in parallel for all p.

Block GLMNET. GLMNET [26, 8] is a sequential coordinate descent method that has been demonstrated to be very promising for sequential solution of l_1 regularized problems with logistic loss. At each iteration, GLMNET minimizes the second order Taylor series of f at w^t, followed by line search along the direction generated by this minimizer. We can make a distributed version by choosing f_p^t to be the second order Taylor series approximation of $f(w_{B_p}, w^t_{\bar{B}_p})$ restricted to w_{B_p} while keeping $w_{\bar{B}_p}$ fixed at $w^t_{\bar{B}_p}$.

Block L-BFGS. One can keep a limited history of $w^t_{B_p}$ and $g^t_{B_p}$ and use an $L-BFGS$ approach to build a second order approximation of f in each iteration to form f_p^t.

Decoupled quadratic. Like in existing methods we can also form a quadratic approximation of f that decouples at the variable level. If the second order term is based on the diagonal elements of the Hessian at w^t, then the PCDN algorithm given in [2] can be viewed as a special case of our DBCD method. PCDN [2] is based on Gauss-Seidel variable selection. But it can also be used in combination with the distributed greedy scheme that we propose in Section 3.2.

Approximate stopping. In step (b) of Algorithm 1 we mentioned the possibility of approximately solving (3). This is irrelevant for previous methods which solve individual variable level quadratic optimization in closed form, but very relevant to our method. Here we propose an approximate relative stopping criterion and later, in Section 3.4, also prove convergence theory to support it.

Let ∂u^j be the set of sub-gradients of the regularizer term $u^j = \lambda|w^j|$, i.e.,

$$\partial u^j = [-\lambda, \lambda] \ \text{ if } w^j = 0; \ \ \lambda \operatorname{sign}(w^j) \ \text{ if } w^j \neq 0. \tag{6}$$

A point $\bar{w}^t_{B_p}$ is optimal for (3) if, at that point,

$$(g_p^t)^j + \xi^j = 0, \ \text{ for some } \ \xi^j \in \partial u^j \ \ \forall\, j \in S_p^t. \tag{7}$$

An approximate stopping condition can be derived by choosing a tolerance $\epsilon > 0$ and requiring that, for each $j \in S_p^t$ there exists $\xi^j \in \partial u^j$ such that

$$\delta^j = (g_p^t)^j + \xi^j, \ \ |\delta^j| \le \epsilon |(d^t_{B_p})^j| \ \ \forall\, j \in S_p^t \tag{8}$$

Method used for solving (3). Now (3) is an l_1 regularized problem restricted to $w_{S_p^t}$. It has to be solved within node p using a suitable sequential method. Going by the state of the art for sequential solution of such problems [25] we use the coordinate-descent method described in [25] for solving (3).

3.2 Variable selection

Let us now turn to step (a) of Algorithm 1. We propose two schemes for variable selection, i.e., choosing $S_p^t \subset B_p$.

Gauss-Seidel scheme. In this scheme, we form cycles - each cycle consists of a set of consecutive iterations - while making sure that every variable is touched once in each cycle. We implement a cycle as follows. Let τ denote the iteration where a cycle starts. Choose a positive integer T (T may change with each cycle). For each p, randomly partition B_p into T equal parts: $\{S_p^t\}_{t=\tau}^{\tau+T-1}$. Use these variable selections to do T iterations. *Henceforth, we refer to this scheme as the R-scheme.*

Distributed greedy scheme. This is a greedy scheme which is purely distributed and so more specific than the Gauss-Southwell schemes in [24].[3] In each iteration, our scheme chooses variables based on how badly (2) is violated for various j. For one j, an expression of this violation is as follows. Let g^t and H^t denote, respectively, the gradient and Hessian at w^t. Form the following one variable quadratic approximation:

$$q^j(w^j) = (g^t)^j(w^j - (w^t)^j) + \frac{1}{2}(H^t_{jj} + \nu)(w^j - (w^t)^j)^2 + \lambda|w^j| - \lambda|(w^t)^j| \tag{9}$$

where ν is a small positive constant. Let $\bar{q}^j$ denote the optimal objective function value obtained by minimizing $q^j(w^j)$ over all w^j. Since $q^j((w^t)^j) = 0$, clearly $\bar{q}^j \le 0$. The more negative $\bar{q}^j$ is, the better it is to choose j.

[3] Yet, our distributed greedy scheme can be shown to imply the Gauss-Southwell-q rule for a certain parameter setting. See Appendix for details.

Our distributed greedy scheme first chooses a positive integer k and then, in each node p, it chooses the top k variables from B_p according to smallness of $\bar{q}^j$, to form S_p^t. *Hereafter, we refer to this scheme as the S-scheme.*

3.3 Line search

Line search (step (c) of Algorithm 1) forms an important component for making good decrease in F at each iteration. For non-differentiable optimization, there are several ways of doing line search. For our context, Tseng and Yun [24] and Patriksson [14] give two good ways of doing line search based on Armijo backtracking rule. In this paper we use ideas from the former. Let β and σ be real parameters in the interval $(0, 1)$. (We use the standard choices, $\beta = 0.5$ and $\sigma = 0.01$.) We choose α^t to be the largest element of $\{\beta^k\}_{k=0,1,\ldots}$ satisfying

$$F(w^t + \alpha^t d^t) \leq F(w^t) + \alpha^t \sigma \Delta^t, \tag{10}$$

$$\Delta^t \overset{\text{def}}{=} (g^t)^T d^t + \lambda u(w^t + d^t) - \lambda u(w^t). \tag{11}$$

3.4 Convergence

We now establish convergence for the class of algorithmic choices discussed in Sections 3.1-3.3. To do this, we make direct use of the results of Tseng and Yun [24]. An interesting aspect of this use is that, whereas the results of Tseng and Yun [24] are stated only for f_p^t being quadratic, we employ a simple trick that lets us apply the results to our algorithm which involves non-quadratic approximations.

Apart from the conditions in **P1** (see Section 3.1) we need one other technical assumption.

