Journal of Machine Learning Research 18 (2017) 1-35 Submitted 11/14; Revised 11/16; Published 9/17

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

Dhruv Mahajan DHRUVM@FB.COM
Applied Machine Learning Group
Facebook Research
Menlo Park, CA 94025, USA

S. Sathiya Keerthi KEERTHI@MICROSOFT.COM
Office Data Science Group
Microsoft
Mountain View, CA 94043, USA

S. Sundararajan SSRAJAN@MICROSOFT.COM
Microsoft Research
Bangalore, India

Editor: Vishwanathan S V N

Abstract

Distributed training of l_1 regularized classifiers has received great attention recently. Most 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 systems 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 and motivate our method. The main idea of our algorithm is to do block optimization of many variables on the actual objective function within each computing node; this increases the computational cost per step that is matched with the communication cost, and decreases the number of outer iterations, thus yielding a faster overall method. Distributed Gauss-Seidel and Gauss-Southwell 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.

Keywords: Distributed learning, l_1 regularization

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. Big data is becoming common nowadays. For example, in online advertising one comes across datasets with about a billion examples and a billion features. A substantial fraction of the features is usually irrelevant; and, l_1 regularization offers a systematic way to choose the small fraction of relevant features and form the classifier using them. In the future, one can

foresee even bigger sized datasets to arise in this and other applications. For such big data, 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; see section 3 for a review.[1] Most of these algorithms are based on coordinate-descent and they assume the data to be feature-partitioned. They are designed for multicore systems in which data communication costs are negligible. Recently, distributed systems with Hadoop running on a cluster of commodity machines have become popular. In such systems, communication costs are generally high; current methods for l_1 regularization are not optimally designed for such systems. Recently there has been been increased attention given to designing communication-efficient algorithms (Jaggi et al., 2014; Ma et al., 2015). *In this paper we develop a distributed block coordinate descent (DBCD) method that is efficient on distributed platforms in which communication costs are high.*

Following are the main contributions of this paper.

1. Most methods for the parallel training of l_1 regularized classifiers (including the ones proposed in this paper) fall into a generic algorithm format (see algorithm 1 in section 2). We make careful choices for the three key steps of this algorithm, 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 provide a detailed cost analysis (section 5) that brings out the computation and communication costs of the generic algorithm clearly for different methods. In the process we motivate the need for new efficient methods such as DBCD that are suited to communication heavy settings.

3. We establish convergence theory (subsection 4.4) for our method using the results of Tseng and Yun (2009) and Yun et al. (2011). It is worth noting the following: (a) though Tseng and Yun (2009) and Yun et al. (2011) 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 these two works use only per-feature decoupled quadratic models in their implementations whereas we work with more powerful approximations that couple features.

4. We give an experimental evaluation (section 6) that shows the strong performance of DBCD against key current methods in scenarios where communication cost is significant. Based on the experiments we make a final recommendation for the best method to employ for such scenarios.

The paper is organized as follows. The generic algorithm format is described in section 2. This gives a clear view of existing methods and allows us to motivate the new method. In section 3 we discuss the key related work in some detail. In section 4 we describe the DBCD method in detail and prove its convergence. The analysis of computation and communication costs in section 5 gives a firmer motivation of our DBCD method. Experiments

1. In this paper we only consider synchronous distributed training algorithms in which various computing nodes share their information and complete one update. Asynchronous methods (Li et al., 2014) form another important class of methods that needs a separate study.

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. This best version of the DBCD method is described in section 7. Section 8 contains some concluding comments.

2. A generic algorithm

Various distributed solvers of the l_1 regularization problem can be put in a generic algorithm format. We begin this section by describing the problem formulation. Then we state the generic algorithm format. We follow this by discussing the choices various methods make for the steps and point out how new choices for the steps can lead to a better design.

2.1 Problem formulation

Let w be the weight vector with m variables, w_j, $j = 1, \ldots, m$, and $x_i \in R^m$ denote the i-th example. Let there be n training examples and let X denote the $n \times m$ data matrix, whose i-th row is x_i^T. Note that we have denoted vector components by subscripts, e.g., w_j is the j-th component of w; we have also used subscripts for indexing examples, e.g., x_i is the i-th example, which itself is a vector. But this will not cause confusion anywhere. A linear classifier produces the output $y_i = w^T x_i$. The loss is a nonlinear convex function applied on the output. For binary class label $c_i \in \{1, -1\}$, the loss is given by $\ell(y_i; c_i)$. Let us simply view $\ell(y_i; c_i)$ as a function of y_i with c_i acting as a parameter. We will assume that ℓ is non-negative and convex, $\ell \in \mathcal{C}^1$, the class of continuously differentiable functions, and that ℓ' is Lipschitz continuous[2]. Loss functions such as least squares loss, logistic loss, SVM squared hinge loss and Huber loss satisfy these assumptions. All experiments reported in this paper use the squared hinge loss, $\ell(y_i; c_i) = \max\{0, 1 - c_i y_i\}^2$. The total loss function, $f : R^m \to R$ is $f(w) = \frac{1}{n} \sum_i \ell(y_i; c_i)$. 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

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

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

$$\forall j : \quad g_j + \lambda \, \text{sign}(w_j) = 0 \text{ if } |w_j| > 0; \quad |g_j| \leq \lambda \text{ if } w_j = 0. \tag{2}$$

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 $\mathcal{M} = \{1, \ldots, m\}$, i.e., $B_p \subset \mathcal{M}$ $\forall p$ and $\cup_p B_p = \mathcal{M}$. *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.

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

2.2 Generic algorithm

Algorithm 1 gives the generic algorithm. In each iteration t, the following steps happen, in parallel, in each node (p): (a) a subset of variables, S_p^t is chosen; (b) a suitable approximating function, f_p^t is formed and the chosen variables are optimized so as to define a direction; (c) a step size is chosen along that direction to update the weight vector on the chosen variables. The outputs of all examples are then computed using an *AllReduce* operation (step (d)) and the algorithm is terminated if optimality conditions are satisfied (step (e)).

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 *AllReduce* operation (Agarwal et al., 2013) over the nodes and then y becomes available in all the nodes. The gradient subvector $g_{B_p}^t$ (which is needed for solving (3)) 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 $y^0 = Xw^0$;
for $t = 0, 1 \ldots$ **do**
  for $p = 1, \ldots, P$ **do**
    (a) Select a working subset of variables[3], $S_p^t \subset B_p$;
    (b) Form $f_p^t(w_{B_p})$, an approximation of f and minimize, exactly or approximately, $f_p^t + u$ over only the weights corresponding to S_p^t:

$$\min f_p^t(w_{B_p}) + u(w_{B_p}) \quad \text{s.t.} \quad w_j = w_j^t \; \forall \; j \in B_p \setminus S_p^t \tag{3}$$

    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 α^t and update: $w_{B_p}^{t+1} = w_{B_p}^t + \alpha^t d_{B_p}^t$;
  end
  (d) Update $y^{t+1} = y^t + \alpha^t \sum_p X_{B_p} d_{B_p}^t$;
  (e) Terminate if optimality conditions hold;
end

Steps (d) and (e) of Algorithm 1 are quite straight-forward. But the first three steps, (a)-variable sampling, (b)-function approximation, and (c)-step size determination, can be implemented in various ways and require a detailed discussion.

Step (a) - variable sampling. Some choices are:

- **(a.1)** random selection (Bradley et al., 2011; Richtárik and Takáč, 2014);
- **(a.2)** random cyclic: over a set of consecutive iterations (t) all variables are touched once (Bian et al., 2013);
- **(a.3)** greedy: always choose a set of variables that, in some sense violate (2) the most at the current iterate (Peng et al., 2013; Facchinei et al., 2014); and,
- **(a.4)** greedy selection using the Gauss-Southwell rule (Tseng and Yun, 2009; Yun et al., 2011).

Step (b) - function approximation. It would be ideal to choose f_p^t to be f itself. However, to make the solution simple and efficient, most methods choose a quadratic approximation that is decoupled at the individual variable level:

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

The main advantages of (4) are its simplicity and closed-form minimization when used in (3). Choices for L^j that have been tried are:

- **(b.1)** $L_j =$ a Lipschitz constant for g_j (Bradley et al., 2011; Peng et al., 2013);
- **(b.2)** $L_j =$ a large enough bound on the Lipschitz constant for g_j to suit the sampling in step (a) (Richtárik and Takáč, 2014);
- **(b.3)** adaptive adjustment of L_j (Facchinei et al., 2014); and
- **(b.4)** $L_j = H_{jj}^t$, the j-th diagonal term of the Hessian at w^t (Bian et al., 2013).

Step (c) - step size. The choices are:

- **(c.1)** always fix $\alpha^t = 1$ (Bradley et al., 2011; Richtárik and Takáč, 2014; Peng et al., 2013);
- **(c.2)** use stochastic approximation ideas to choose $\{\alpha^t\}$ so that $\sum_t (\alpha^t)^2 < \infty$ and $\sum_t |\alpha^t| = \infty$ (Facchinei et al., 2014); and
- **(c.3)** choose α^t by line search that is directly tied to the optimization of F in (1) (Bian et al., 2013).

2.3 Discussion of choices for steps (a)-(c)

To understand the role of the various choices better, let us first focus on the use of (4) for f_p^t. Algorithm 1 may not converge to the optimal solution due to one of the following decisions: (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). This is because each of the above has the potential to cause large step sizes leading to increases in F value and, if this happens uncontrolled at all iterations then convergence to the minimum cannot occur. 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 $|S_p^t|$. Choice **(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.

3. We will refer to the working subset size, i.e., $|S_p^t|$, as WSS.

Except for Bian et al. (2013), Tseng and Yun (2009) and Yun et al. (2011)[4], **(c.3)** has been unused in other methods because it is considered as 'not-in-line' with a proper parallel approach as it requires a separate α^t determination step requiring distributed computations and also needing F computations for several α^t values within one t. With line search, the actual implementation of Algorithm 1 merges steps **(c)** and **(d)** and so it deviates slightly from the flow of Algorithm 1. Specifically, we compute $\delta y = \sum_p X_{B_p} d^t_{B_p}$ before line search using AllReduce. Then each node can compute f at any α locally using $y + \alpha\, \delta y$. Only a scalar corresponding to the l_1 regularization term needs to be communicated for each α. This means that the communication cost associated with line search is minimal.[5] 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^t_p in step (b) (in particular, **(b.4)**) is pretty much unanimously used in all previous works. This is done to make the optimization simple. While this is fine for communication friendly systems such as multicore, 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^t_p. We propose the use of a function f^t_p that couples the variables in S^t_p. 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^t_p. 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.

3. 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 (Bradley et al., 2011; Richtárik and Takáč, 2015; Bian et al., 2013; Peng et al., 2013). In contrast, there exist only a few efficient algorithms to solve (1) when the data is distributed (Richtárik and Takáč, 2016; Ravazzi et al., 2013) 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 (Peng et al., 2013; Boyd et al., 2011): (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 (Peng et al., 2013) or ADMM (Boyd et al., 2011) are useful for (A), the block splitting method (Parikh and Boyd, 2013) is useful for (C). We are interested in (B), and the most relevant and important class of methods is parallel/distributed coordinate descent methods, as abstracted in algorithm 1. Most of these methods set f^t_p in step (b) of algorithm 1 to be a quadratic

4. Among these three works, Tseng and Yun (2009) and Yun et al. (2011) mainly focus on general theory and little on distributed implementation.

