# Naive Parallelization of Coordinate Descent Methods and an Application on Multi-core L1-regularized Classification

Yong Zhuang[*] Yuchin Juan[†] Guo-Xun Yuan[‡] Chih-Jen Lin[§]

**Abstract**

It is well known that a direct parallelization of sequential optimization methods (e.g., coordinate descent and stochastic gradient methods) is often not effective. The reason is that at each iteration, the number of operations may be too small. In this paper, we point out that because of the skewed distribution of non-zero values in real-world data sets, this common understanding may not be true if the method sequentially accesses data in a feature-wise manner. Because some features are much denser than others, a direct parallelization of loops in a sequential method may result in excellent speedup. This approach possesses an advantage of retaining all convergence results because the algorithm is not changed at all. We apply this idea on a coordinate descent (CD) method for L1-regularized classification, and explain why direct parallelization should work in practice. Further, an investigation on the shrinking technique commonly used to remove some features in the training process shows that this technique helps the parallelization of CD methods. Experiments indicate that a naive parallelization achieves better speedup than existing methods that laboriously modify the algorithm to achieve parallelism. Though a bit ironic, we conclude that the naive parallelization of the CD method is the best and the most robust multi-core implementation for L1-regularized classification.

## 1 Introduction

Among convex optimization methods for large-scale linear classification we can roughly categorize them to two types according to the amount of information accessed each time for updating the model. The first is "sequential methods," which at each step use either only one data point or one feature vector. The second type is "batch methods," which use more information (sometimes the whole set) at a time. We are interested in their parallelization in multi-core environments.

Examples of batch methods for linear classification include gradient descent, Newton methods [11], and quasi Newton methods [12]. It is often easy to parallelize such methods because each time a significant number of operations are assigned to a thread. Note that because of the overhead, paralleling few operations is not useful. For example, at each iteration of batch methods we must calculate

$$\boldsymbol{w}^T\boldsymbol{x}_i, \forall i,$$

where $\boldsymbol{x}_i, \forall i$ are training instances and $\boldsymbol{w}$ is the model. Not only can these independent inner products be conducted in parallel, but also each task (i.e., an inner product between two vectors) assigned to a thread involves a substantial number of operations. The direct parallelization of batch methods has been successfully reported in, for example, [10].

For sequential methods such as coordinate descent (CD) [3, 8, 9] or stochastic gradient (SG), the amount of information used at each step is much less than that in batch methods. Typically one instance or one feature is considered. Because a relatively smaller number of operations are conducted, the common understanding is that it is not effective to parallelize sequential methods like CD or SG. For example, assume the main task at a step is to calculate a single $\boldsymbol{w}^T\boldsymbol{x}_i$. We can use multiple threads for the inner product, but the speedup is often poor because very few operations are assigned to each thread. Therefore, instead of directly parallelizing the sequential methods, existing works often modify the algorithm so that a significant amount of operations can be conducted at a thread. For example, a mini-batch algorithm considers some instances or features at a time so parallel computation can be applied on them. However, with the change of algorithms, convergence and implementation issues must be carefully checked.

In this paper, we point out that the above conventional wisdom on sequential methods is not always true. The direct parallelization can achieve excellent speedup for some types of problems. We will see in Section 2 that because of the skewed distribution of non-zero elements in most real-world data sets, if an algorithm

[*]Carnegie Mellon University. yongzhua@andrew.cmu.edu. Most of the work done during the internship in Criteo Research.

[†]Criteo Research. yc.juan@criteo.com. Contribute equally with Yong Zhuang.

[‡]Facebook, Inc. gxyzuan@gmail.com

[§]National Taiwan University. cjlin@csie.ntu.edu.tw accesses a data set in a feature-wise manner, then by simply parallelizing the original algorithm, we can get a competitive or better speedup than some sophisticated modifications. Our finding, though very simple, is very useful in practice.

This paper is organized as follows. In Section 2, by detailed statistics we show that the distribution of non-zero values across features is very skewed. From this finding we conjecture that a sequential-type algorithm that accesses one feature at a time can be directly parallelized to achieve good speedup. In Section 3, we discuss CD for L1-regularized linear classification as an example. Further, an investigation on the shrinking technique commonly used to remove some features in the training process shows that this technique helps the parallelization of CD methods. Existing modifications to parallelize CD are reviewed in Section 4. Experiments in Section 5 show the effectiveness of naive parallelization – our strategy achieves better speedup than sophisticated modifications. Section 6 concludes our work.

Our proposed approach has been available in the multi-core extension of the package LIBLINEAR [6]: `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/multicore-liblinear`. Programs for experiments in this work can be found through the same web page.