P2. For any given t, w_{B_p} and $\hat{w}_{B_p}$, $\exists$ a positive definite matrix $\hat{H} \geq \mu I$ (note: $\hat{H}$ can depend on t, w_{B_p} and $\hat{w}_{B_p}$) such that

$$g_p^t(w_{B_p}) - g_p^t(\hat{w}_{B_p}) = \hat{H}(w_{B_p} - \hat{w}_{B_p}) \tag{12}$$

Except *Proximal-Jacobi*, the other instantiations of f_p^t mentioned in Section 3.1 are quadratic functions; for these, g_p^t is a linear function and so (12) holds trivially. Let us turn to *Proximal-Jacobi*. If $f_p^t \in \mathcal{C}^2$, the class of twice continuously differentiable functions, then **P2** follows directly from mean value theorem; note that, since $f_p^t - \frac{\mu}{2}\|w\|^2$ is convex, $H_p \geq \mu I$ at any point, where H_p is the Hessian of f_p^t. Thus **P2** easily holds for least squares loss and logistic loss. Now consider the SVM squared hinge loss, which is not in $\mathcal{C}^2$. **P2** holds for it because $g = \sum_i \ell'(y^i; t^i)x_i$ and, for any two real numbers z_1, z_2, $\ell'(z_1; t^i) - \ell'(z_2; t^i) = \kappa(z_1, z_2, t^i)(z_1 - z_2)$ where $0 \leq \kappa(z_1, z_2, t^i) \leq 1$.

The main convergence theorem can now be stated. Its proof is given in the Appendix.

Theorem 1. Suppose, in Algorithm 1: (i) step (a) is done via the Gauss-Seidel or distributed greedy schemes of Section 3.2; (ii) f_p^t in step (b) satisfies **P1** and **P2**; (iii) (8) is used to terminate (3) with $\epsilon = \mu/2$ (where μ is as in **P1**); and (iv) in step (c), α^t is chosen via Armijo backtracking of Section 3.3. Then Algorithm 1 is well defined and produces a sequence, $\{w^t\}$ such that any accumulation point of $\{w^t\}$ is a solution of (1). If, in addition, the total loss, f is strongly convex, then $\{F(w^t)\}$ converges Q-linearly and $\{w^t\}$ converges at least R-linearly.[4]

4. RELATED WORK

Our interest is mainly in parallel/distributed computing methods. There are many parallel algorithms targeting a single machine having multi-cores with shared memory (see [4], [19], [2], [16]). In contrast, there exist only a few efficient algorithms to solve (1) when the data is distributed ([20], [17]) and communication is an important aspect to consider. In this setting, the problem (1) can be solved in several ways depending on how the data is distributed across machines [16, 3]: (a) example (horizontal) split, (b) feature (vertical) split and (c) combined example and feature split (a block of examples/features per node). While methods such as distributed FISTA [16] or ADMM [3] are useful for (a), the block splitting method [13] is useful for (c). We are interested in (b), and the most relevant and important class of methods is *p*arallel/*d*istributed *c*oordinate *d*escent (PCD/DCD) methods, as abstracted in Algorithm 1. Table 1 compares these methods along various dimensions; most dimensions arise naturally from the steps of this algorithm, as explained in Section 2. (Due to space limitation, it is difficult to give a thorough discussion of these papers (see Table 1) from a theoretical convergence perspective on various assumptions and conditions under which results hold.) Two important points to note are: (a) except [20] and our method, none of the PCD methods target and sufficiently discuss distributed setting involving communication and (b) from a practical view point, it is difficult to ensure stability and get good speed-up with no line search and non-monotone methods. For example, methods such as [4, 18, 19, 16] that do not do line search are shown to have the monotone property only in expectation and that too under certain conditions. Furthermore, variable selection rules, proximal coefficients and other method-specific parameter settings play important roles in achieving monotone convergence and improved efficiency. As we show in Section 6, our method and the PCD Newton method [2] (see below for a discussion) enjoy robustness to various settings and come out as clear winners.

Generic Coordinate Descent Method [21, 22] Scherrer et al [21, 22] presented an abstract framework for coordinate descent methods (GenCD) suitable for parallel computing environments. Several coordinate descent algorithms such as stochastic coordinate descent (SCD) [23], *Shotgun* [4] and GRock [16] are covered by GenCD. GRock is a thread greedy algorithm [21] in which the variables are selected greedily using gradient information. One important issue is that algorithms such as Shotgun and GRock may not converge in practice due to their non-monotone nature with no line search; we faced convergence issues on some datasets in our experiments with GRock. Therefore, the practical utility of such algorithms is limited without ensuring necessary descent property through certain spectral radius conditions on the data matrix.

Flexible Parallel Algorithm (FPA) [7] This method has some similarities with our method in terms of the approximate function optimized at the nodes. Though [7] suggests several approximations, it uses only (4) in its final implementation. More importantly, FPA is a non-monotone method using a stochastic approximation step size rule. Tuning this step size rule along with the proximal parameter L_j to ensure convergence and speed-up is hard. Unlike our method, FPA's inner optimization stopping criterion is un-

[4] See chapter 9 of [12] for definitions of Q-linear and R-linear convergence.

verifiable (for e.g., with (5)); also, FPA does not address the communication cost issue.

Distributed Coordinate Descent Method [20] Richtárik and Takác [20] extended their initial multi-core parallel coordinate descent method [19] to the distributed setting. With no line search, their algorithm Hydra (*HY*bri*D* coo*R*din*A*te descent) has (expected) descent property only for certain sampling types of selecting variables and L^j values. One key issue is setting the right L^j values for good performance. Doing this accurately is a costly operation; on the other hand, inaccurate setting using cheaper computations (e.g., using number of non-zero elements as suggested in their work) results in slower convergence (see Section 6).

Parallel Coordinate Descent Newton (PCDN) [2] One key difference between other methods discussed above and our DBCD method is the use of line search. Note that the PCDN method can be seen as a special case of DBCD (see Section 3.1). In DBCD, we optimize per-node block variables jointly, and perform line search across the blocks of variables; as shown later in our experimental results, this has the advantage of reducing the number of outer iterations, and overall wall clock time due to reduced communication time (compared to PCDN).