5. Later, in section 5 when we write costs, we write it to be consistent with Algorithm 1. The total cost of all the steps is the same for the implementation described here for line search. For genericity sake, we keep Algorithm 1 as it is even for the line search case. The actual details of the implementation for the line search case will become clear when we layout the final algorithm in section 7.

approximation that is decoupled at the individual variable level. Table 3 compares these methods along various dimensions.[6]

Most dimensions arise naturally from the steps of algorithm 1, as explained in section 2. Two important points to note are: (i) except Richtárik and Takáč (2016) and our method, none of these methods target and sufficiently discuss distributed setting involving communication and, (ii) 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 Bradley et al. (2011); Richtárik and Takáč (2014, 2015); Peng et al. (2013) that do not do line search are shown to have the monotone property only in expectation and that too only 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 parallel coordinate descent Newton method (Bian et al., 2013) (see below for a discussion) enjoy robustness to various settings and come out as clear winners.

It is beyond the scope of this paper to give a more detailed discussion, beyond Table 3, of the methods from a theoretical convergence perspective on various assumptions and conditions under which results hold. We only briefly describe and comment on them below.

Generic Coordinate Descent Method (Scherrer et al., 2012a,b) Scherrer et al. (2012a) and Scherrer et al. (2012b) presented an abstract framework for coordinate descent methods (GenCD) suitable for parallel computing environments. Several coordinate descent algorithms such as stochastic coordinate descent (Shalev-Shwartz and Tewari, 2011), Shotgun (Bradley et al., 2011) and GROCK (Peng et al., 2013) are covered by GenCD. GROCK is a thread greedy algorithm (Scherrer et al., 2012a) 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 (see section 6). Therefore, the practical utility of such algorithms is limited without ensuring necessary descent property through certain spectral radius conditions on the data matrix.

Distributed Coordinate Descent Method (Richtárik and Takáč, 2016) The multi-core parallel coordinate descent method of Richtárik and Takáč (2014) is a much refined version of GenCD with careful choices for steps (a)-(c) of algorithm 1 and a supporting stochastic convergence theory. Richtárik and Takáč (2016) extended this to the distributed setting; so, this method is more relevant to this paper. With no line search, their algorithm HYDRA (Hybrid coordinate 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 the number of non-zero elements as suggested in their work) results in slower convergence (see section 6).

Necoara and Clipici (2014) suggest another variant of parallel coordinate descent in which all the variables are updated in each iteration. HYDRA and GROCK can be

6. Although our method will be presented only in section 4, we include our method's properties in the last row of Table 3. This helps to easily compare our method against the rest.

considered as two key, distinct methods that represent the set of methods discussed above. So, in our analysis as well as experimental comparisons in the rest of the paper, we do not consider the methods in this set other than these two.

Flexible Parallel Algorithm (FPA) (Facchinei et al., 2014) This method has some similarities with our method in terms of the approximate function optimized at the nodes. Though Facchinei et al. (2014) suggest several approximations, they use 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. (In section 6 we conduct experiments to show this.) Unlike our method, FPA's inner optimization stopping criterion is unverifiable (for e.g., with (6)); also, FPA does not address the communication cost issue.

Parallel Coordinate Descent Newton (PCD) (Bian et al., 2013) One key difference between other methods discussed above and our DBCD method is the use of line search. Note that the PCD method can be seen as a special case of DBCD (see subsection 5.1). 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 PCD).

Synchronized Parallel Algorithm (Patriksson, 1998b) Patriksson (1998b) proposed a Jacobi type synchronous parallel algorithm with line search using a generic cost approximation (CA) framework for differentiable objective functions (Patriksson, 1998a). Its local linear rate of convergence results hold only for a class of strong monotone CA functions. If we view the approximation function, f_p^t as a mapping that is dependent on w^t, Patriksson (1998b) requires this mapping to be continuous, which is unnecessarily restrictive.

ADMM Methods Alternating direction method of multipliers is a generic and popular distributed computing method. It does not fit into the format of Algorithm 1. This method can be used to solve (1) in different data splitting scenarios (Boyd et al., 2011; Parikh and Boyd, 2013). 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 (Deng and Yin, 2016; Deng et al., 2013). Recently, an accelerated version of ADMM (Goldstein et al., 2014) derived using the ideas of Nesterov's accelerated gradient method (Nesterov, 2012) 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.

Based on the above study of related work, we choose HYDRA, GROCK, PCD and FPA as the main methods for analysis and comparison with our method.[7] Thus, Table 3 gives various dimensions only for these methods.

7. In the experiments of section 6, we also include ADMM.

Method	**Is $\mathbf{F}(w^t)$ monotone?**	**Are limits forced on $\lvert S_p^t \rvert$?**	**How is $\mathbf{S}_p^t$ chosen?**	**Basis for choosing $\mathbf{L}_j$**	**How is α^t chosen**	**Convergence type**	**Convergence rate**
Existing methods							
HYDRA	No	No, if L_j is varied suitably	Random	Lipschitz bound for g_j suited to S_p^t choice	Fixed	Stochastic	Linear
GROCK	No	Yes	Greedy	Lipschitz bound for g_j	Fixed	Deterministic	Sub-linear
FPA	No	No	Random	Lipschitz bound for g_j	Adaptive	Deterministic	None
PCD	Yes	No	Random	Hessian diagonal	Armijo line search	Stochastic	Sub-linear
Our method							
DBCD	Yes	No	Random/Greedy	Free	Armijo line search	Deterministic	Locally linear

Table 1: Properties of selected methods that fit into the format of Algorithm 1. Methods: HYDRA (Richtárik and Takáč, 2016), GROCK (Peng et al., 2013), FPA (Facchinei et al., 2014), PCD (Bian et al., 2013).

4. 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 a general convergence theory for the class of algorithms that fall under DBCD. We recommend three specific instantiations of DBCD, analyze their costs in section 5, empirically study them in section 6 and make one final best recommendation in section 7. In this section, we also show the relations of DBCD to other methods on aspects such as variable selection, function approximation, line search, etc. As this section is traversed, it is also useful to re-visit table 3 and compare DBCD against other key methods.

Our goal is to develop an efficient distributed learning method that jointly optimizes the costs involved in the various steps of the algorithm. We observed in the previous section that the methods discussed there lack this careful optimization in one or more steps, resulting in inferior performance. This can be understood better via a cost analysis. To avoid too much deviation, we give the gist of this cost analysis here and postpone the full details to section 5. The cost of Algorithm 1 can be written as $T^P(C^P_{\text{comp}} + C^P_{\text{comm}})$ where P denotes the number of nodes, T^P is the number of outer iterations[8], and, C^P_{comp} and C^P_{comm} respectively denote the computation and communication costs per-iteration. In communication heavy situations, existing algorithms have $C^P_{\text{comp}} \ll C^P_{\text{comm}}$. Our method aims to improve overall efficiency by making each iteration more complex (C^P_{comp} is increased) and, in the process, making T^P much smaller.

4.1 Variable selection

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

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: $\{S^t_p\}_{t=\tau}^{\tau+T-1}$. 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 Tseng and Yun (2009).[9] 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:

$$q_j(w_j) = g^t_j(w_j - w^t_j) + \frac{1}{2}(H^t_{jj} + \nu)(w_j - w^t_j)^2 + \lambda|w_j| - \lambda|w^t_j| \tag{5}$$

8. For practical purposes, one can view T^P as the number of outer iterations needed to reach a specified closeness to the optimal objective function value. We will say this more precisely in section 6.
9. Yet, our distributed greedy scheme can be shown to imply the Gauss-Southwell-q rule for a certain parameter setting. See the appendix for details.

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 $q_j(w_j^t) = 0$, clearly $\bar{q}_j \leq 0$. The more negative $\bar{q}_j$ is, the better it is to choose j.

Our distributed greedy scheme first chooses a working set size, WSS (the size of S_p^t) and then, in each node p, it chooses the top WSS 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.*

It is worth pointing out that, our distributed greedy scheme requires more computation than the Gauss-Seidel scheme. However, since the increased computation is local, non-heavy and communication is the real bottleneck, it is not a worrisome factor.

4.2 Function approximation

Let us begin with step (b). There are three key items involved: (i) what are some of the choices of approximate functions possible, used by our methods and others? (ii) what is the stopping criterion for the inner optimization (i.e., local problem), and, (iii) what is the method used to solve the inner optimization? We discuss all these details below. 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.

Condition 1. $f_p^t \in \mathcal{C}^1$; $g_p^t = \nabla f_p^t$ 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 $w_{B_p}^t$, i.e., $g_p^t(w_{B_p}^t) = g_{B_p}(w^t)$.

This assumption is not restrictive. 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 $d_{B_p}^t$ a descent direction for F at $w_{B_p}^t$, 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. Our experiments indicate that it is sufficient to set μ to be a very small positive value. Lipschitz continuity is another technical condition that is needed for ensuring boundedness of various quantities; also, it is easily satisfied by most loss functions.

Choice of f_p^t. 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* Condition 1.

- **Proximal-Jacobi.** We can follow the classical Jacobi method in choosing f_p^t to be the restriction of f to $w_{S_p^t}^t$, 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

$$f_p^t(w_{B_p}) = f(w_{B_p}, w_{\bar{B}_p}^t) + \frac{\mu}{2}\|w_{B_p} - w_{B_p}^t\|^2 \tag{6}$$

  where $\mu > 0$ is the proximal constant. It is worth pointing out that, since each node p keeps a copy of the full classifier output vector y aggregated over all the nodes, 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) of Algorithm 1 can be executed in parallel for all p.

- **Block GLMNET.** GLMNET (Yuan et al., 2012; Friedman et al., 2010) is a sequential coordinate descent method that has been demonstrated to be very promising for the 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 $f(w_{B_p}, w_{\bar{B}_p}^t)$ with respect to w_{B_p} while keeping $w_{\bar{B}_p}$ fixed at $w_{\bar{B}_p}^t$. In other words, we can choose f_p^t as
$$f_p^t(w_{B_p}) = Q^t(w_{B_p}) + \frac{\mu}{2}\|w_{B_p} - w_{B_p}^t\|^2 \tag{7}$$
where Q^t is the quadratic approximation of $f(w_{B_p}, w_{\bar{B}_p}^t)$ with respect to w_{B_p} at $w_{B_p}^t$ with $w_{\bar{B}_p}^t$ fixed.

- **Block L-BFGS.** One can keep a limited history of $w_{B_p}^t$ and $g_{B_p}^t$ and use an $L-BFGS$ approach to build a second order approximation of f in each iteration to form f_p^t:
$$f_p^t(w_{B_p}) = (g_{B_p}^t)^T(w_{B_p} - w_{B_p}^t) + \frac{1}{2}(w_{B_p} - w_{B_p}^t)^T H_{BFGS}(w_{B_p} - w_{B_p}^t) + \frac{\mu}{2}\|w_{B_p} - w_{B_p}^t\|^2 \tag{8}$$
where H_{BFGS} is a limited memory BFGS approximation of the Hessian of $f(w_{B_p}, w_{\bar{B}_p}^t)$ with respect to w_{B_p} with $w_{\bar{B}_p}^t$ fixed, formed using $\{g_{B_p}^\tau\}_{\tau \le t}$.