## 2 Naive Parallelization and Distribution of Non-zeros

Roughly speaking, the running time of a multi-threading task is

$$\frac{\#\text{operations} \times \text{time per operation}}{\#\text{threads}} + \text{overhead}. \tag{2.1}$$

If the number of operations is small, the overhead may cause longer running time than that of using a single thread. Take an inner product as an example. While it is easy to split the computation to several independent sub-tasks, simple experiments show that we may need 500 components to make parallelization not harmful, and 100,000 components to reach the maximum speedup. Unfortunately, for large-scale linear classification on sparse data, an instance $\boldsymbol{x}_i$ may possess too few non-zero elements so that an inner product $\boldsymbol{w}^T\boldsymbol{x}_i$ cannot be effectively parallelized. This explains why traditionally CD and SG are considered not suitable for effective parallelization.

Interestingly, our finding is that the direct parallelization of each CD or SG step may not be entirely hopeless. In fact, for suitable algorithms, the speedup can be dramatic. We begin with the idea of running each CD or SG step in the following way:

```
if number of non-zeros ≥ a threshold then
    Run the step by multiple threads
else
    Run the step by a single thread
```

Clearly, we get good speedup only if a small subset of instances or features includes most non-zeros of the entire data set. That is, there are few dense features or instances, while all others are sparse. Intuitively, this situation should rarely happen, but to our surprise, it happens frequently. Subsequently we analyze all large and sparse binary classification problems in LIBSVM Data Sets.[1]

Assume a data set includes $l$ instances. We rearrange them so that

$$\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l$$

are in the descending order according to their number of non-zero entries. Then we investigate the distribution of non-zeros and obtain the following information.

- $a$: the subset $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_a\}$ contains 50% of all non-zero entries
- $b$: the subset $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_b\}$ contains 80% of all non-zero entries
- $\text{nnz}_a$: number of non-zero entries in $\boldsymbol{x}_a$
- $\text{nnz}_b$: number of non-zero entries in $\boldsymbol{x}_b$
- $\overline{\text{nnz}}_a$: average number of non-zeros in $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_a\}$
- $\overline{\text{nnz}}_b$: average number of non-zeros in $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_b\}$

If $n$ is the number of features, we can consider the data from a feature-wise setting to have

$$\bar{\boldsymbol{x}}_1, \ldots, \bar{\boldsymbol{x}}_n,$$

and obtain the same statistics. Table 1 presents all obtained values.

We observe a huge difference between instance-wise and feature-wise settings. From an instance-wise perspective in general the values $a/l$ and $b/l$ are respectively close to 0.5 and 0.8, so the number of non-zeros per instance does not vary much. Further, each instance in most problems has no more than a few hundred non-zeros. In contrast, from a feature-wise perspective, most non-zero elements are associated with a small subset of features. For example, in the `url_combined` data set, 0.00002% of features (i.e., around 60 out of more than three millions) contain half of all non-zero elements. Therefore, each of these "dense" features has many non-zeros. We plot the distribution of a real-world data set `criteo` in Figure 1. The distribution is extremely skewed under the feature-

[1] `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary`. We use data sets that have more than ten million non-zeros and density less than 0.001. We also add a non-public data set `yahoo-korea`.

| | Instance-wise | | | | | | Feature-wise | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Data set | $a/l$ | $b/l$ | $\text{nnz}_a$ | $\text{nnz}_b$ | $\overline{\text{nnz}}_a$ | $\overline{\text{nnz}}_b$ | $a/n$ | $b/n$ | $\text{nnz}_a$ | $\text{nnz}_b$ | $\overline{\text{nnz}}_a$ | $\overline{\text{nnz}}_b$ |
| `avazu-app` | 0.50 | 0.80 | 15 | 15 | 15 | 15 | 0.002 | 0.01 | 759,125 | 72,405 | 2,422,727 | 538,899 |
| `criteo` | 0.50 | 0.80 | 39 | 39 | 39 | 39 | 0.0001 | 0.002 | 1,114,789 | 40,477 | 4,150,931 | 456,739 |
| `kdd2010-a` | 0.40 | 0.73 | 37 | 30 | 44 | 39 | 0.0003 | 0.02 | 5,734 | 46 | 31,609 | 536 |
| `kdd2012` | 0.50 | 0.80 | 11 | 11 | 11 | 11 | 0.00003 | 0.005 | 45,151 | 331 | 456,195 | 5,154 |
| `rcv1_test` | 0.24 | 0.54 | 100 | 53 | 151 | 107 | 0.01 | 0.05 | 22,990 | 4,172 | 54,818 | 20,329 |
| `splice_site.t.10%` | 0.50 | 0.80 | 3,331 | 3,309 | 3,343 | 3,335 | 0.09 | 0.57 | 169 | 106 | 1,034 | 270 |
| `url_combined` | 0.44 | 0.76 | 113 | 105 | 130 | 121 | 0.00002 | 0.00006 | 2,264,387 | 115,088 | 2,360,644 | 1,204,444 |
| `webspam` | 0.29 | 0.55 | 4,910 | 3,570 | 6,451 | 5,383 | 0.006 | 0.02 | 98,573 | 16,401 | 165,841 | 74,478 |
| `yahoo-korea` | 0.20 | 0.48 | 502 | 265 | 857 | 571 | 0.0007 | 0.005 | 11,986 | 1,067 | 35,698 | 7,525 |

Table 1: The values of $a$, $b$, $\text{nnz}_a$, $\text{nnz}_b$, $\overline{\text{nnz}}_a$, $\overline{\text{nnz}}_b$ in selected data sets under instance-wise and feature-wise perspectives. Additional information of these sets is in Table 2.