Synchronized Parallel Algorithm [15] Patriksson [15] proposed a Jacobi type synchronous parallel algorithm with line search using a generic cost approximation (CA) framework for differentiable objective functions [14]. Its local linear rate of convergence results hold only for a class of strong monotone CA functions. Necora and Clipici [10] proposed an iterative algorithm where they solved (9) for all coordinates (block wise) in parallel, using coordinate Lipschitz constants for the gradient. The weights are updated in parallel as a weighted combination of previous weight and the incremental update obtained from solving (9). So it is only a minor variant of the GenCD class.

ADMM Methods Alternating direction method of multipliers is a generic and popular distributed computing method. This method can be used to solve (1) in different data splitting scenarios([3],[13]). Several variants of global convergence and rate of convergence (e.g., $O(\frac{1}{k})$) results exist under different weak/strong convexity assumptions on the two terms of the objective function [6],[5]. Recently, an accelerated version of ADMM [9] derived using the ideas of Nesterov's accelerated gradient method [11] has been proposed; this method has dual objective function convergence rate of $O(\frac{1}{k^2})$ under a strong convexity assumption. ADMM performance is quite good when the augmented Lagrangian parameter is set to the right value; however, getting a reasonably good value comes with computational cost. In Section 6 we evaluate our method and find it to be much faster.

5. COST ANALYSIS

In this section, we analyze the cost of Algorithm 1 for different methods. For ease of reference, we list the methods that we implemented and studied: (1)ADMM: accelerated alternating direction method of multipliers [9], (2) Hyd: Hydra [20], (3) GRock: *GR*eedy co*O*rdinate-bl*ock* [16], (4) FPA: Flexible Parallel Algorithm [7], (5) PCD: Parallel Coordinate Descent Newton method [2] and (6) DCD: Distributed block Coordinate Descent - our method. We use these abbreviations for ease of reference in the plots and discussion below. We considered two variable selection schemes (i.e., R-scheme and S-scheme) discussed in Section 3.2 for our method and PCD [2]. We refer these variants as DCD-R, DCD-S, PCD-R and PCD-S with variable selection rule indicated after hyphenation.

Let nz and $|S| = \sum_p |S_p^t|$ denote the number of non-zero entries in the data matrix X and number of variables updated in each iteration respectively. $\beta(\gg 1)$ is the relative computation to communication speed in a typical distributed system. Recall n, m and P denote the number of examples, features and nodes respectively. Table 2 gives cost expressions for different steps of the algorithm in one outer iteration. Here c_1, c_2, c_3, c_4 and c_5 are method dependent parameters. We briefly discuss different costs below.

Table 2: Cost of various steps of Algorithm 1.

Cost	Steps of Algorithm 1.			
	a	b	c	d
Comp.	$c_1\frac{nz}{P}$	$c_2\frac{nz}{P}\frac{\lvert S\rvert}{m}$	$c_3\lvert S\rvert + c_4 n$	$c_5\frac{nz}{P}\frac{\lvert S\rvert}{m}$
Comm.	-	-	$c_4\beta logP$	$\beta nlogP$

Table 3: Values of cost parameters for different methods. Note that q lies in the range: $1 \leq q \leq \frac{m}{|S|}$.

Method	c_1	c_2	c_3	c_4	c_5
Hyd	0	1	1	0	1
GRock	1	q	1	0	q
FPA	1	q	1	1	q
PCD	0	1	t_{ls}	t_{ls}	1
PCD-S	1	q	t_{ls}	t_{ls}	q
DCD-R	0	k	t_{ls}	t_{ls}	1
DCD-S	1	kq	t_{ls}	t_{ls}	q

Step a: Methods like our DCD-S, GRock, FPA and PCD-S need to calculate the gradient and model update to determine which variables to update. Hence, they need to go through the whole data once ($c_1 = 1$). On the other hand Hyd, PCD and DCD-R select variables randomly or in a cyclic order. As a result variable subset selection cost is negligible for them.

Step b: All the methods except DCD-S and DCD-R use the decoupled quadratic approximation (4). For DCD-R and DCD-S, an additional factor of k comes in c_2 since we do k inner cycles of CDN in each iteration. Hyd, PCD and DCD-R do a random or cyclic selection of variables. Hence, a factor of $\frac{|S|}{m}$ comes in the cost since only a subset $|S|$ of variables is updated in each iteration. However, methods that do selection of variables based on magnitude of update or expected objective function decrease (DCD-S, GRock, FPA and PCD-S) favour variables with low sparsity. As a result, c_2 for these methods has an additional factor q where $1 \leq q \leq \frac{m}{|S|}$.

Step c: For methods that do not use line-search, $c_3 = 1$ and $c_4 = 0$[5]. The overall cost is $|S|$ to update the variables. For methods like DCD-S, DCD-R, PCD and PCD-S that do line-search, $c_3 = c_4 = t_{ls}$ where t_{ls} is number of line search steps. For each line search step, we need to recompute the loss function which involves going over n examples once. Moreover, *AllReduce* step needs to be performed to sum over the distributed l_1 regularizer term. Hence, an additional $\beta log(P)$ cost is incurred to communicate the local

[5] For FPA, $c_4 = 1$ since objective function needs to be computed to automatically set the proximal term parameter

Table 1: Properties of Various Methods. + indicates distributed versions (like our method) implemented in our experiments. △ represents the class of smooth function optimization and all variables are updated in parallel; variants are possible.

Method	F descent	Feature limit/selection	Proximal term	Line search	Convergence	Convergence rate
SHOTGUN [4]	Non-monotone	Limited/Random	Lipschitz	None	In expectation	Sub-linear
PCD [18]	Non-monotone	No limits/Random	Maximal bound	None	In expectation	Linear
HYD [19]$^+$	Non-monotone	No limits /Random	Maximal bound	None	In expectation	Linear
GROCK [16]$^+$	Non-monotone	Limited/Greedy	Lipschitz	None	Deterministic	Sub-linear
PCDN [2]$^+$	Descent	No limits/Random	Hessian diag	Armijo	In expectation	Sub-linear
FPA [7]$^\triangle$	Non-monotone	No limits/Random	Lipschitz	None	Deterministic	None
PCD [10]$^\triangle$	Descent	Full	Lipschitz	None	Deterministic	Linear
SPA [15]$^\triangle$	Descent	Full	None	Armijo	Deterministic	Locally linear
DBCD	Descent	No limits/Random,Greedy	Free	Armijo	Deterministic	Locally linear

regularizer. As pointed out in [2], the line search steps typically increase with increasing number of nodes. Hence we expect this cost to increase with P.