- **Decoupled quadratic.** Like in existing methods we can also form a quadratic approximation of f that decouples at the variable level - see (4). (An additional proximal term can be added.) If the second order term is based on the diagonal elements of the Hessian at w^t, then the PCDN algorithm given in Bian et al. (2013) can be viewed as a special case of our DBCD method. PCDN (Bian et al., 2013) is based on Gauss-Seidel variable selection. But it can also be used in combination with the distributed greedy scheme that we propose in subsection 4.1 below.

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 subsection 4.4, also give convergence theory to support it.

Let ∂u_j be the set of subgradients of the regularizer term $u_j = \lambda|w_j|$, i.e.,
$$\partial u_j = [-\lambda, \lambda] \text{ if } w_j = 0; \;\; \lambda \,\text{sign}(w_j) \text{ if } w_j \neq 0. \tag{9}$$
A point $\bar{w}_{B_p}^t$ is optimal for (3) if, at that point,
$$(g_p^t)_j + \xi_j = 0, \;\; \text{for some } \; \xi_j \in \partial u_j \;\; \forall \; j \in S_p^t. \tag{10}$$
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
$$\delta^j = (g_p^t)_j + \xi_j, \;\; |\delta_j| \le \epsilon |d_j^t| \;\; \forall \; j \in S_p^t \tag{11}$$

A practical alternative is to replace the stopping condition (11) by simply using a fixed number of cycles of coordinate descent to minimize f_p^t.

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 (Yuan et al., 2010) we use the coordinate-descent method described in Yuan et al. (2010) for solving (3). For logistic regression loss, it is appropriate to use the new-GLMNET method (Yuan et al., 2012).

4.3 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 (2009) and Patriksson (1998a) 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

$$F(w^t + \alpha^t d^t) \leq F(w^t) + \alpha^t \sigma \Delta^t, \tag{12}$$

$$\Delta^t \stackrel{\text{def}}{=} (g^t)^T d^t + \lambda u(w^t + d^t) - \lambda u(w^t). \tag{13}$$

4.4 Convergence

We now establish convergence for the class of algorithmic choices discussed in subsections 4.2-4.3. To do this, we make use of the results of Tseng and Yun (2009). An interesting aspect of this use is that, while the results of Tseng and Yun (2009) 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 Condition 1 (see subsection 4.2) we need one other technical assumption.

Condition 2. 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

$$\hat{g}^t_{B_p}(w_{B_p}) - \hat{g}^t_{B_p}(\hat{w}_{B_p}) = \hat{H}(w_{B_p} - \hat{w}_{B_p}) \tag{14}$$

In the above, $\hat{g}^t_{B_p}$ is the gradient with respect to w_{B_p} of the approximate function f_p^t formed in step (b) of algorithm 1. Note that $\hat{g}^t_{B_p}(w^t_{B_p}) = g^t_{B_p}$.

Except *Proximal-Jacobi*, the other instantiations of f_p^t mentioned in subsection 4.2 are quadratic functions; for these, g_p^t is a linear function and so (14) holds trivially. Let us turn to *Proximal-Jacobi*. If $f_p^t \in \mathcal{C}^2$, the class of twice continuously differentiable functions, then Condition 2 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 Condition 2 easily holds for least squares loss and logistic loss. Now consider the SVM squared hinge loss, $\ell(y_i; c_i) = 0.5(\max\{0, 1 - y_i c_i\})^2$, which is not in $\mathcal{C}^2$. Condition 2 holds for it because $g = \sum_i \ell'(y_i; c_i)x_i$ and, for any two real numbers z_1, z_2, $\ell'(z_1; c_i) - \ell'(z_2; c_i) = \kappa(z_1, z_2, c_i)(z_1 - z_2)$ where $0 \leq \kappa(z_1, z_2, c_i) \leq 1$.

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

Theorem 1. Suppose, in Algorithm 1, the following hold:

(i) step (a) is done via the Gauss-Seidel or distributed greedy schemes of subection 5.2;

(ii) f_p^t in step (b) satisfies Condition 1 and Condition 2;

(iii) (11) is used to terminate (3) with $\epsilon = \mu/2$ (where μ is as in Condition 1); and,

(iv) in step (c), α^t is chosen via Armijo backtracking of subection 5.3.

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.[10]

4.5 Specific instantiations of DBCD

For function approximation we found the proximal-Jacobi choice, (6) to be powerful. This choice, combined with variable selection done using the R-scheme and S-scheme leads to the two specific instantiations, DBCD-R and DBCD-S. As we explained in subsection 4.2, the PCDN algorithm given by Bian et al. (2013) (referred to as PCD) is a special case of DBCD using a decoupled quadratic function approximation and the R-scheme for variable selection. We also recommend the use of S-scheme with this method and call that instantiation as PCD-S.

5. DBCD method: Cost analysis

As pointed out earlier, the DBCD method is motivated by an analysis of the costs of various steps of algorithm 1. In this section, we present a detailed cost analysis that explains this more clearly. Based on section 3, we select the following five methods for our study: (1) HYDRA (Richtárik and Takáč, 2016), (2) GROCK (Greedy coordinate-block) (Peng et al., 2013), (3) FPA (Flexible Parallel Algorithm) (Facchinei et al., 2014), (4) PCD (Parallel Coordinate Descent Newton method) (Bian et al., 2013), and (5) DBCD. We will use the above mentioned abbreviations for the methods in the rest of the paper.

Let nz and $|S| = \sum_p |S_p^t|$ denote the number of non-zero entries in the data matrix X and the number of variables updated in each iteration respectively. To keep the analysis simple, we make the homogeneity assumption that the number of non-zero data elements in each node is nz/P. Let $\beta(\gg 1)$ be the relative computation to communication speed in the given distributed system; more precisely, it is the ratio of the times associated with communicating a floating point number and performing one floating point operation. On a cluster with 10 Gbps communication bandwidth where we did our experiments, we found the value of β to be in the range $30 - 100$. Recall that n, m and P denote the number of examples, features and nodes respectively. Table 5 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. Table 5 gives the cost constants for various methods.We briefly discuss different costs below.

10. See chapter 9 of Ortega and Rheinboldt (1970) for definitions of Q-linear and R-linear convergence.

Cost	Steps of Algorithm 1			
	Step (a) Variable selection	Step (b) Inner optimization	Step (c) Choosing step size	Step (d) Updating output
Computation	$c_1\frac{nz}{P}$	$c_2\frac{nz}{P}\frac{\lvert S\rvert}{m}$	$c_3\lvert S\rvert + c_4 n$	$c_5\frac{nz}{P}\frac{\lvert S\rvert}{m}$
Communication	-	-	≈ 0	$\approx \beta n$[11]

Table 2: Cost of various steps of Algorithm 1. C^P_{comp} and C^P_{comm} are respectively, the sums of costs in the computation and communication rows.

Method	c_1	c_2	c_3	c_4	c_5	Computation cost per iteration	Communication cost per iteration
Existing methods							
HYDRA	0	1	1	0	1	$2\frac{nz}{P}\frac{\lvert S\rvert}{m} + \lvert S\rvert$	βn
GROCK	1	q	1	0	q	$\frac{nz}{P} + 2q\frac{nz}{P}\frac{\lvert S\rvert}{m} + \lvert S\rvert$	βn
FPA	1	q	1	1	q	$\frac{nz}{P} + 2q\frac{nz}{P}\frac{\lvert S\rvert}{m} + \lvert S\rvert + n$	βn
PCD	0	1	τ_{ls}	τ_{ls}	1	$2\frac{nz}{P}\frac{\lvert S\rvert}{m} + \tau_{ls}\lvert S\rvert + \tau_{ls} n$	βn
Variations of our method							
PCD-S	1	q	τ_{ls}	τ_{ls}	q	$\frac{nz}{P} + 2q\frac{nz}{P}\frac{\lvert S\rvert}{m} + \tau_{ls}\lvert S\rvert + \tau_{ls} n$	βn
DBCD-R	0	k	τ_{ls}	τ_{ls}	1	$(k+1)\frac{nz}{P}\frac{\lvert S\rvert}{m} + \tau_{ls}\lvert S\rvert + \tau_{ls} n$	βn
DBCD-S	1	kq	τ_{ls}	τ_{ls}	q	$\frac{nz}{P} + q(k+1)\frac{nz}{P}\frac{\lvert S\rvert}{m} + \tau_{ls}\lvert S\rvert + \tau_{ls} n$	βn

Table 3: Cost parameter values and costs for different methods. q lies in the range: $1 \leq q \leq \frac{m}{\lvert S\rvert}$. R and S refer to variable selection schemes for step (a); see subsection 4.1. PCD uses the R scheme and so it can also be referred to as PCD-R. Typically τ_{ls}, the number of α values tried in line search, is very small; in our experiments we found that on average it is not more than 10. Therefore all methods have pretty much the same communication cost per iteration.

Step a: Methods like our DBCD-S[12], 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 HYDRA, PCD and DBCD-R select variables randomly or in a cyclic order. As a result variable subset selection cost is negligible for them ($c_1 = 0$).

11. Note that the communication latency cost (the time taken to communicate zero bytes) is ignored in the communication cost expressions because it is dominated by the throughput cost for large n. Moreover, as in Agarwal et al. (2013), the broadcast and reduce operators are pipelined over the vector entries. This means that communication cost increases sub-linearly wrt. $logP$. If n is assumed to be large (as in our case), it is almost independent of $logP$ and can be written approximately as βn.
12. The DBCD and PCD methods have two variants, R and S corresponding to different ways of implementing step a; see subsection 4.1.

Step b: All the methods except DBCD-S and DBCD-R use the decoupled quadratic approximation (4). For DBCD-R and DBCD-S, an additional factor of k comes in c_2 since we do k inner cycles of CDN in each iteration. HYDRA, PCD and DBCD-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 the magnitude of update or expected objective function decrease (DBCD-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 $c_4 = 0$[13]. The overall cost is $|S|$ to update the variables. For methods like DBCD-S, DBCD-R, PCD and PCD-S that do line-search, $c_3 = c_4 = \tau_{ls}$ where τ_{ls} is the average number of steps (α values tried) in one line search. 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. Since only one scalar needs to be communicated per line search step, the communication cost is dominated by the communication latency, i.e. the time taken to communicate zero bytes. As pointed out in Bian et al. (2013), τ_{ls} can increase with P; but it is still negligible compared to n. Combined with the fact that n is large in step (d), we will ignore this cost in the subsequent analysis.

Step d: This step involves computing and doing *AllReduce* on updated local predictions to get the global prediction vector for the next iteration and is common for all the methods. Note that because we are dealing with linear models, the updated predictions need to be communicated only once in each iteration even for the methods like ours that require line search, i.e., there is no need to communicate the updated predictions again and again for every line search step in each iteration.

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 Takáč (2015). For illustration, take the case of $|S| = P$, i.e., one variable is updated per node per iteration. For large P, $C^P \approx T^P C^P_{\text{comm}} = T^P\ \beta n$; 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 $speedup = \frac{T^1}{T^P}\frac{C^1}{C^P} = \frac{T^1}{T^P}\frac{\frac{nz}{m}}{\beta n}$. Richtárik and Takáč (2015) 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.* Given this, we ask how various choices in the steps of Algorithm 1 can be made to decrease C^P. Suppose we devise choices such that (a) C^P_{comp} is increased while still remaining in the zone where $C^P_{\text{comp}} \ll C^P_{\text{comm}}$, and (b) in the process, T^P is decreased greatly, then C^P can be decreased. The basic idea of our method is to use a more complex f^t_p than the simple quadratic in (4), due to which, T^P becomes much smaller. The use of line search, **(c.3)** for step **c** aids this further. We see

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

in table 5 that, DBCD-R and DBCD-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. In section 6, we will see on various datasets how, 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. These will become amply clear in section 6; see, for example, table 6.3 in that section.