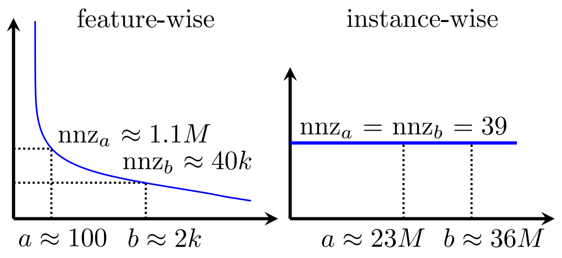

Figure 1: The distribution of non-zero elements in `criteo`. $x$-axis of left: $\{1, \ldots, n\}$, right: $\{1, \ldots, l\}$; $y$-axis of left: log scaled, right: linear scaled.

wise setting but is uniform under the instance-wise setting. We explain why the difference may commonly happen in practice. For `criteo`, as a data set for CTR (click-through rate) prediction, it may contain features such as one "device type" and one "device id." Because the same type of devices is used by many people, this feature has a large number of non-zero values. In contrast, "device id" is the identifier of a user device, so it may correspond to very few data instances. With these sparse features we have a long tail of the distribution. In contrast, an instance often corresponds to only one "device type" and one "device id." Thus each instance has very few non-zeros and the number of non-zeros is similar (or exact the same). The same situation happens for `webspam`, which is a document set generated by the bag-of-words setting. From a feature-wise perspective, frequent words may occur in millions of documents and rare words may only appear in tens of documents. On the other hand, from an instance-wise perspective, for documents collected from the same or similar sources their numbers of words may not significantly vary.

The above discussion indicates that if a sequential-type algorithm processes a feature at a time and the main task is to go over the feature's non-zero entries, then a naive parallelization can be very useful. The reason is that most operations are associated with those super-dense features and can be easily parallelized. On the other hand, for an algorithm processing an instance at a time, because the number of non-zeros is small, a naive parallelization may not be effective.

For `splice_site.t.10%`, it seems that even with feature-wise data accesses, direct parallelization may not be effective because $\text{nnz}_a$ and $\text{nnz}_b$ are both small. However, for L1-regularized classification, we show in Section 5.2 that the speedup is still good because a technique called "shrinking" to remove sparse features helps to make direct parallelization effective.

For dense data, the effectiveness of parallelization under a feature-wise setting depends on the value of $l$ (the number of instances). If a data set has enough instances, then the parallelization should be effective.

## 3 CD for L1-regularized Linear Classification

Based on the result in Section 2, we discuss why L1-regularized classification is a type of problems suitable for CD to be directly parallelized.

Given label-instance pairs $\{(y_i, \boldsymbol{x}_i)\}$, $y_i \in \{-1, +1\}$, $\boldsymbol{x}_i \in \boldsymbol{R}^n$, $i = 1, ..., l$, a binary linear classifier predicts the class label of a test instance $\boldsymbol{x}$ by the decision value $\boldsymbol{w}^T\boldsymbol{x}$. The weight vector $\boldsymbol{w} \in \boldsymbol{R}^n$ fits to the data set by solving the optimization problem:

$$\min_{\boldsymbol{w}} \; f(\boldsymbol{w}), \text{ where}$$
$$f(\boldsymbol{w}) \equiv \begin{cases} \|\boldsymbol{w}\|_1 \\ \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} \end{cases} + C \sum_{i=1}^{l} \xi(\boldsymbol{w}^T\boldsymbol{x}_i, y_i) \tag{3.2}$$

depending on the use of L1 regularization $\|\boldsymbol{w}\|_1$ or L2 regularization $\boldsymbol{w}^T\boldsymbol{w}/2$. The loss function $\xi(\boldsymbol{w}^T\boldsymbol{x}, y)$ measures the difference between the predicted value $\boldsymbol{w}^T\boldsymbol{x}$ and the true label $y$, and $C$ is the regularization parameter. Two commonly used loss functions are

$$\xi(\boldsymbol{w}^T\boldsymbol{x}, y) \equiv \begin{cases} \log(1 + \exp(-y\boldsymbol{w}^T\boldsymbol{x})) & \text{logistic loss,} \\ \max(0, 1 - y\boldsymbol{w}^T\boldsymbol{x})^2 & \text{squared hinge loss.} \end{cases} \tag{3.3}$$

To apply a CD method to solve (3.2), a coordinate $w_j$ is updated at a time. If

$$w_j \leftarrow w_j + d \tag{3.4}$$

is the change of the variable $w_j$, to get the new loss values, we need all non-zero values of feature $j$

$$\boldsymbol{w}^T\boldsymbol{x}_i \leftarrow \boldsymbol{w}^T\boldsymbol{x}_i + d(\boldsymbol{x}_i)_j,\ \forall i. \tag{3.5}$$

Examples of CD to solve (3.2) include [3, 21] for L2-regularized problems and [7, 18, 16] for L1 problems.