Step d: This step involves computing and doing *AllReduce* on updated local predictions to get the global prediction vector for next iteration and is common for all the methods.

The analysis given above is only for C^P_{comp} and C^P_{comm}, the computation and communication costs in one iteration. If T^P is the number of iterations to reach a certain optimality tolerance, then the total cost of Algorithm 1 is: $C^P = T^P(C^P_{\text{comp}} + C^P_{\text{comm}})$. For P nodes, speed-up is given by C^1/C^P. To illustrate the ill-effects of communication cost, let us take the method of Richtárik and Takác [19]. For illustration, take the case of $|S| = P$, i.e., one variable is updated per node per iteration. For large P, $C^P \approx T^P C^P_{\text{comm}} = T^P\ \beta n \log P$; both β and n are large in the distributed setting. On the other hand, for $P = 1$, $C^P_{\text{comm}} = 0$ and $C^P = C^P_{\text{comp}} \approx \frac{nz}{m}$. Thus *speedup* $= \frac{T^1}{T^P}\frac{C^1}{C^P} = \frac{T^1}{T^P}\frac{\frac{nz}{m}}{\beta n \log P}$. Richtárik and Takác [19] show that T^1/T^P increases nicely with P. But, the term βn in the denominator of C^1/C^P has a severe detrimental effect. Unless a special distributed system with efficient communication is used, speed up has to necessarily suffer. When the training data is huge and so the data is forced to reside in distributed nodes, the right question to ask is not whether we get great speed up, but to ask which method is the fastest. In that sense, our analysis gives great insight: when C^P_{comm} dominates C^P_{comp}, reducing T^P is crucial; this is exactly what our method does. We see in Table 3 that, DCD-R and DCD-S have the maximum computational cost. On the other hand, communication cost is more or less the same for all the methods (except for few scalars in the line search step) and dominates the cost. As we will see in Section 6, by doing more computation, our methods reduce T^P substantially over the other methods while incurring a small computation overhead (relative to communication) per iteration.

6. EXPERIMENTAL EVALUATION

In this section, we present experimental results on real-world datasets. We compare our methods with several state of the art methods listed in the previous section. To the best of our knowledge, such a detailed study has not been done for parallel and distributed l_1 regularized solutions in terms of (a) accuracy and solution optimality performance, (b) variable selection schemes, (c) computation versus communication time and (d) solution sparsity. The results demonstrate the effectiveness of our methods in terms of total (computation + communication) time on both accuracy and objective function measures.

6.1 Experimental Setup

Datasets: We conducted our experiments on two popular benchmark datasets KDD and URL[6]. KDD has $n = 8.41M$, $m = 20.21M$ and $nz = 0.31B$. URL has $n = 2.00M$, $m = 3.23M$ and $nz = 0.22B$. These datasets have sufficiently interesting characteristics of having large number of examples and features such that (1) feature partitioning, (2) l_1 regularization and (3) communication are important.

Methods and Metrics: We evaluate the performance of all the methods using (a) Area Under Precision-Recall Curve (AUPRC) [1] and (b) Relative Function Value Difference (RFVD) as a function of time taken. RFVD is computed as $\log(\frac{F(w^t)-F^*}{F^*})$ where F^* is taken as the best value obtained across the methods after a long duration. We stopped the algorithm after a fixed number of outer iterations in each method. We also report per node computation time statistics and sparsity pattern behavior of all the methods.

Parameter Settings: We experimented with the λ values of $(123, 13.7, 4.6)\times 10^{-7}$ and $(727, 242, 9)\times 10^{-8}$ for the *KDD* and *URL* datasets respectively. These values are chosen in such a way that they are centered around the respective optimal λ value and have good sparsity variations over the optimal solution. With respect to Algorithm 1, the working set size (WSS) per node and number of nodes (P) are common across all the methods. We set WSS as a fraction (r) of the number of features per node, i.e., WSS$=r\,m/P$. Note that WSS will change with P for a given fraction r. For *KDD* and *URL*, we used three r values $(0.01, 0.1, 0.25)$ and $(0.001, 0.01, 0.1)$ respectively. We experimented with $P = 25, 100$. Also, r does not play a role in ADMM since all variables are optimized in each node.

Platform: We ran all our experiments in a Hadoop cluster with 200 nodes. Each node has Intel (R) Xeon (R) E5-2450L (2 processors) running at 1.8 GHz and 192 GB RAM. (Though both the datasets can fit in this memory configuration, our intention is to test the performance in a distributed setting.) All our implementations were done in $C\#$ including our binary tree *AllReduce* support [1] on Hadoop.

6.2 Experimental Results

Method Specific Parameter Setting: We discuss method specific parameter setting used in our experiments and associated practical implications.

[6] See `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`. We refer to kdd2010 (algebra) dataset as KDD.

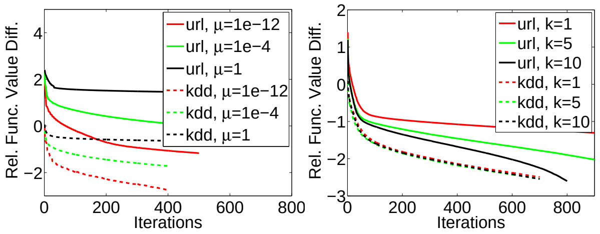


Figure 1: Left: the effect of μ. Right: the effect of the number of cycles to minimize f_p^t. $\mu = 10^{-12}$ and $k = 10$ are good choices. $P = 100$.