6. Experimental Evaluation

In this section, we present experimental results on real-world datasets for the training of l_1 regularized linear classifiers using the squared hinge loss. Here training refers to the minimization of the function F in (1). We compare our methods with several state of the art methods, in particular, those analyzed in section 5 (see the methods in the first column of table 5) together with ADMM, the accelerated alternating direction method of multipliers (Goldstein et al., 2014). 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 four datasets: KDD, URL, ADS and WEBSPAM[14]. The key properties of these datasets are given in table 6.1. These datasets have a large number of features and l_1 regularization is important. The number of examples is large for KDD, URL and ADS. WEBSPAM has a much smaller number of examples and hence communication costs are low for this dataset.

Dataset	n	m	nz	$s = nz/m$
KDD	8.41×10^6	20.21×10^6	0.31×10^9	15.34
URL	2.00×10^6	3.23×10^6	0.22×10^9	68.11
ADS	18.56×10^6	0.20×10^6	5.88×10^9	29966.83
WEBSPAM	0.26×10^6	16.60×10^6	0.98×10^9	58.91

Table 4: Properties of datasets. n is the number of examples, m is the number of features, nz is the number of non-zero elements in the data matrix, and s is the average number of non-zero elements per feature.

Methods and Metrics: We evaluate the performance of all the methods using (a) Area Under Precision-Recall Curve (AUPRC) (Sonnenburg and Franc, 2010; Agarwal et al.,

14. KDD, URL and WEBSPAM are popular benchmark datasets taken from `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`. ADS is a proprietary dataset from Microsoft.

2013)[15] and (b) Relative Function Value Difference (RFVD) as a function of time taken. RFVD is computed as $\frac{F(w^t)-F^*}{F^*}$ where F^* is taken as the best value obtained across the methods after a long duration. We also report per node computation time statistics and sparsity pattern behavior of all the methods.

Parameter Settings: For each dataset we used cross validation to find the optimal λ value that gave the best AUPRC values. For each dataset we experimented with a range of λ values centred around the optimal value that have good sparsity variations over the optimal solution. Since the relative performance between methods was quite consistent across different λ values, we give details of the performance only for the optimal λ value. With respect to algorithm 1, the working set size (WSS) per node and the number of nodes (P) are common across all the methods. We set WSS in terms of the fraction (r) of the number of features per node, i.e., WSS=rm/P. Note that WSS will change with P for a given fraction r. For all datasets we give results for two r values $(0.01, 0.1)$. Note that r does not play a role in ADMM since all variables are optimized in each node. We experimented with $P = 25, 100$. Only for *ADS* dataset we used $P = 100, 200$ because it has many more examples than others.

Platform: We ran all our experiments on a Hadoop cluster with 379 nodes and 10 Gbit interconnect speed. Each node has Intel (R) Xeon (R) E5-2450L (2 processors) running at 1.8 GHz and 192 GB RAM. (Though 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 (Agarwal et al., 2013) on Hadoop. We implemented the pipelined *AllReduce* operation described in Agarwal et al. (2013) that reduces the communication cost from $\beta nlogP$ to βn for large n.

6.2 Method Specific Parameter Settings

We discuss method specific parameter setting used in our experiments and associated practical implications.

Choice of μ and k for DBCD: 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 side 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 (11) by simply using a fixed number of cycles of coordinate descent to minimize f_p^t. The right side plot in figure 1 shows the effect of number of cycles, k. We found that $k = 5, 10$ are good choices. Since computations are heavier for DBCD-S, we used $k = 5$ for it and used $k = 10$ for DBCD-R.

Let us begin with ADMM. We use the feature partitioning formulation of ADMM described in subsection 8.3 of Boyd et al. (2011). ADMM does not fit into the format of algorithm 1, but the communication cost per outer iteration is comparable to the other methods that fit into algorithm 1. In ADMM, the augmented Lagrangian parameter (ρ) plays an important role in getting good performance. In particular, the number of iterations required by ADMM for convergence is very sensitive with respect to ρ. While many schemes have been discussed in the literature (Boyd et al., 2011) we found that selecting ρ using the objective function value gave a good estimate; we selected ρ^* from a handful of ρ

15. We employed AUPRC instead of AUC because it differentiates methods more finely.

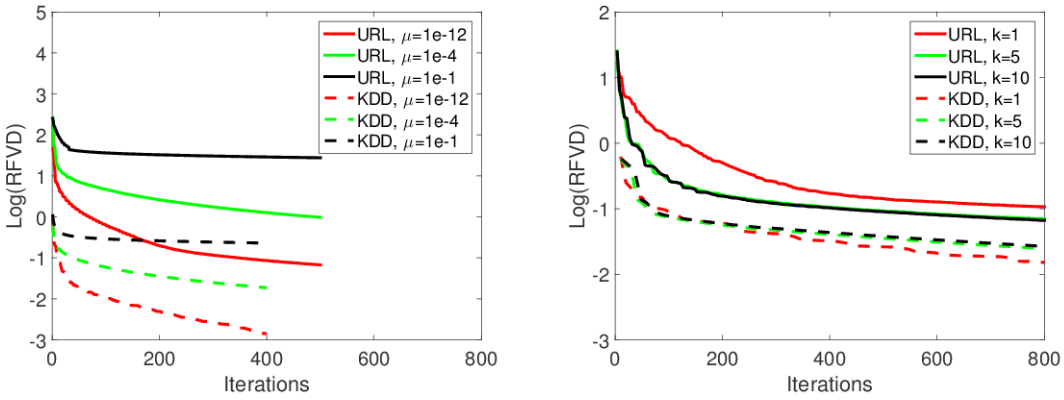


Figure 1: Study of μ and k on *KDD* and *URL*. Left: the effect of μ. Right: the effect of k, the number of cycles to minimize f_p^t. $\mu = 10^{-12}$ and $k = 10$ are good choices. $P = 100$.

values with ADMM run for 10 iterations (i.e., not full training) for each ρ value tried.[16] However, this step incurred some computational/communication time. Note that each ADMM iteration optimizes all variables and involves many inner iterations, thus causing even the ten iterations each for several ρ values to be significantly large. In our time plots shown later, the late start of ADMM results is due to this cost. Note that this minimal number of ten iterations was essential to get a decent ρ^*.

Now consider GROCK, FPA and HYDRA which are based on using Lipschitz constants (L_j). We found GROCK to be either unstable and diverging or extremely slow. The left side 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 is that setting right values for these parameters to get good performance can be tricky and highly dataset dependent. The right side plot in figure 2 shows the extremely slow convergence behavior of FPA; its objective function also shows a non-monotone behavior. Therefore, we do not include GROCK and FPA further in our study.

For HYDRA we tuned L_j as follows. We set the first value of L_j to the theoretical default value proposed in Richtárik and Takáč (2016) and decreased it by a factor of $\beta = 2$ each time to create five values for L_j. Then we ran 25 iterations of HYDRA for each of those five values to chose the best value for L_j and then used that value for all remaining iterations. We found that this simple procedure was sufficient to arrive at a near-best single value for L_j. Unlike ADMM, the cost of this tuning step is negligible compared to the overall cost. Richtárik, and Takáč (Richtárik and Takáč, 2016) also showed results with the asynchronous implementation. For fair comparison with other synchronous approaches, we show results with the synchronous implementation only. Extending our work to the asynchronous setting and comparing with the asynchronous variants of other algorithms is an interesting future work.

16. These initial "tuning" iterations are not counted against the limit of 800 we set for the number of iterations. Thus, for ADMM, the total number of iterations can go higher than 800.

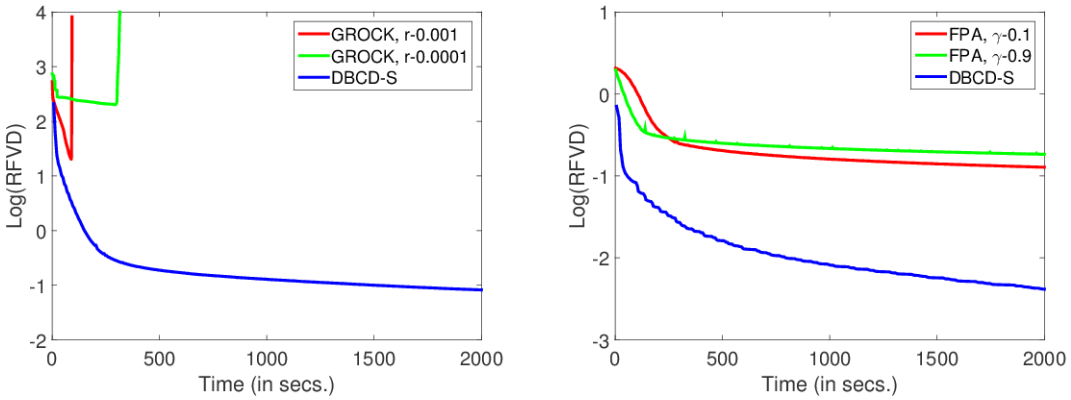


Figure 2: Left: Divergence and slow convergence of GROCK on the URL dataset ($\lambda = 2.4 \times 10^{-6}$ and $P = 25$). Right: Extremely slow convergence of FPA on the KDD dataset ($\lambda = 4.6 \times 10^{-7}$ and $P = 100$).

6.3 Performance Evaluation

We begin by comparing the efficiency of various methods and demonstrating the superiority of the new methods that were developed in section 4 and motivated in section 5. After this we analyze and explain the reasons for the superiority.

Study on AUPRC and RFVD: We have compared the performance of all methods by studying the variation of AUPRC and RFVD as a function of time, for various choices of λ, r (note that r defines the working set size, WSS=rm/P) and the number of nodes (P). To avoid cluttering with too many plots, we provide only representative ones - for each dataset, we choose one value for λ and two values each, for r and P.

Figures 3-6 show the RFVD versus time plots for the four datasets; note the use of log scale for RFVD in those plots. Figures 7-10 show the AUPRC versus time plots. The following observations can be made from these plots.

Superior performance of DBCD-S. In most cases DBCD-S is the best performer. In several of these cases, DBCD-S beats other methods very clearly; for example, on URL with $P = 100$ and $r = 0.01$ (see the bottom left plot of figure 4), the time needed to reach log RFVD=-0.5 is many times smaller than any other method. As another example, with KDD and $r = 0.01$ (see the left side plots in figure 3), if we set the log RFVD value to -2 as the stopping criterion, DBCD-S and PCD-S are faster than all other methods by an order of magnitude. Even in cases where DBCD-S is not the best (e.g., the case in the bottom left plot of figure 5), DBCD-S performs quite close to the best method.

How good is PCD-S? Recall from subsection 4.5 that PCD-S is the variation of the PCDN method Bian et al. (2013) using the S-scheme for variable selection. In some cases such as the last one pointed out in the previous paragraph, PCD-S gives an excellent performance. However, in many other cases, PCD-S does not perform well. This shows that working with quadratic approximations (like PCD-S does) can be quite inferior compared to using the actual nonlinear objective like DBCD-S does.