On the other hand, instead of (3.2), which is often referred to as the primal problem, we may solve the dual problem. If L2 regularization is used with the squared hinge loss, the dual problem of (3.2) is

$$\begin{aligned}\min_{\boldsymbol{\alpha}}\ & \frac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l} y_i y_j \boldsymbol{x}_i^T\boldsymbol{x}_j\alpha_i\alpha_j + \frac{1}{4C}\sum_{i=1}^{l}\alpha_i^2 - \sum_{i=1}^{l}\alpha_i \\ & \text{subject to}\ \ \alpha_i \geq 0\ \forall i.\end{aligned} \tag{3.6}$$

Existing CD works to solve the dual problem (3.6) include [8, 17]. Each time a coordinate $\alpha_i$ is updated, and while we do not give details, the instance $\boldsymbol{x}_i$ is used. Therefore, CD methods to solve the dual problem access/use data in an instance-wise manner. In contrast, CD methods for the primal problem (3.2) access/use data in a feature-wise manner. From features' skewed non-zero distribution shown in Section 2, we anticipate that the direct parallelization of a primal CD method to solve (3.2) can achieve a better speedup than a dual CD method for solving (3.6). We will experimentally confirm this conjecture.

Primal CD methods have been developed for both L1 and L2 regularization, but we focus on L1 problems because of the following reasons. Currently, primal CD is among the most efficient single-thread training methods for L1-regularized problems. However, it does not enjoy the same status for L2 problems because under L2 regularization, dual CD is generally considered more efficient than primal CD [8]. In contrast, for L1-regularized problems, dual-based methods (not necessarily CD-type methods) have not been very successful because the dual problem involves a more complicated $L_\infty$-ball constraint. Taking the L1-regularized problem with the squared hinge loss as an example, the dual is

$$\begin{aligned}\min_{\boldsymbol{\alpha}}\ & \frac{1}{4C}\sum\nolimits_{i=1}^{l}\alpha_i^2 - \sum\nolimits_{i=1}^{l}\alpha_i \\ \text{subject to}\ & \alpha_i \geq 0\ \forall i \\ & \|\alpha_1 y_1\boldsymbol{x}_1 + ... + \alpha_l y_l \boldsymbol{x}_l\|_\infty \leq 1.\end{aligned} \tag{3.7}$$

For (3.6), if an $\alpha_i$ is updated by fixing others, the problem is reduced to a single-variable sub-problem with a simple constraint $\alpha_i \geq 0$ and can be easily solved. For (3.7), to update an $\alpha_i$, it is unclear how to easily form and solve a sub-problem. Therefore, between primal CD for L1 and L2 problems, it is more important to study the former. Besides, in Section 3.2 we will show that the model sparsity by L1 regularization and the features' skewed distribution of non-zeros can together make parallel primal CD more effective.

Next we discuss a primal CD for L1-regularized problems and its direct parallelization.

**3.1 CDN: A Primal CD Method** For L1-regularized problems with both squared hinge and logistic losses, the comparison [18] has shown that primal CD is the best among all state-of-the-art algorithms. For the logistic loss, later [19] proposed a new method called newGLMNET. It involves a sequence of sub-problems, but each one is still solved by a CD procedure. Therefore, CD plays a vital role for L1-regularized linear classification. For simplicity, we consider the standard CD setting and leave the details of newGLMNET in the supplementary materials.

At each CD step, if the $j$-th feature is chosen, we aim to solve the following one-variable sub-problem:

$$\min\nolimits_z\ \ f(\boldsymbol{w} + z\boldsymbol{e}_j) - f(\boldsymbol{w}), \tag{3.8}$$

where $\boldsymbol{w}$ is the current solution and

$$\boldsymbol{e}_j \equiv [\underbrace{0, ..., 0}_{j-1}, 1, 0, ..., 0]^T \in \boldsymbol{R}^n. \tag{3.9}$$

Regardless of using the logistic loss or the squared hinge loss, (3.8) does not have a closed-form solution. Thus [18] develops a Newton method with line search to approximately solve (3.8), and their method is referred to as CDN (CD Newton). Specifically, we consider the second-order approximation of the loss term at $w_j$.

$$\begin{aligned} & \min_z f(\boldsymbol{w} + z\boldsymbol{e}_j) - f(\boldsymbol{w}) \\ =& \min_z |w_j + z| + L_j(z; \boldsymbol{w}) \\ \approx& \min_z |w_j + z| + L_j'(0; \boldsymbol{w})z + \frac{1}{2}L_j''(0; \boldsymbol{w})z^2,\end{aligned} \tag{3.10}$$

where

$$L_j(z; \boldsymbol{w}) \equiv C\sum\nolimits_{i=1}^{l}\xi((\boldsymbol{w} + z\boldsymbol{e}_j)^T\boldsymbol{x}_i, y_i), \tag{3.11}$$

$$L_j'(0; \boldsymbol{w}) = C\sum_{i:(\boldsymbol{x}_i)_j\neq 0}(\boldsymbol{x}_i)_j \cdot \partial_{\boldsymbol{w}^T\boldsymbol{x}}\xi(\boldsymbol{w}^T\boldsymbol{x}_i, y_i), \tag{3.12}$$

$$L_j''(0; \boldsymbol{w}) = C\sum_{i:(\boldsymbol{x}_i)_j\neq 0}(\boldsymbol{x}_i)_j^2 \cdot \partial^2_{\boldsymbol{w}^T\boldsymbol{x}}\xi(\boldsymbol{w}^T\boldsymbol{x}_i, y_i). \tag{3.13}$$