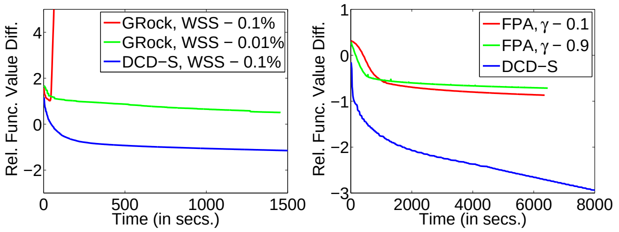


Figure 2: Left: Divergence and slow convergence of GRock on the URL dataset ($\lambda = 2.4 \times 10^{-6}$ and $P = 25$). Right: Extremely slow convergence of FPA on the KDD dataset ($\lambda = 4.6 \times 10^{-7}$ and $P = 100$).

In ADMM, the augmented Lagrangian parameter (ρ) plays an important role in getting good performance. While many schemes have been discussed in the literature [3] we found that selecting ρ using the objective function value gave a good estimate; we selected ρ^* from a handful of values with ADMM run for 10 iterations (i.e., not full training). However, this step incurred some computational/communication time. In our time plots shown later, late start of ADMM results is due to this cost. Note that this minimal number of iterations was required to get a decent ρ^*.

To get a practical implementation that gives good performance in our method, we deviate slightly from the conditions of Theorem 1. First, we find that the proximal term does not contribute usefully to the progress of the algorithm (see the left plot in Figure 1). So we choose to set μ to a small value, e.g., $\mu = 10^{-12}$. Second, we replace the stopping condition (8) by simply using a fixed number of cycles of coordinate descent to minimize f_p^t. The second plot in Figure 1 shows the effect of number of cycles, k. We found that a good choice for the number of cycles is 10 and used this value in all our experiments.

For GRock, FPA and Hyd we set the constants (L^j) as suggested in respective papers. Unfortunately, we found GRock to be either unstable and diverging or extremely slow. The first plot in Figure 2 depicts these behaviors. The solid red line shows the divergence case. FPA requires an additional parameter (γ) setting for the stochastic approximation step size rule. Our experience was that setting right values for these parameters to get good performance can be tricky and highly dataset dependent. The second plot in Figure 2 shows the extremely slow convergence behavior of FPA. Therefore, we do not include GRock and FPA further in our study.

Performance Evaluation We compare the performance of all methods by studying the time versus AUPRC and RFVD plots for various choices of λ, working set size (WSS) and the number of nodes (P) on KDD and URL. Due to space limitation, we provide only representative plots; but, the observations that we make below hold for other plots too.

Figure 3 shows the objective function plots for (*KDD*) with λ set to 4.6×10^{-7}. We see that DCD-S clearly outperforms all other methods; for example, if we set the RFVD value to 0.01 as the stopping criterion, DCD-S is faster by an order of magnitude. PCD-S comes as the second best. The S-scheme gives significant speed improvement over the R-scheme. As we compare the performance for two different WSS (see Figure 3(a)(b)), larger WSS gives some improvement and this speed-up is significant for Hyd, PCD-R and DCD-R. Note that ADMM is WSS independent since all the variables are updated. Because all variables are updated, ADMM performs slightly better than Hyd, PCD-R and DCD-R when WSS is small (see Figure 3(a)(c)). In this case, other methods take some time to reach optimal values, when the working set is selected randomly using the R-scheme.

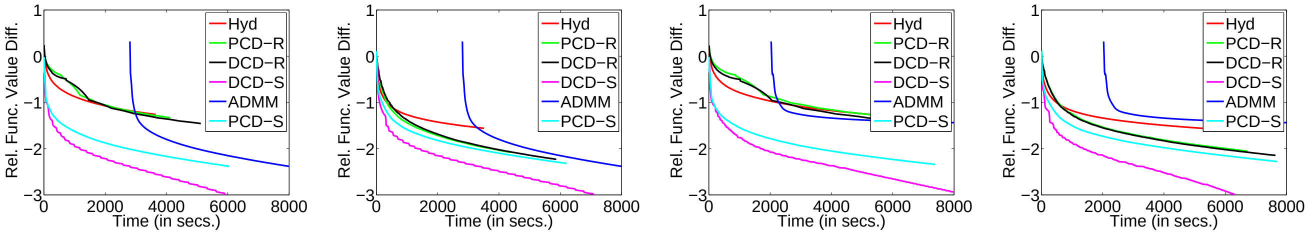
Table 4: Computation and communication costs per iteration (in secs.) for KDD, $P = 25$.

Method	Comp.	Comm.	Comp.	Comm.
	WSS - 1%		WSS - 10%	
Hyd	0.022	5.192	0.131	4.888
PCD-R	0.138	5.752	0.432	5.817
PCD-S	1.564	7.065	1.836	7.032
DCD-R	0.991	6.322	1.978	6.407
DCD-S	5.054	6.563	5.557	8.867

Figure 4 shows the objective function plots for (*URL*) with λ set to 9×10^{-8}. Here again, DCD-S gives the best RFVD performance with order of magnitude speed-up. Hyd suffers slow convergence and ADMM gives a decent second best performance. Interestingly, the new variable selection rule did not help PCD-R for large WSS. On comparing the performance for two different WSS, some speed improvement is achieved as in the case of KDD with similar observations. All these objective function progress behaviors are consistent with the AUPRC plots (Figures 5 and 6) as well except in one case. For example, the AUPRC performance of PCD-S is quite good although it is a bit slow on the objective function.

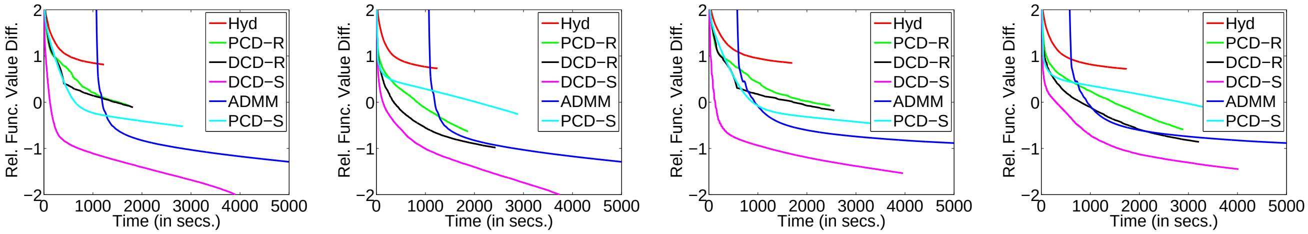
Figures 7 and 8 show the performance plots for another choice of different λ values for the datasets. DCD-S gives the best performance on *URL*. On *KDD*, it is the second best. This happens because the S scheme selects features having a large number of non-zero feature values. As a result, computation cost goes up a bit as we do more inner iterations compared to PCD-S. Nevertheless, it is still the second best. Overall, the results clearly show that DCD-S is preferred as it is highly effective with order of magnitude improvement under almost all conditions. Note that our proposal of using the S-scheme with the PCDN method [2] is of significant help.