S-scheme versus R-scheme. In general, the S-scheme of selecting variables (namely, DBCD-S and PCD-S) performs much better than the R-scheme (namely, DBCD-R and

PCD-R), Only on WEBSPAM, PCD-R does slightly better than PCD-S in some cases. One possible reason for this is that WEBSPAM has a small number of examples, causing the communication cost to be much lower than the computation cost; note that the S-scheme requires more computation than the R-scheme.

Effect of r. The choice of r has an effect on the speed of various methods. But the sensitivity is not great. For DBCD-S, a reasonable choice is $r = 0.1$.

Performance of HYDRA. Though HYDRA has a good rate of descent in the objective function during the very early stages, it becomes quite slow soon after, leading to inferior performance. This shows up clearly even in the AUPRC plots.

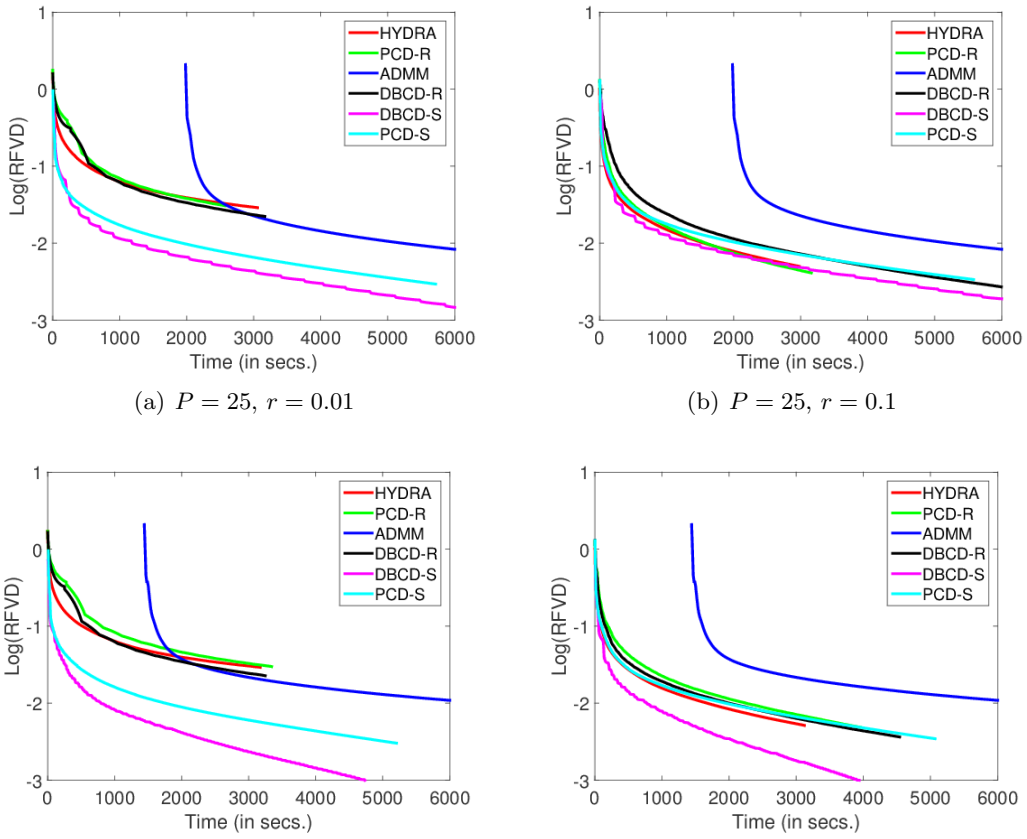
Performance of ADMM. First note that ADMM is independent of r since all the variables are updated. ADMM has a late start due to the time needed for tuning the augmented lagrangian parameter, ρ. (In some cases - see the top two plots in figure 6 - the ADMM curves are not even visible due to the initial tuning cost being relatively large.) Unfortunately, this tuning step is unavoidable; without it, ADMM's performance will be adversely affected. In many cases, DBCD-S reaches excellent solution accuracies even before ADMM begins making any progress.

Consistency between RFVD and AUPRC plots. On KDD, URL and ADS datasets there is good consistency between the two sets of plots. For example, the clear superiority of DBCD-S seen in the top left RFVD plot of figure 4 is also seen in the top left AUPRC plot of figure 8. Only on WEBSPAM (see figure 6 and figure 10) the two sets of plots have some inconsistency; in particular, note that, in figure 6, the initial decrease of the objective function is faster for HYDRA than DBCD-S, while, in figure 10, DBCD-S shows better initial increase in AUPRC than HYDRA. This happens because DBCD-S makes many more variables non-zero and touches many more examples than HYDRA in the initial steps.

Overall, the results point to the choice of DBCD-S as the preferred method as it is highly effective with an order of magnitude improvement over existing methods in many cases. Let us now analyze the reason behind the superior performance of DBCD-S. It is very much along the motivational ideas laid out in section 5: since communication cost dominates computation cost in each outer iteration, DBCD-S reduces overall time by decreasing the number of outer iterations.

Study on the number of outer iterations: We study T^P, the number of outer iterations needed to reach log RFVD$\leq \tau$. Table 6.3 gives T^P values for various methods in various settings. DBCD-S clearly outperforms other methods in terms of having much smaller values for T^P. PCD-S is the second best method, followed by ADMM. The solid reduction of T^P by DBCD-S validates the design that was motivated in section 5. The increased computation associated with DBCD-S is immaterial; because communication cost overshadows computation cost in each iteration for all methods, DBCD-S is also the best in terms of the overall computing time. The next set of results gives the details.

Computation and Communication Time: As emphasized earlier, communication plays an important role in the distributed setting. To study this effect, we measured the computation and communication time separately at each node. Figure 11 shows the computation time per node on the *KDD* dataset. In both cases, ADMM incurs significant computation time compared to other methods. This is because it optimizes over all variables in each node. DBCD-S and DBCD-R come next because our method involves both line search



(a) $P = 25$, $r = 0.01$

(b) $P = 25$, $r = 0.1$

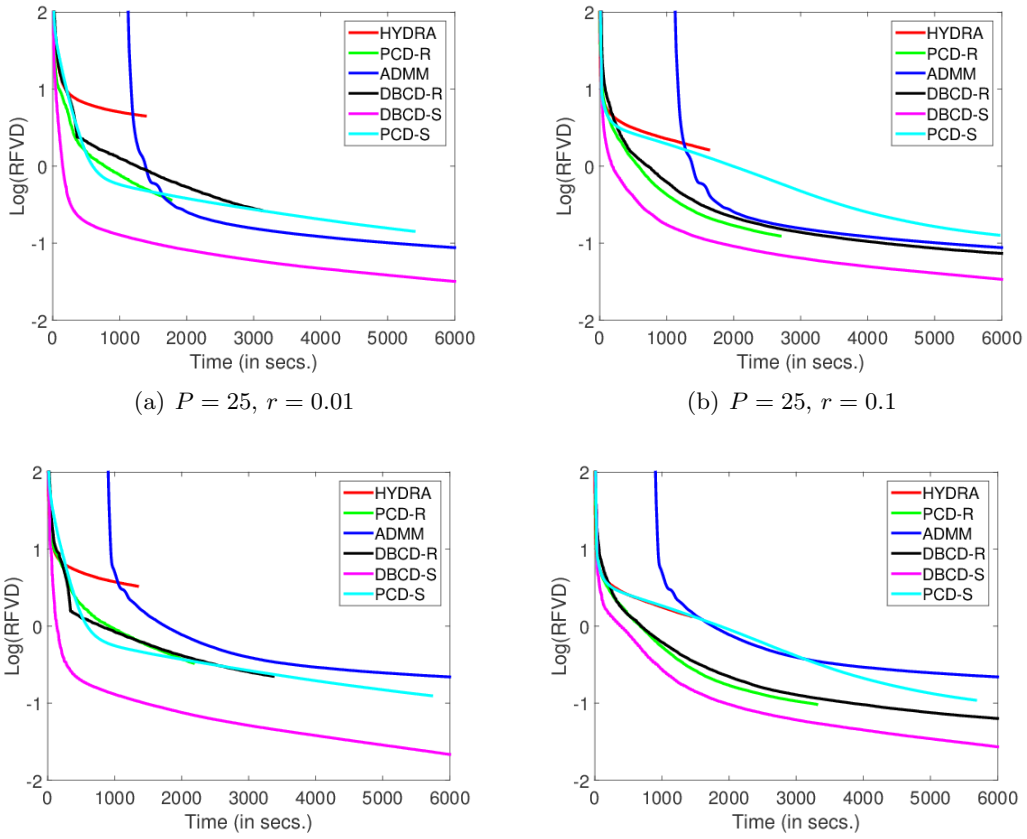
(c) $P = 100$, $r = 0.01$

(d) $P = 100$, $r = 0.1$

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

and 10 inner iterations. PCD-R and PCD-S take a little more time than HYDRA because of the line search. As seen in both DBCD 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 r settings. From table 6.3 (which gives representative results for one situation, KDD and $P = 25$), we see that the communication time dominates the cost in HYDRA and PCD-R. DBCD-R takes more computation time than PCD-R and HYDRA 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. DBCD-S takes the maximum computation time and is quite comparable to the communication time. Recall our earlier observation of DBCD-S giving order of magnitude speed-up in the overall time compared to methods such as HYDRA and PCD-R (see figures 3-10). Though the computation times taken by HYDRA, 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 DBCD. All these observations point to the fact our DBCD method nicely trades-off the computation versus communication cost,



(a) $P = 25$, $r = 0.01$

(b) $P = 25$, $r = 0.1$

(c) $P = 100$, $r = 0.01$

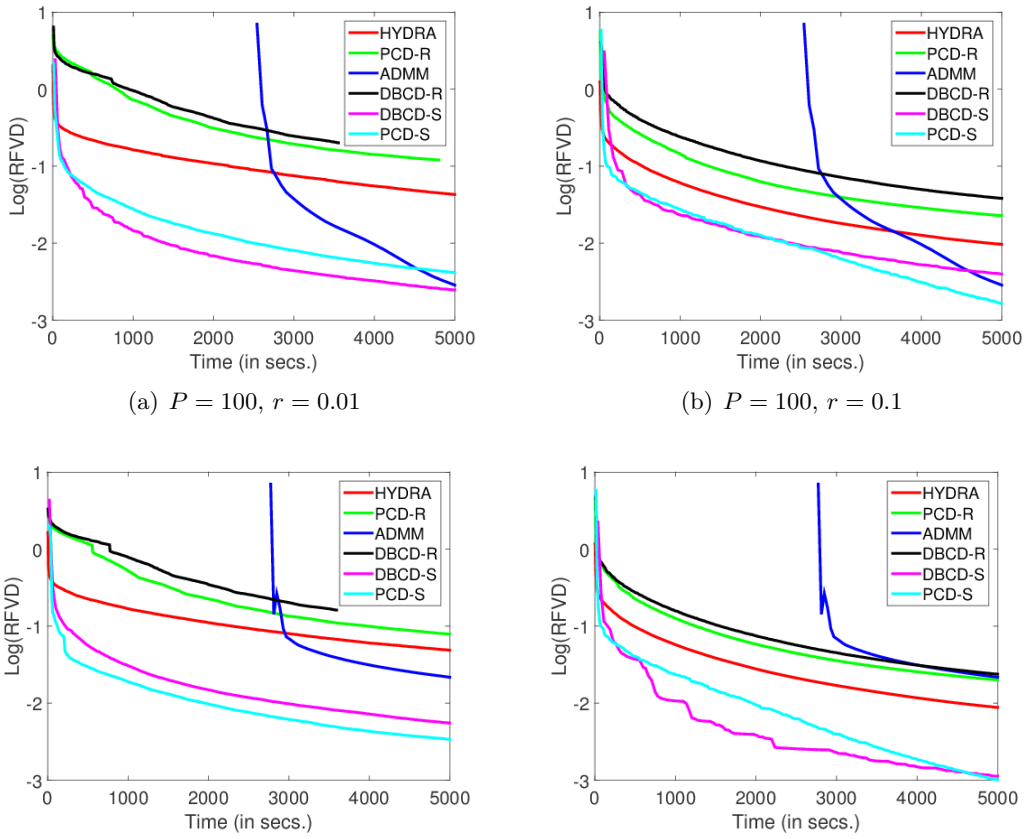
(d) $P = 100$, $r = 0.1$

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

and gives an excellent order of magnitude improvement in overall time. With the additional benefit provided by the S-scheme, DBCD-S clearly turns out to be the method of choice for the distributed setting.