The work [18] takes a direction $d$ by solving (3.10), which has a closed-form solution (details not shown). To ensure the convergence, [18] conducts a line search

**Algorithm 1** The CDN procedure

```
Given $\boldsymbol{w}$; set $\boldsymbol{b} = [\boldsymbol{w}^T\boldsymbol{x}_1, ..., \boldsymbol{w}^T\boldsymbol{x}_l]^T$
while true do
    for $j = 1, 2, ..., n$ do
        Find $d$ by solving (3.10); let $t \leftarrow 0$
        while (3.14) fails do
            Update $\boldsymbol{b}$ by (3.15)
            $t \leftarrow t + 1$
        $w_j \leftarrow w_j + \beta^t d$
```

process to check if $d, \beta d, \beta^2 d, ...$ satisfy

$$\begin{aligned} &(3.14) \qquad f(\boldsymbol{w} + \beta^t d\boldsymbol{e}_j) - f(\boldsymbol{w}) \\ &= |w_j + \beta^t d| - |w_j| + C \sum_{i:(\boldsymbol{x}_i)_j \neq 0} \big(\xi((\boldsymbol{w} + \beta^t d\boldsymbol{e}_j)^T\boldsymbol{x}_i, y_i) \\ &\quad -\xi(\boldsymbol{w}^T\boldsymbol{x}_i, y_i)\big) \leq \sigma\beta^t \big(L'_j(0; \boldsymbol{w})d + |w_j + d| - |w_j|\big), \end{aligned}$$

where $t = 0, 1, 2, ...,$ and $\beta \in (0, 1)$ and $\sigma \in (0, 1)$ are given constants. For both logistic and squared hinge losses, the convergence of CDN was established in [18]. Clearly, in each of the following two places we need a loop to go over feature $j$'s non-zero entries.
1. The calculation of $L'_j(0; \boldsymbol{w})$ and $L''_j(0; \boldsymbol{w})$ in (3.12) and (3.13).
2. The function-value evaluation in line search; see the summation in (3.14).

At the first glance, these operations are not the bottleneck because calculating $(\boldsymbol{w} + \beta^t d\boldsymbol{e}_j)^T\boldsymbol{x}_i, \forall i$ in (3.14) is much more expensive. CDN considers a cost-saving technique by maintaining $\boldsymbol{b} \equiv [\boldsymbol{w}^T\boldsymbol{x}_1, ..., \boldsymbol{w}^T\boldsymbol{x}_l]^T$. That is, at the line search we update $\boldsymbol{b}$ by

$$(3.15) \qquad \begin{aligned} &b_i \leftarrow b_i + d(\boldsymbol{x}_i)_j, \text{ if } t = 0, \\ &b_i \leftarrow b_i - (\beta^{t-1}d - \beta^t d)(\boldsymbol{x}_i)_j, \text{ otherwise.} \end{aligned}$$

Therefore, we always have the current $\boldsymbol{w}^T\boldsymbol{x}_i, \forall i$. A summary of the CDN procedure is in Algorithm 1.[2] From (3.12)–(3.15), all we need are loops to go over $(\boldsymbol{x}_i)_j \neq 0, \forall i$. We can directly parallelize these operations though from Section 2, the effectiveness depends on the number of non-zeros in $\bar{\boldsymbol{x}}_j$. For (3.12) and (3.13), we need a reduce operation to sum up values obtained from different threads, while (3.15) can be conducted by a simple parallel loop. The implementation can be easily done by, for example, using `OpenMP` [5].

### 3.2 Shrinking Technique Helps the Parallelization

For L1-regularized problems, because of the model sparsity, a shrinking technique is often used to accelerate the training. It skips those features that are likely to have corresponding $w_j = 0$ in the final model. More details can be found in [18, 19].

An interesting question is whether shrinking influences the parallelization of the algorithm. For this question, there are three possible answers:
1. Sparse features tend to be shrunk earlier than dense features. In this case dense features (which are easy to be parallelized) are updated more times than sparse features (which are hard to be parallelized), so the speedup of using shrinking is better than not.
2. Dense features tend to be shrunk earlier than sparse features. In this case shrinking causes worse speedup.
3. Shrinking is not related to feature density, and therefore has no influence on parallelization.

Conceptually, a dense feature, usually more important, may have a non-zero weight in the final model and is less likely be shrunk. Because operations on a dense feature can be more effectively parallelized, shrinking may improve the speedup. In Section 5.2 we experimentally confirm this result.