Computation and Communication Time: As emphasized earlier, communication plays on important role in the distributed setting. To study this effect, we measured the computation and communication time separately at each node. Figure 9 shows the computation time per node on the *KDD* dataset. In both cases, ADMM incurs significant computation time compared to other methods. This is be-



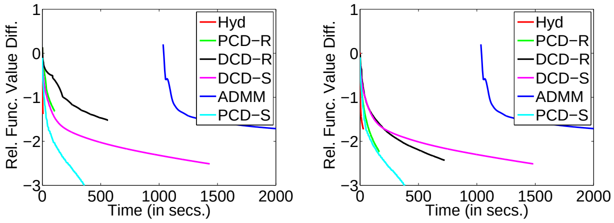
(a) $P = 25$, $WSS = 8086$ (b) $P = 25$, $WSS = 80867$ (c) $P = 100$, $WSS = 2021$ (d) $P = 100$, $WSS = 20216$

Figure 3: KDD dataset. Relative function value difference in log scale. $\lambda = 4.6 \times 10^{-7}$



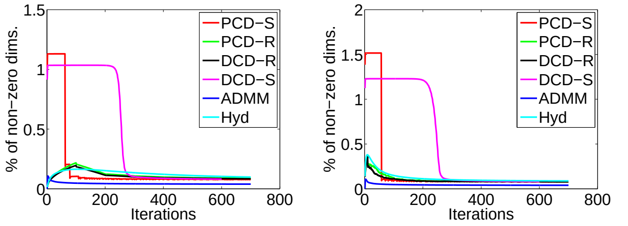
(a) $P = 25$, $WSS = 1292$ (b) $P = 25$, $WSS = 12927$ (c) $P = 100$, $WSS = 323$ (d) $P = 100$, $WSS = 3231$

Figure 4: URL dataset. Relative function value difference in log scale. $\lambda = 9.0 \times 10^{-8}$



(a) $WSS = 2021$ (b) $WSS = 20216$

Figure 9: Per-node computation time on the KDD dataset ($\lambda = 1.2 \times 10^{-5}$ and $P = 100$).



(a) $WSS = 2021$ (b) $WSS = 20216$

Figure 10: KDD dataset: Percentage of non-zero weights. $\lambda = 1.2 \times 10^{-5}$ **and** $P = 100$.

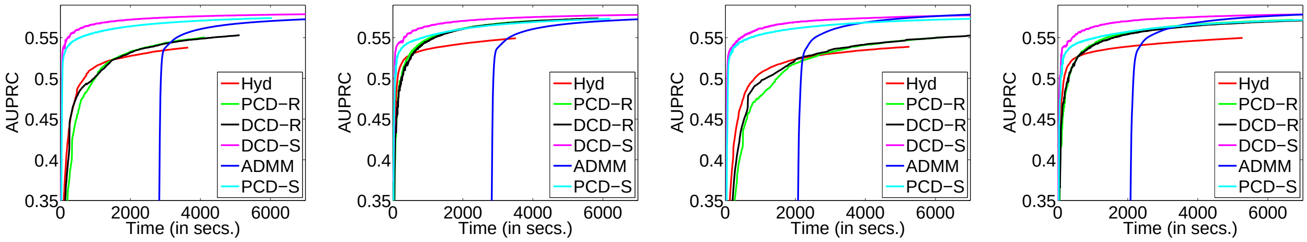
cause it optimizes over all variables in each node. DCD-S and DCD-R come next because our method involves both line search and 10 inner iterations. PCD-R and PCD-S take a little more time than Hyd because of the line search. As seen in both DCD and PCD cases, a marginal increase in time is incurred due to the variable selection cost with the S-scheme compared to the R-scheme.

We measured the computation and communication time taken per iteration by each method for different P and WSS settings. From Table 4 (which gives representative results for one situation, KDD and $P = 25$), we see that the communication time dominates the cost in Hyd and PCD-R. DCD-R takes more computation time than PCD-R and Hyd since we run through 10 cycles of inner optimization. Note that the methods with S-scheme take more time; however, the increase is not significant compared to the communication cost. DCD-S takes the maximum computation time and is quite comparable to the communication time. Recall our earlier observation of DCD-S giving order of magnitude speed-up in the overall time compared to methods such as Hyd and PCD-R (see Figures 3-8). Though the computation times taken by Hyd, PCD-R and PCD-S are lesser, they need significantly more number of iterations to reach some specified objective function optimality criterion. As a result, these methods become quite inefficient due to extremely large communication cost compared to DCD. All these observations point to the fact our DCD method nicely trades-off the computation versus communication cost, and gives an excellent order of magnitude improvement in overall time. With the additional benefit provided by the S-scheme, DCD-S clearly turns out to be the method of choice for the distributed setting.

Sparsity Pattern To study weight sparsity behaviors of various methods during optimization, we computed the percentage of non-zero weights (ρ) as a function of outer iterations. We set the initial weight vector to zero. Figure 10 shows similar behaviors for all the random (variable) selection methods. After a few iterations of rise they fall exponentially and remain at the same level. For methods with the S-scheme, many variables remain non-zero for some initial period of time and then ρ falls a lot more sharply. It is interesting to note that such an initial behavior seems necessary to make good progress in terms of both function value and AUPRC (Figure 7(a)(b) and Figure 8(a)(b)) In all the cases, many variables stay at zero after initial iterations; therefore, shrinking ideas can be used to improve efficiency.