The methods considered in this paper are all synchronous methods. Also, as l_1-regularization gives sparse solutions, load balancing can be prominent and lead to the "curse of last reducer" issue. The increase in waiting time (in our measurement, waiting time is counted as a part of the communication time) is higher for methods that involve greater computation; this is clear from table 6.3 where, roughly, communication time per iteration is higher for methods with higher computation time per iteration. In spite of this, our DBCD-S method, which has the largest computation and communication times per iteration, wins because of the drastically reduced number of iterations compared to other methods.

Sparsity Pattern: To study variable sparsity behaviors of various methods during optimization, we computed the percentage of non-zero variables (ρ) as a function of outer iterations. We set the initial values of the variables to zero. Figure 12 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.



(a) $P = 100$, $r = 0.01$ (b) $P = 100$, $r = 0.1$

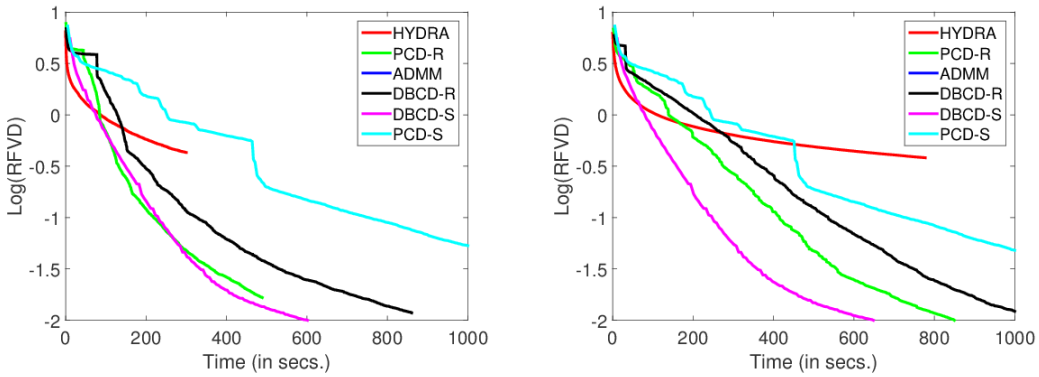
(c) $P = 200$, $r = 0.01$ (d) $P = 200$, $r = 0.1$

Figure 5: ADS dataset. Relative function value difference in log scale. $\lambda = 2.65 \times 10^{-6}$.

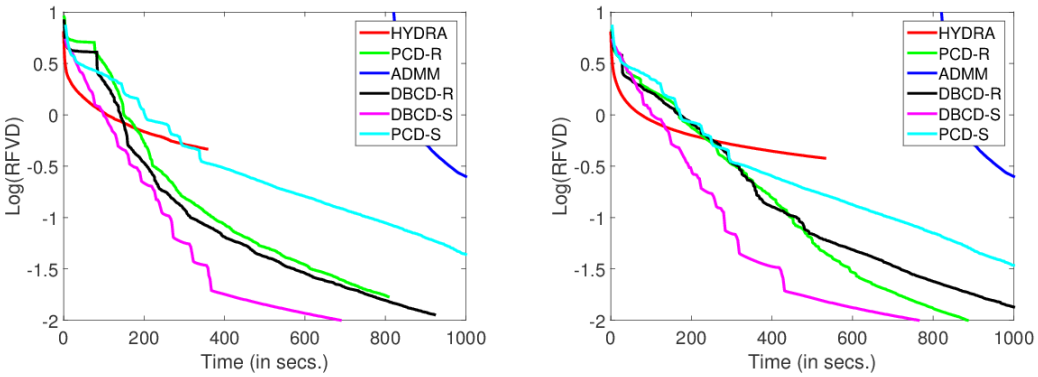
It is interesting to note that such an initial behavior seems necessary to make good progress in terms of both function value and AUPRC. In all the cases, many variables stay at zero after initial iterations; therefore, shrinking ideas (i.e., do not consider for selection those variables that tend to remain at zero) can be used to improve efficiency.

Remark on Speed up: Let us consider the RFVD plots corresponding to DBCD-S in figures 3 and 4. It can be observed that the times associated with $P = 25$ and $P = 100$ for reaching a certain tolerance, say log RFVD=-2, are close to each other. This means that using 100 nodes gives almost no speed up over 25 nodes, which may prompt the question: *Is a distributed solution really necessary?* There are two answers to this question. First, as we already mentioned, when the training data is huge[17] and so *the data is generated and 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. Second, for a given dataset, if the time taken to reach a certain optimality tolerance is plotted as a function of P, it may have a minimum at a value different from $P = 1$. In such a case, it is appropriate to choose a P (as well as r) optimally to minimize training time. Many applications involve

17. The KDD and URL datasets are really not huge in the *Big data* sense. In this paper we used them only because of lack of availability of much bigger public datasets.



(a) $P = 25$, $r = 0.01$ (b) $P = 25$, $r = 0.1$



(c) $P = 100$, $r = 0.01$ (d) $P = 100$, $r = 0.1$

Figure 6: WEBSPAM dataset. Relative function value difference in log scale. $\lambda = 3.92 \times 10^{-5}$

periodically repeated model training. For example, in Advertising, logistic regression based click probability models are retrained on a daily basis on incrementally varying datasets. In such scenarios it is worthwhile to spend time to tune parameters such as P and r in an early deployment phase to minimize time, and then use these parameter values for future runs.

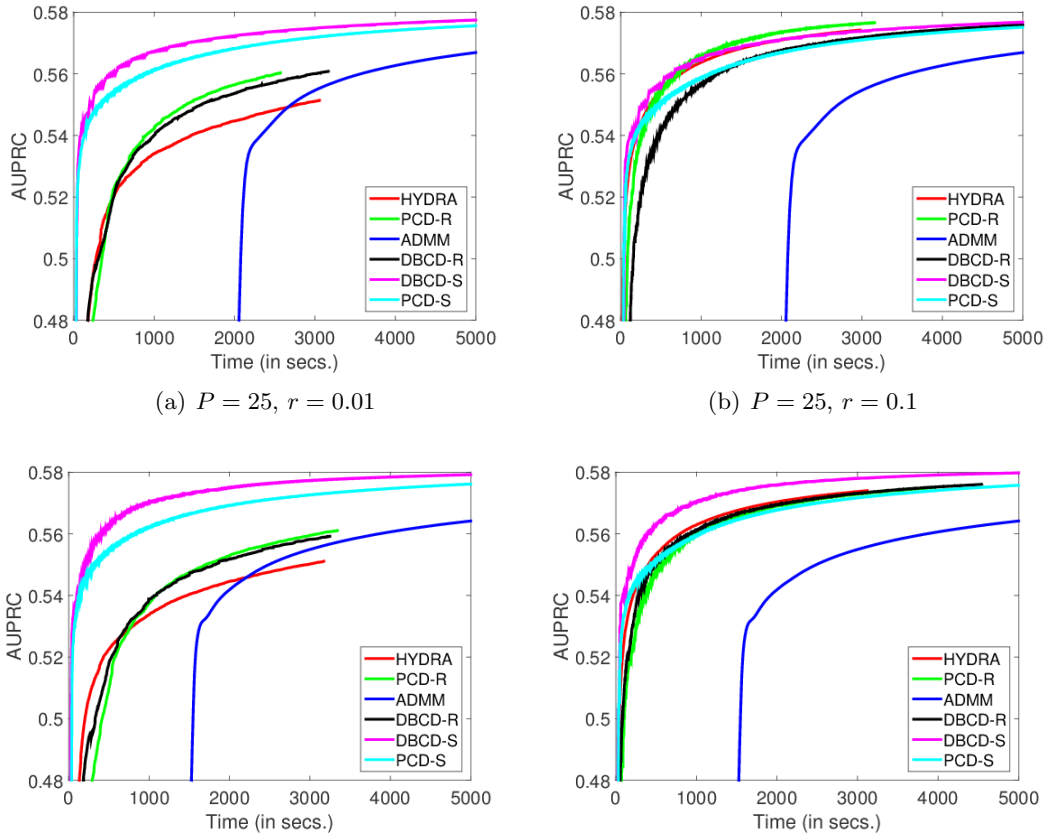
It is also important to point out that the above discussion is relevant to distributed settings in which communication causes a bottleneck. If communication cost is not heavy, e.g., when the number of examples is not large and/or communication is inexpensive such as in multicore solution, then good speed ups are possible; see, for example, the results in Richtárik and Takáč (2015).

7. Recommended DBCD algorithm

In section 4 we explored various options for the steps of Algorithm 1 looking beyond those considered by existing methods and proposing new ones, and empirically analyzing the various resulting methods in section 6. The experiments clearly show that DBCD-S is the best method. We collect full implementation details of this method in Algorithm 2.