## 4 Existing Parallel CD Methods for L1-regularized Classification

Several works have modified the CD method to solve (3.2) in parallel, e.g., [2, 15, 1, 14, 13]. They mostly change the sequential update rule to parallel updates so that multiple threads can work simultaneously. We focus on two CDN-based parallel methods.

**Shotgun [2]**: This method is an asynchronous CD approach. Each time a feature subset $B$ is randomly obtained, where $|B|$ is the number of cores. Then these cores conduct the CDN procedure to update all $w_j \in B$ in parallel. The procedure is summarized in Algorithm 2. A known issue of using asynchronous CD is that the procedure may not converge to an optimal solution. A convergence result was established in [2] by using the gradient direction with a fixed step size, but the same work states that "we can avoid divergence by imposing a step size, but our experiments showed that approach to be impractical." Therefore, for experiments they implemented Algorithm 2 that uses a Newton direction with line search. Further, cores are applied to the full feature set $\{1, 2, ..., n\}$ rather than a subset $B$ to achieve better parallelism. Unfortunately, we will show in experiments that such practical settings may lead to the divergence in some situations.

In the implementation, like CDN, Shotgun maintains $b_i = \boldsymbol{w}^T\boldsymbol{x}_i \; \forall i$ in an array and updates the array through (3.5). Because cores may change $b_i$ at the same time, an atomic operation is imposed for the update. Here we see another issue of asynchronous CD. A direc-

[2]In [18], a technique is developed to future reduce the line-search cost, though details are not shown here.

**Algorithm 2** Shotgun CDN procedure

```
Given $\boldsymbol{w}$; set $\boldsymbol{b} = [\boldsymbol{w}^T\boldsymbol{x}_1, ..., \boldsymbol{w}^T\boldsymbol{x}_l]^T$
while true do
    Randomly get $|B|$ indices, where $|B| = \#$cores
    for $j$ in $B$ in parallel do
        Find $d$ by solving (3.10); let $t \leftarrow 0$
        while (3.14) fails do
            atomic: update $\boldsymbol{b}$ by (3.15)
            $t \leftarrow t + 1$
        $w_j \leftarrow w_j + \beta^t d$
```

tion $d$ is found based on $\boldsymbol{w}$, but when doing line search, $\boldsymbol{w}$ may have been changed to $\boldsymbol{w}'$. Hence, $d\boldsymbol{e}_j$ may not be a descent direction at $\boldsymbol{w}'$ and line search may fail.

**Bundle** CDN: [1] proposed a method to address the divergence issue of Shotgun, which is caused by the dependency breaking of coordinate updates. The proposed method basically resolves the issue by introducing an extra layer of coordination to parallel updates. Following the block CD framework in [20], at each iteration, a feature subset $B$ is considered and the direction $\boldsymbol{d}$ is obtained by solving

$$\begin{aligned} &\min_{\boldsymbol{d}_B} \ \|\boldsymbol{w}_B + \boldsymbol{d}_B\|_1 + \nabla_B L(\boldsymbol{w})^T \boldsymbol{d}_B + \frac{1}{2}\boldsymbol{d}_B^T H \boldsymbol{d}_B \\ &\text{subject to} \ \ d_j = 0, \ j \notin B, \end{aligned} \tag{4.16}$$

where $L(\boldsymbol{w}) \equiv C\sum_{i=1}^{l} \xi(\boldsymbol{w}^T\boldsymbol{x}_i, y_i)$ and the matrix $H$ can be any approximation of $\nabla^2_{BB}L(\boldsymbol{w})$. By considering

$$H = \text{diag}(\nabla^2_{BB}L(\boldsymbol{w})) \tag{4.17}$$

to include all diagonal entries of $\nabla^2_{BB}L(\boldsymbol{w})$, (4.16) becomes $|B|$ independent one-variable sub-problems, each of which is the same as (3.10) considered in CDN. Thus these sub-problems can be solved in parallel to get a direction $\boldsymbol{d}_B$. Note that $|B|$ can be any value regardless of the number of cores. Then a line search process finding the largest $\beta^t, \ t = 0, 1, ...$ such that

$$\begin{aligned} &f(\boldsymbol{w} + \beta^t \begin{bmatrix} \boldsymbol{d}_B \\ \boldsymbol{0} \end{bmatrix}) - f(\boldsymbol{w}) \\ \leq &\sigma\beta^t \big(\nabla_B L(\boldsymbol{w})^T \boldsymbol{d}_B + \|\boldsymbol{w}_B + \boldsymbol{d}_B\|_1 - \|\boldsymbol{w}_B\|_1\big) \end{aligned} \tag{4.18}$$

ensures the convergence by the theoretical result in [20].