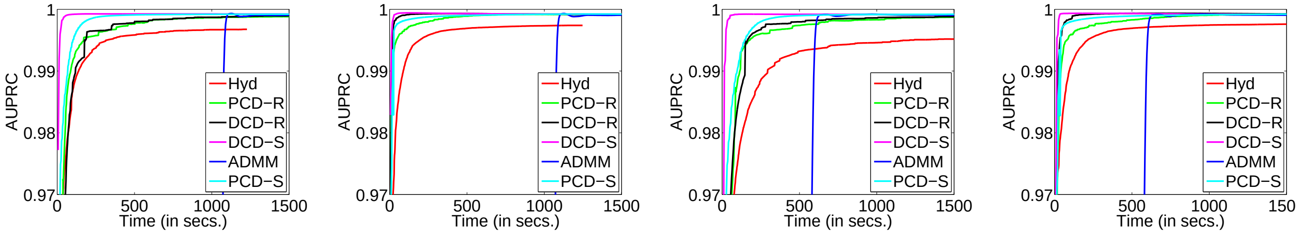
7. CONCLUSION

In this paper we have proposed a class of efficient block coordinate methods for the distributed training of l_1 regularized linear classifiers. In particular, the proximal-Jacobi approximation together with a distributed greedy scheme for variable selection came out as a strong performer. There are



(a) $P = 25$, $WSS = 8086$ (b) $P = 25$, $WSS = 80867$ (c) $P = 100$, $WSS = 2021$ (d) $P = 100$, $WSS = 20216$

Figure 5: KDD dataset. AUPRC Plots. $\lambda = 4.6 \times 10^{-7}$



(a) $P = 25$, $WSS = 1292$ (b) $P = 25$, $WSS = 12927$ (c) $P = 100$, $WSS = 323$ (d) $P = 100$, $WSS = 3231$

Figure 6: URL dataset. AUPRC plots. $\lambda = 9.0 \times 10^{-8}$

several useful directions for the future. It would be useful to explore other approximations such as block GLMNET and block L-BFGS suggested in Section 3.1. Like Richtárik and Takác [19], developing a complexity theory for our method that sheds insight on the effect of various parameters (e.g., P) on the number of iterations to reach a specified optimality tolerance is worthwhile. It is possible to extend our method to non-convex problems, e.g., deep net training, which has great value.

8. REFERENCES

[1] A. Agarwal, O. Chapelle, M. Dudik, and J. Langford. A reliable effective terascale linear system. *arXiv:1110.4198*, 2013.

[2] Y. Bian, X. Li, and Y. Liu. Parallel coordinate descent Newton for large scale L1 regularized minimization. *arXiv:1306.4080v1*, 2013.

[3] 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*, pages 1–122, 2011.

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

[5] W. Deng, M.-J. Lai, and W. Yin. On the $o(\frac{1}{k})$ convergence and parallelization of the alternating direction method of multipliers. *arXiv:1312.3040*, 2013.

[6] W. Deng and W. Yin. On the global and linear convergence of the generalized alternating direction method of multipliers. 2012.

[7] F. Facchinei, S. Sagratella, and G. Scutari. Flexible parallel algorithms for big data optimization. *arXiv:1311.2444*, 2013.

[8] J. H. Friedman, T. Hastie, and R. Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33:1–22, 2010.

[9] T. Goldstein, O'Donoghue, S. Setzer, and R. Baraniuk. Fast alternating direction optimization methods. *Technical Report, UCLA Mathematics*, 2012.

[10] I. Necoara and D. Clipici. Efficient parallel coordinate descent algorithm for convex optimization problems with separable constraints: application to distributed MPC. *arXiv:1302.3092*, 2013.

[11] Y. Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal of Optimization*, pages 341–362, 2012.

[12] J. M. Ortega and W. C. Rheinboldt. *Iterative solution of nonlinear equations in several variables.* Academic Press, New York, 1970.

[13] N. Parikh and S. Boyd. Block splitting of distributed optimization. *Math. Prog. Comp.*, 2013.

[14] M. Patriksson. Cost approximation: A unified framework of descent algorithms for nonlinear programs. *SIAM J. Optim.*, 8:561–582, 1998.

[15] M. Patriksson. Decomposition methods for differentiable optimization problems over cartesian product sets. *Comput. Optim. Appl.*, 9:5–42, 1998.

[16] Z. Peng, M. Yan, and W. Yin. Parallel and distributed sparse optimization. *Preprint, UCLA*, 2013.

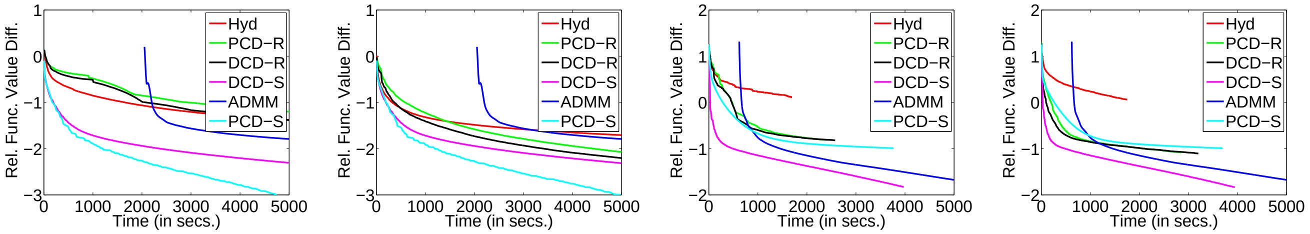
[17] C. Ravazzi, S. M. Fosson, and E. Magli. Distributed soft thresholding for sparse signal recovery. *arXiv:1301.2130*, 2012.

[18] P. Richtárik and M. Takác. Iteration complexity of randomized block-coordinate descent methods for minimizing a composite function. *arXiv:1212.0873*, 2012.

[19] P. Richtárik and M. Takác. Parallel coordinate descent methods for big data optimization. *arXiv:1212.0873*, 2012.