Algorithm 2: Recommended DBCD algorithm

Parameters: Proximal constant $\mu > 0$ (Default: $\mu = 10^{-12}$);
WSS = # variables to choose for updating per node (Default: WSS=$r\,m/P$, $r = 0.1$);
k = # CD iterations to use for solving (3) (Default: $k = 10$);
Line search constants: $\beta, \sigma \in (0, 1)$ (Default: $\beta = 0.5$, $\sigma = 0.01$);
Choose w^0 and compute $y^0 = Xw^0$;
for $t = 0, 1 \ldots$ **do**
  for $p = 1, \ldots, P$ *(in parallel)* **do**
    (a) For each $j \in B_p$, solve (5) to get q_j. Sort $\{q_j : j \in B_p\}$ and choose WSS indices with least q_j values to form S_p^t;
    (b) Form $f_p^t(w_{B_p})$ using (6) and solve (3) using k CD iterations to get $\bar{w}_{B_p}^t$ and set direction: $d_{B_p}^t = \bar{w}_{B_p}^t - w_{B_p}^t$;
    (c) Compute $\delta y^t = \sum_p X_{B_p} d_{B_p}^t$ using AllReduce;
    (d) $\alpha = 1$;
    while *(12-13) are not satisfied* **do**
      $\alpha \leftarrow \alpha\beta$;
      Check (12)-(13) using $y + \alpha\,\delta y$ and aggregating the l_1 regularization value via AllReduce;
    end e
    Set $\alpha^t = \alpha$, $w_{B_p}^{t+1} = w_{B_p}^t + \alpha^t d_{B_p}^t$ and $y^{t+1} = y^t + \alpha^t\,\delta y^t$;
  end f
  Terminate if the optimality conditions (2) hold to the desired approximate level;
end



(a) $P = 25$, $r = 0.01$ (b) $P = 25$, $r = 0.1$

(c) $P = 100$, $r = 0.01$ (d) $P = 100$, $r = 0.1$

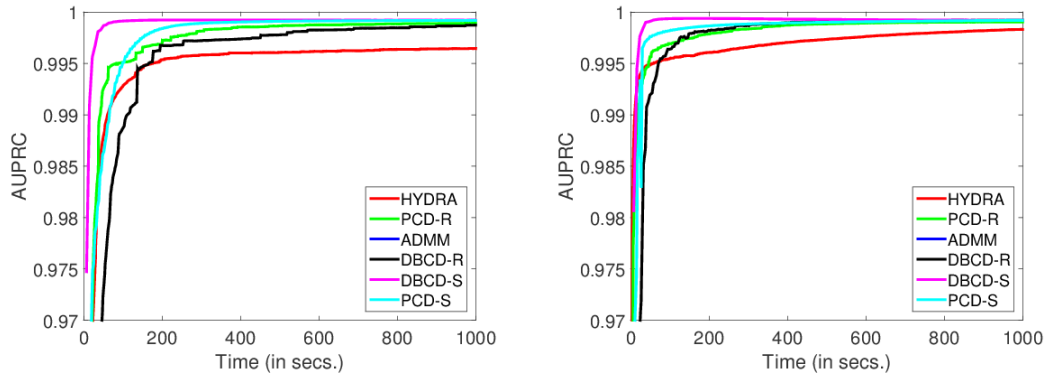
Figure 7: KDD dataset. AUPRC Plots. $\lambda = 4.6 \times 10^{-7}$

8. Conclusion

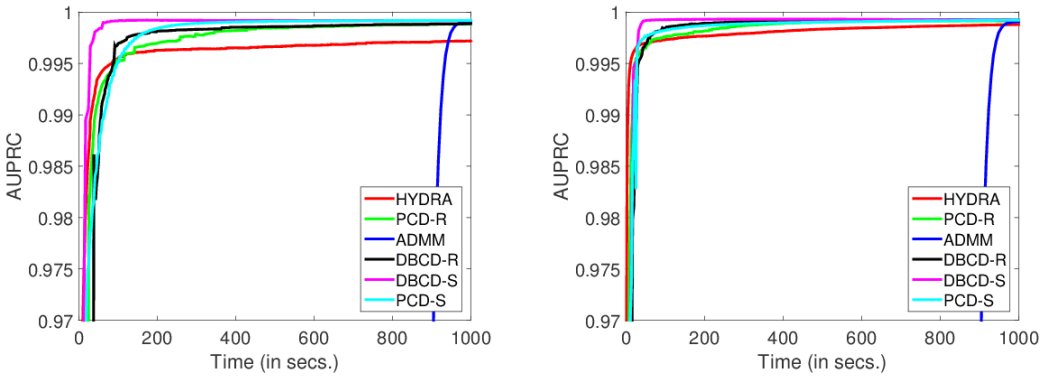
In this paper we have proposed a class of efficient block coordinate methods for the distributed training of l_1 regularized linear classifiers. In particular, the proximal-Jacobi approximation together with a distributed greedy scheme for variable selection came out as a strong performer. There are several useful directions for the future. It would be useful to explore other approximations such as block GLMNET and block L-BFGS suggested in subsection 4.2. Like Richtárik and Takáč (2015), 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.

Proof of Theorem 1

First let us write δ_j in (11) as $\delta_j = E_{jj} d_j^t$ where $E_{jj} = \delta_j / (d_{B_p}^t)^j$. Note that $|E_{jj}| \leq \mu/2$. Use the condition (14) in Condition 2 with $w_{B_p} = \bar{w}_{B_p}^t$ and $\hat{w}_{B_p} = w_{B_p}$ in (11) together



(a) $P = 25$, $r = 0.01$ (b) $P = 25$, $r = 0.1$



(c) $P = 100$, $r = 0.01$ (d) $P = 100$, $r = 0.1$

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

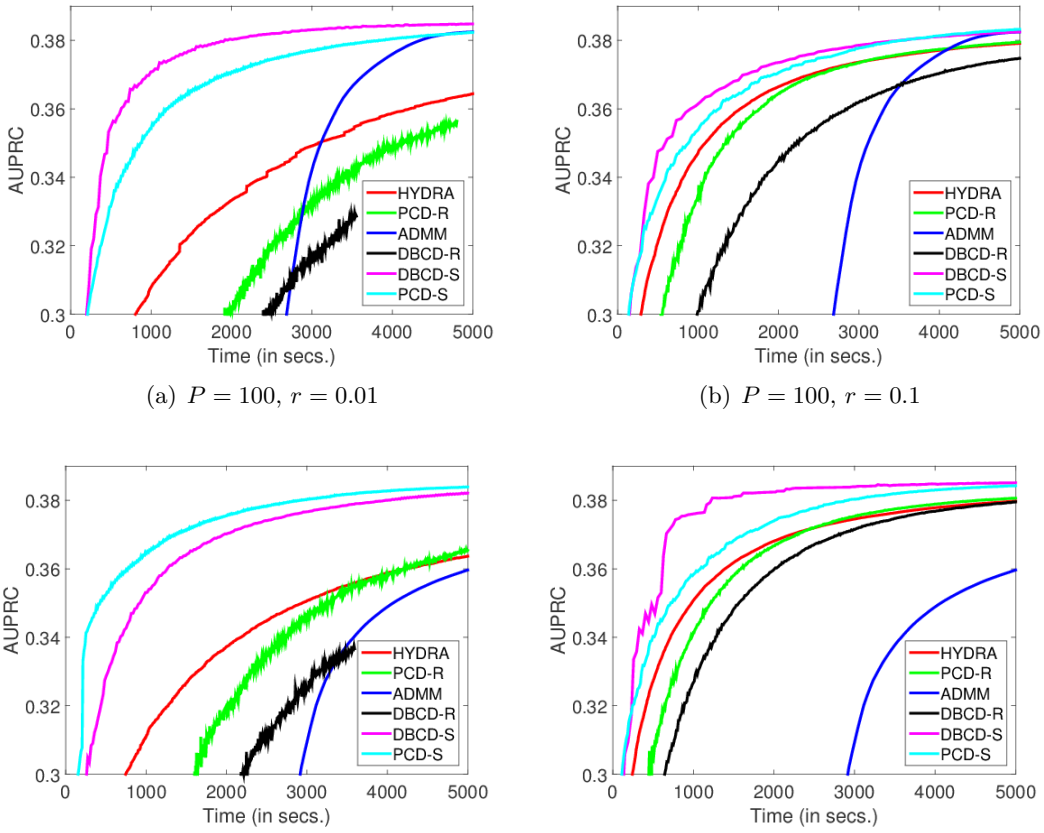
with the gradient consistency property of Condition 1 to get

$$g^t_{S^t_p} + H^t_{S^t_p} d^t_{S^t_p} + \xi_{S^t_p} = 0, \tag{15}$$

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 $\hat{H} \geq \mu I$ and $|E_{jj}| \leq \mu/2$, we get $H^t_{S^t_p} \geq \frac{\mu}{2} I$. 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 (15) is the optimality condition for the quadratic minimization,

$$d^t_{B_p} = \arg\min_{d_{B_p}} \; (g^t_{B_p})^T d_{B_p} + \frac{1}{2} (d_{B_p})^T H_{B_p} d_{B_p} + \sum_{j \in B_p} \lambda \, |w^t_j + d_j| \;\; \text{s.t.} \;\; d_j = 0 \; \forall \; j \in B_p \setminus S^t_p \tag{16}$$



(a) $P = 100$, $r = 0.01$ (b) $P = 100$, $r = 0.1$

(c) $P = 200$, $r = 0.01$ (d) $P = 200$, $r = 0.1$

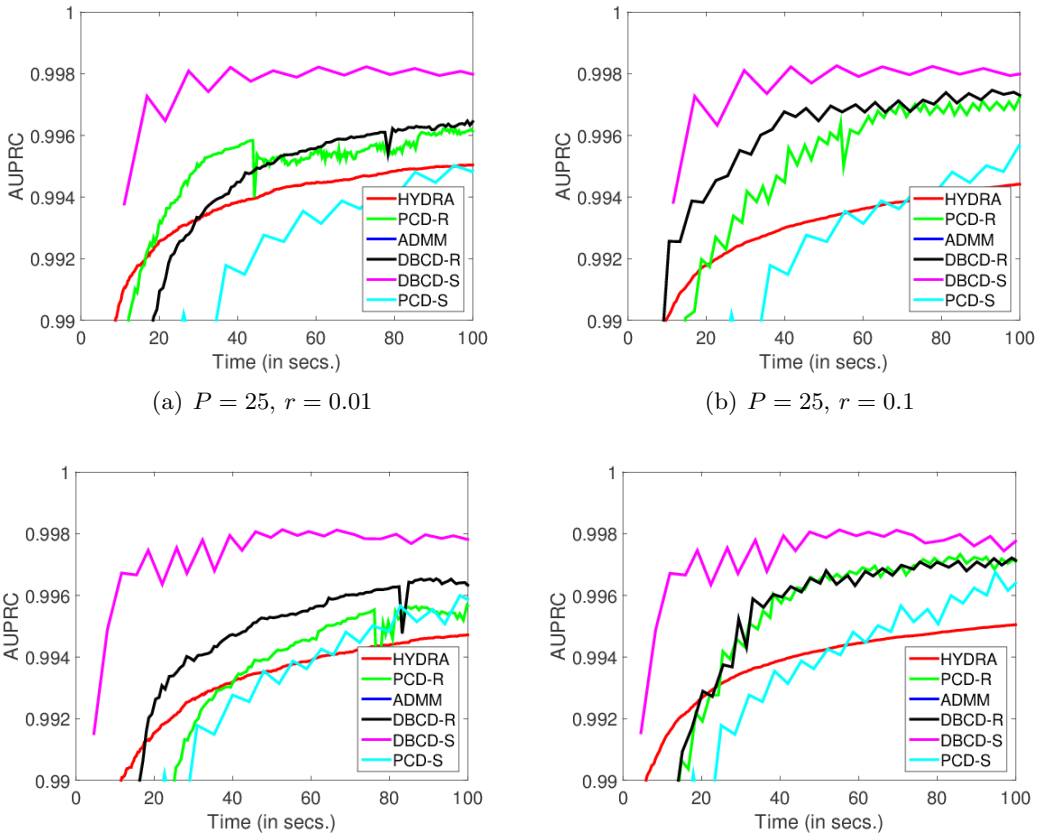
Figure 9: ADS dataset. AUPRC Plots. $\lambda = 2.65 \times 10^{-6}$

Combined over all p,

$$d^t = \arg\min_d \ (g^t)^T d + \frac{1}{2} d^T H d + u(w^t + d)$$
$$\text{s.t. } d_j = 0 \ \forall \ j \in \cup_p (B_p \setminus S_p^t) \tag{17}$$

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 (Tseng and Yun, 2009).