The bundle size $|B|$ can be as large as $n$, but the diagonal approximation in (4.17) may result in a poor direction. In experiments we follow [1]'s setting to use a rather large bundle size (10,000 or more), but find that the convergence is slow. This result is not surprising because CD is designed to greedily update $\boldsymbol{w}$ in a sequential manner, but now we obtain $|B|$ CD steps based on the same $\boldsymbol{w}$. To improve the consequence we may consider a smaller bundle size (e.g., a value slightly larger than the number of cores). However, some implementation issues occur. To conduct line search, the following vector indicating the change of $\boldsymbol{w}^T\boldsymbol{x}_i, \ \forall i$ must be obtained

$$\sum\nolimits_{j \in B} d_j \bar{\boldsymbol{x}}_j. \tag{4.19}$$

Because cores may simultaneously update the same entry in (4.19), in [1], a compare-and-swap atomic array update is implemented. These atomic updates will cause a significant waiting time in some situations (e.g., there are some dense features in the bundle). Further, the implementation requires that (4.19) is stored as a dense array. To have $(\boldsymbol{w} + \beta^t d)^T\boldsymbol{x}_i, \ \forall i$ for the new function value, we need $O(l)$ for summing up two vectors. This $O(l)$ cost is very expensive when $|B|$ is small. For sparse sets, the cost of obtaining each $d_j, \ j \in B$ is only $O(\#\text{non-zeros in feature } j)$, which is much smaller than $O(l)$. In summary, our discussion fully demonstrates the difficulty in making bundle CDN an efficient approach.

## 5 Experiments

In this section, we experimentally demonstrate that a naive parallelization of CDN is very effective for L1-regularized classification. We further investigate the role of shrinking and the ineffectiveness of parallelizing dual CD for L2-regularized classification.

We consider 11 public sets,[3] where statistics are in Table 2. For `webspam`, the tri-gram version is used. For `splice_site.t.10%`, it is a 10% random subset because the original set is too large for our machine. Experiments were conducted on an Amazon EC2 r4.4xlarge machine with an 8-core Intel Xeon E5-2686 v4 Processor.[4] For parameters, we choose $C = 1$ and apply LIBLINEAR's default stopping condition [6]. We use 500 as the threshold to decide if a feature is dense enough and operations should be parallelized. In supplementary materials, we show this threshold can be easily selected because we simply need to use neither a too small nor a too large value.

### 5.1 Comparison Between State-of-the-art Methods

We extensively compare the following approaches implemented in C++ and OpenMP.

• **Naive** CDN: We parallelize loops in the CDN implementation of LIBLINEAR. For LR we use an earlier version 1.7 because in the current LIBLINEAR the solver

[3]`https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html`. Note that `yahoo-korea` is not available.

[4]We disable hyperthreading in the machine.

| Data set | #instances | #features |
|---|---|---|
| avazu-app | 14,596,137 | 1,000,000 |
| criteo | 45,840,617 | 1,000,000 |
| epsilon | 400,000 | 2,000 |
| HIGGS | 11,000,000 | 28 |
| kdd2010-a | 8,407,752 | 20,216,830 |
| kdd2012 | 149,639,105 | 54,686,452 |
| rcv1_test | 677,399 | 47,236 |
| splice_site.t.10% | 462,033 | 11,725,480 |
| url_combined | 2,396,130 | 3,231,961 |
| webspam | 350,000 | 16,609,143 |
| yahoo-korea | 460,554 | 3,052,939 |

Table 2: Data statistics.

has been changed to newGLMNET.[5]

• **Shotgun** CDN [2]: Although the code is publicly available,[6] to make a fair comparison by using, for example, the same data structure, we implement this method based on LIBLINEAR.

• **Bundle** CDN [1]: We use the authors' code[7] because it is directly extended from LIBLINEAR-1.7. Their code considers the model with a bias term (i.e., $\boldsymbol{w}^T\boldsymbol{x}+b$ is the decision value), so we modify it to use the same decision value $\boldsymbol{w}^T\boldsymbol{x}$ as others. We use 10,000 as the bundle size, but for kdd2010-a, the same size (29,500) as in [1] is considered.

For all the above methods, the shrinking technique discussed in Section 3.2 is applied. We give experimental analysis of this technique in Section 5.2.

Speedup is commonly used to measure parallel algorithms. Following [2], we define speedup as

$$\frac{\text{Time to reach } 1.005 \times f^* \text{ with 1 thread}}{\text{Time to reach } 1.005 \times f^* \text{ with multiple threads}}, \tag{5.20}$$

where $f^*$ is the objective value obtained by LIBLINEAR with the default stopping condition. All three methods reduce to the standard CDN if one thread is used. Thus, the comparison is fair because the same numerator is used in (5.20).

From results in Table 3, we observe that the speedup of Naive CDN is generally better than Shotgun, and is significantly better than Bundle CDN. Further the performance of Shotgun is not stable. While it is the best for highly sparse sets such as kdd2010-a and kdd2012, it fails to converge on the dense data epsilon and HIGGS when 8 threads are used. This result is consistent with earlier findings in asynchronous CD experiments [4] (for dual L2-regularized problems), where they found that for dense data, threads more easily collide with each other. For Bundle CDN, in many data sets it is even slower than the baseline. Besides the analysis in Section 4, we study Bundle CDN in more details in supplementary materials.