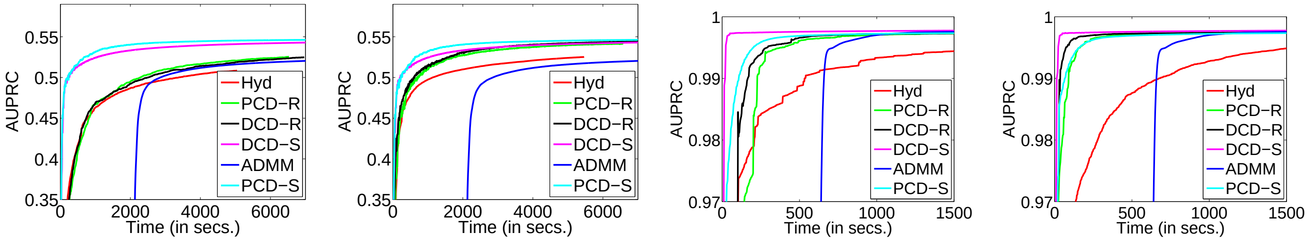
[20] P. Richtárik and M. Takác. Distributed coordinate descent method for learning with big data. *arXiv:1310.2059*, 2013.

[21] C. Scherrer, M. Halappanavar, A. Tewari, and



(a) KDD, $WSS = 2021$ (b) KDD, $WSS = 20216$ (c) URL, $WSS = 323$ (d) URL, $WSS = 3231$

Figure 7: Relative function value difference in log scale. KDD dataset: $\lambda = 1.2 \times 10^{-5}$. URL dataset: $\lambda = 7.3 \times 10^{-7}$



(a) KDD, $WSS = 2021$ (b) KDD, $WSS = 20216$ (c) URL, $WSS = 323$ (d) URL, $WSS = 3231$

Figure 8: AUPRC plots. KDD dataset: $\lambda = 1.2 \times 10^{-5}$. URL dataset: $\lambda = 7.3 \times 10^{-7}$

D. Haglin. Scaling up coordinate descent algorithms for large l_1 regularization problems. *Technical Report, PNNL*, 2012.

[22] C. Scherrer, A. Tewari, M. Halappanavar, and D. Haglin. Feature clustering for accelerating parallel coordinate descent. *NIPS*, pages 28–36, 2012.

[23] S. Shalev-Shwartz and A. Tewari. Stochastic methods for $l1$ regularized loss minimization. *JMLR*, 2011.

[24] P. Tseng and S. Yun. A coordinate gradient descent method for nonsmooth separable minimization. *Mathematical Programming*, 117:387–423, 2009.

[25] G. X. Yuan, K. W. Chang, C. J. Hsieh, and C. J. Lin. A comparison of optimization methods and software for large-scale l_1-regularized linear classification. *JMLR*, pages 3183–3234, 2010.

[26] G. X. Yuan, C. H. Ho, and C. J. Lin. An improved GLMNET for L1-regularized logistic regression and support vector machines. *JMLR*, pages 1999–2030, 2012.

[27] S. Yun, P. Tseng, and K. Toh. A coordinate gradient descent method for L1-regularized convex minimization. *Computational Optimization and Applications*, 48:273–307, 2011.

APPENDIX

Proof of Theorem 1. First let us write δ^j in (8) as $\delta^j = E_{jj}(d^t_{B_p})^j$ where $E_{jj} = \delta^j/(d^t_{B_p})^j$. Note that $|E_{jj}| \le \mu/2$. Use (12) with $w_{B_p} = \bar{w}^t_{B_p}$ and $\hat{w}_{B_p} = w_{B_p}$ in (8) together with the gradient consistency property of **P1** to get

$$g_{S^t_p} + H^t_{S^t_p} d^t_{S^t_p} + \xi_{S^t_p} = 0, \qquad (13)$$

where $H^t_{S^t_p} = \hat{H}_{S^t_p} - E_{S^t_p}$ and $\hat{H}_{S^t_p}$ is the diagonal submatrix of $\hat{H}$ corresponding to S^t_p. Since $\hat{H} \ge \mu I$ and $|E_{jj}| \le \mu/2$, we get $H^t_{S^t_p} \ge \frac{\mu}{2} I$. Let us extend the diagonal matrix $E^t_{S^t_p}$ to E_{B_p} by defining $E_{jj} = 0 \;\forall j \in B_p \setminus S^t_p$. This lets us extend $H^t_{S^t_p}$ to H_{B_p} via $H^t_{B_p} = \hat{H}_{B_p} - E_{B_p}$.

Now (13) is the optimality condition for the quadratic minimization,

$$d^t_{B_p} = \arg\min_{d_{B_p}} \; (g_{B_p})^T d_{B_p} + \frac{1}{2}(d_{B_p})^T H_{B_p} d_{B_p} + \sum_{j \in B_p} \lambda \, |(w^t)^j + d^j_{B_p}| \;\text{ s.t. }\; d^j_{B_p} = 0 \;\forall\, j \notin B_p \setminus S^t_p \qquad (14)$$

Combined over all p,

$$d^t = \arg\min_{d} \; (g^t)^T d + \frac{1}{2} d^T H d + u(w^t + d) \quad \text{s.t. } \; d^j = 0 \;\forall\, j \notin \cup_p (B_p \setminus S^t_p) \qquad (15)$$

where H is a block diagonal matrix with blocks, $\{H_{B_p}\}$. Thus d^t corresponds to the minimization of a positive definite quadratic form, exactly the type covered by the Tseng-Yun theory [24].

The line search condition (10)-(11) is a special case of the line search condition in [24]. The Gauss-Seidel scheme of Section 3.2 is an instance of the Gauss-Seidel scheme of [24]. Now consider the distributed greedy scheme in Section 3.2. Let $j_{\max} = \arg\max_{1 \le j \le m} \bar{q}^j$. By the way the S^t_p are chosen, $j_{\max} \in \cup_p S^t_p$. Therefore, $\sum_{j \in \cup_p S^t_p} \bar{q}^j \le \frac{1}{m} \sum_{j=1}^m \bar{q}^j$, thus satisfying the Gauss-Southwell-q rule condition of [24]. Now Theorems 1-3 of [24] can be directly applied to prove our Theorem 1.