The line search condition (12)-(13) is a special case of the line search condition in Tseng and Yun (2009). The Gauss-Seidel scheme of subsection 4.1 is an instance of the Gauss-Seidel scheme of Tseng and Yun (2009). Now consider the distributed greedy scheme in subsection 4.1. Let $j_{\max} = \arg\max_{1 \le j \le m} \bar{q}_j$. By the way the S_p^t are chosen, $j_{\max} \in \cup_p S_p^t$. Therefore, $\sum_{j \in \cup_p S_p^t} \bar{q}_j \le \frac{1}{m} \sum_{j=1}^m \bar{q}_j$, thus satisfying the Gauss-Southwell-q rule condition of Tseng and Yun (2009). Now Theorems 1-4 of Tseng and Yun (2009) can be directly applied to prove our Theorem 1. Note that Theorem 4 of Tseng and Yun (2009) ensures that Assumption 2(a) of that paper holds; Assumption 2(b) of that paper trivially holds because, in our case F is convex.



(a) $P = 25$, $r = 0.01$ (b) $P = 25$, $r = 0.1$

(c) $P = 100$, $r = 0.01$ (d) $P = 100$, $r = 0.1$

Figure 10: WEBSPAM dataset. AUPRC Plots. $\lambda = 3.92 \times 10^{-5}$. Because the initial ρ tuning time for ADMM is large, its curves are not seen in the shown time window of 0-100 secs.

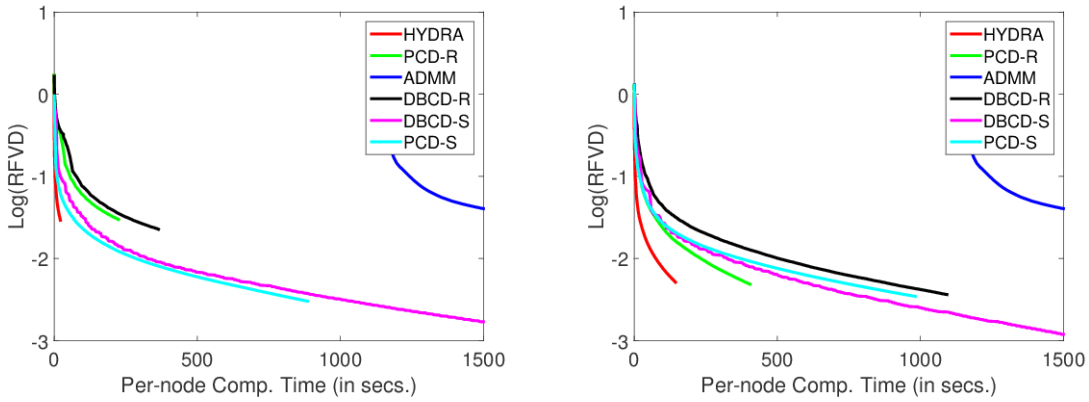
References

A. Agarwal, O. Chapelle, M. Dudik, and J. Langford. A reliable effective terascale linear learning system. *JMLR*, 15:1111–1133, 2013.

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

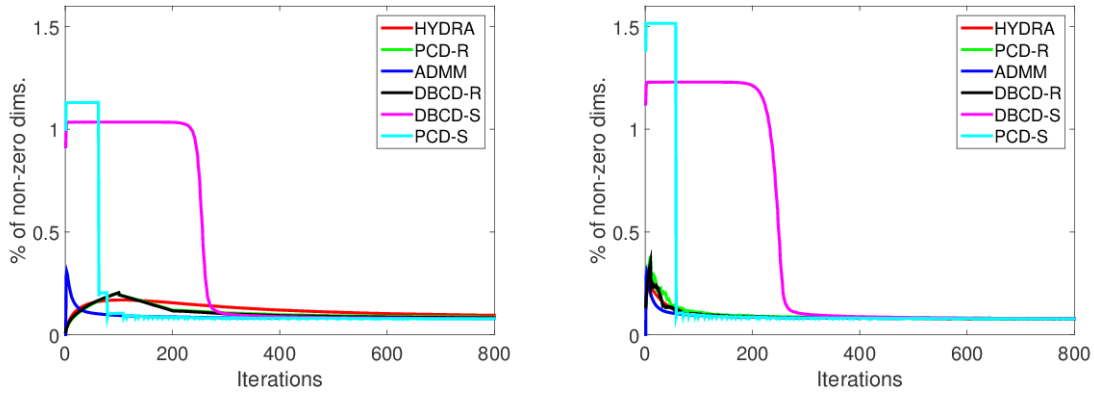
S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, pages 1–122, 2011.

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



(a) $r = 0.01$ (b) $r = 0.1$

Figure 11: Per-node computation time on the KDD dataset ($\lambda = 4.6 \times 10^{-7}$ and $P = 100$).



(a) $r = 0.01$ (b) $r = 0.1$

Figure 12: KDD dataset: Typical behavior of the percentage of non-zero variables.

W. Deng and W. Yin. On the global and linear convergence of the generalized alternating direction method of multipliers. *Journal of Scientific Computing*, pages 889–916, 2016.

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.

F. Facchinei, S. Sagratella, and G. Scutari. Flexible parallel algorithms for big data optimization. *ICASSP*, 2014.

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.

T. Goldstein, O'Donoghue, S. Setzer, and R. Baraniuk. Fast alternating direction optimization methods. *SIAM Journal on Imaging Sciences*, 7:1588–1623, 2014.

M. Jaggi, V. Smith, M Takáč, J. Terhorst, S. Krishnan, T. Hofmann, and M.I. Jordan. Communication-efficient distributed dual coordinate ascent. *NIPS*, 2014.

M. Li, D.G. Andersen, J.W. Park, A.J. Smola, A. Ahmed, J. Josifovski, J. Long, E.J. Shekita, and B. Su. Scaling distributed machine learning with the parameter server. *OSDI*, 2014.

C. Ma, V. Smith, M. Jaggi, M.I. Jordan, M Takáč, and P. Richtárik. Adding vs. averaging in distributed primal-dual optimization. *ICML*, 2015.

I. Necoara and D. Clipici. Efficient parallel coordinate descent algorithm for convex optimization problems with separable constraints: application to distributed MPC. *Journal of Process Control*, 23:243–253, 2014.

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

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

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

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

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

Z. Peng, M. Yan, and W. Yin. Parallel and distributed sparse optimization. *IEEE Asilomar Conference on Signals, Systems, and Computers*, 2013.

C. Ravazzi, S. M. Fosson, and E. Magli. Distributed soft thresholding for sparse signal recovery. *Proceedings of IEEE Global Communications Conference*, 2013.

P. Richtárik and M Takáč. Iteration complexity of randomized block-coordinate descent methods for minimizing a composite function. *Mathematical Programming*, 144:1–38, 2014.

P. Richtárik and M. Takáč. Parallel coordinate descent methods for big data optimization. *Mathematical Programming*, 2015.

P. Richtárik and M. Takáč. Distributed coordinate descent method for learning with big data. *JMLR*, 17:1–25, 2016.

C. Scherrer, M. Halappanavar, A. Tewari, and D. Haglin. Scaling up coordinate descent algorithms for large l_1 regularization problems. *ICML*, pages 1407–1414, 2012a.

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

S. Shalev-Shwartz and A. Tewari. Stochastic methods for $l1$ regularized loss minimization. *JMLR*, 12:1865–1892, 2011.

S. T. Sonnenburg and V. Franc. COFFIN: a computational framework for linear SVMs. *ICML*, 2010.

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

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.

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.

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

KDD, $\lambda = 4.6 \times 10^{-7}$

		Existing methods		Our methods		
P	τ	HYDRA	PCD-R	PCD-S	DBCD-R	DBCD-S
25	-1	298	294	12	236	**8**
	-2	> 800	> 800	331	> 800	**124**
	-3	> 800	> 800	> 800	> 800	**688**
100	-1	297	299	13	230	**10**
	-2	> 800	> 800	311	> 800	**137**
	-3	> 800	> 800	> 800	> 800	**797**

URL, $\lambda = 9.0 \times 10^{-8}$

		Existing methods		Our methods		
25	0	> 2000	878	201	770	**23**
	-0.5	> 2000	> 2000	929	1772	**42**
	-1	> 2000	> 2000	> 2000	> 2000	**216**
100	0	> 2000	840	197	476	**27**
	-0.5	> 2000	> 2000	858	1488	**55**
	-1	> 2000	> 2000	> 2000	> 2000	**287**

WEBSPAM, $\lambda = 3.9 \times 10^{-5}$

		Existing methods		Our methods		
25	0	521	202	56	169	**13**
	-0.5	> 1500	185	109	128	**25**
	-1.0	> 1500	532	180	439	**39**
100	0	511	203	56	180	**28**
	-0.5	> 1500	308	111	265	**49**
	-1.0	> 1500	525	235	403	**74**

ADS, $\lambda = 2.6 \times 10^{-6}$

		Existing methods		Our methods		
100	-1.0	573	> 600	10	> 600	**7**
	-1.5	> 600	> 600	61	> 600	**18**
	-2.0	> 600	> 600	169	> 600	**53**
200	-1.0	569	> 600	10	> 600	**11**
	-1.5	> 600	> 600	35	> 600	**46**
	-2.0	> 600	> 600	154	> 600	**138**

Table 5: T^P, the number of outer iterations needed to reach log RFVD$\leq \tau$, for various P and τ values. We set $r = 0.01$. Best values are indicated in boldface. For each dataset, the τ values were chosen to cover the region where AUPRC values are in the process of reaching the steady state value. The methods were terminated when the number of iterations exceeded a maximum; this maximum was set differently for different datasets.

KDD, $\lambda = 4.6 \times 10^{-7}$

Method	Comp.	Comm.	Comp.	Comm.
	$r = 0.01$		$r = 0.1$	
HYDRA	0.033	1.665	0.250	1.404
PCD-R	0.125	1.955	0.360	2.093
PCD-S	1.483	3.046	1.598	2.703
DBCD-R	0.311	2.173	1.733	2.396
DBCD-S	4.426	2.263	5.317	2.387

URL, $\lambda = 9.0 \times 10^{-8}$

Method	Comp.	Comm.	Comp.	Comm.
	$r = 0.01$		$r = 0.1$	
HYDRA	0.018	0.443	0.123	0.418
PCD-R	0.033	0.834	0.126	1.157
PCD-S	0.780	1.527	0.915	1.606
DBCD-R	0.124	1.380	0.741	1.956
DBCD-S	3.591	2.438	4.151	1.731

WEBSPAM, $\lambda = 3.9 \times 10^{-5}$

Method	Comp.	Comm.	Comp.	Comm.
	$r = 0.01$		$r = 0.1$	
HYDRA	0.029	0.137	0.284	0.146
PCD-R	0.022	0.374	0.201	0.973
PCD-S	1.283	1.389	1.156	1.498
DBCD-R	0.121	0.535	1.046	0.995
DBCD-S	2.254	1.089	2.559	1.080

ADS, $\lambda = 2.6 \times 10^{-6}$

Method	Comp.	Comm.	Comp.	Comm.
	$r = 0.01$		$r = 0.1$	
HYDRA	0.133	3.652	0.695	4.267
PCD-R	0.198	4.208	0.880	5.784
PCD-S	1.636	6.326	2.220	7.888
DBCD-R	0.197	5.923	0.965	11.05
DBCD-S	1.244	8.285	2.709	13.8

Table 6: Computation and communication costs per iteration (in secs.) for KDD, $P = 25$.