Different from Shotgun and Bundle CDN, Naive CDN changes neither the algorithm nor the convergence of the vanilla CDN. Thus we have a concrete example where the direct parallelization of a CD method is faster and more robust than sophisticated modifications.

Besides, we observe that the speedup of Naive CDN on LR is better than that on SVM. The reason is that for LR, exp / log operations are involved in (3.13)-(3.15). Because each exp / log operation is more expensive than regular arithmetic operations, loops suitable to be parallelized occupy a larger portion of the total cost. Further, when a loop involves expensive operations at each element, the speedup is often better. That is, the overhead in (2.1) becomes relatively less important.

**5.2 Effectiveness of Shrinking under L1 Regularization** The speedup with and without shrinking is shown in Table 4. In general the shrinking technique helps to significantly improve the speedup. This result confirms the conjecture in Section 3.2: denser features tend to be retained and are updated more times. We give further details in Table 5 by splitting all features to 10 bins according to the density of features and letting each bin have about the same number of non-zeros. Then the first bin corresponds to the densest features, while the last corresponds to the sparsest ones. We then check the average number of updates of features in each bin. For data sets that benefit more from shrinking, dense features are updated more times than sparse features. Apparently, many sparse features are removed in the middle of training procedure. For url_combined although only features in the last bin are updated fewer times, the speedup after applying shrinking is still improved. This set has a very long tail on the distribution of features' non-zeros – the last bin contains thousand times more features than all other bins combined.

**5.3 Speedup of Dual CD for L2-regularized Problems** We conduct experiments to demonstrate the ineffectiveness of naive parallelization on methods that access data instance-wisely. We consider the dual CD implementation [8] in LIBLINEAR for L2-regularized squared hinge-loss SVM.

Because of space limit, we present in Table 6 only results of some sparse and dense sets. For sparse sets, none of their instances has more non-zeros than the threshold for choosing between multiple- and single-thread tasks, so we always use multiple threads (i.e.,

[5]Results of parallelizing newGLMNET are in supplementary materials.

[6]https://github.com/akyrola/shotgun

[7]https://github.com/bianan/ParallelCDN

| Data set \ #threads | LR | | | | | | | | | SVM | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Naive | | | Bundle | | | Shotgun | | | Naive | | | Bundle | | | Shotgun | | |
| | 2 | 4 | 8 | 2 | 4 | 8 | 2 | 4 | 8 | 2 | 4 | 8 | 2 | 4 | 8 | 2 | 4 | 8 |
| avazu-app | **1.9** | **3.4** | **5.6** | 0.4 | 0.7 | 1.0 | 1.4 | 2.7 | 3.4 | **1.5** | **2.4** | **4.3** | 0.2 | 0.3 | 0.4 | 1.1 | 2.1 | 3.4 |
| criteo | **1.8** | **3.3** | **5.5** | 0.7 | 1.2 | 1.9 | 1.5 | 2.9 | 4.8 | **1.6** | **2.6** | **4.2** | 0.5 | 0.8 | 1.0 | 1.3 | 1.8 | 3.3 |
| epsilon | **2.0** | **4.0** | **7.9** | x | x | x | 1.3 | 2.1 | x | **2.0** | **3.9** | **7.8** | x | x | x | 1.0 | 2.1 | x |
| HIGGS | **2.0** | **3.9** | **7.5** | 0.7 | 0.8 | 0.9 | 1.0 | 1.3 | x | **1.8** | **2.8** | **4.7** | x | x | x | 0.8 | 2.2 | x |
| kdd2010-a | **1.7** | 2.4 | 3.1 | 0.8 | 1.4 | 2.4 | 1.5 | **2.7** | **4.8** | **1.5** | 1.9 | 2.2 | 0.7 | 1.3 | 1.9 | 1.2 | **2.2** | **3.7** |
| kdd2012 | 1.9 | 2.8 | 3.9 | 0.2 | 0.4 | 0.6 | **2.1** | **4.7** | **7.0** | **1.5** | 2.4 | 3.0 | x | x | x | 1.5 | **3.2** | **5.0** |
| rcv1_test | **1.9** | **3.4** | **5.9** | x | x | x | 1.3 | 2.5 | 4.5 | **1.6** | **2.4** | **3.2** | x | 0.1 | 0.2 | 0.7 | 1.3 | 2.3 |
| splice_site.t.10% | **1.9** | **3.6** | **6.2** | x | x | x | 1.6 | 2.7 | 4.3 | **1.8** | **3.1** | **4.4** | x | x | x | 1.0 | 1.3 | 2.2 |
| url_combined | **2.0** | **3.5** | **6.2** | 0.5 | 0.9 | 1.3 | 1.0 | 1.7 | 1.7 | **1.7** | **2.6** | **3.7** | x | 0.1 | 0.2 | 0.5 | 1.0 | 0.8 |
| webspam | **1.8** | **3.2** | **4.8** | 0.1 | 0.3 | 0.5 | 1.4 | 2.5 | 4.1 | **1.5** | **2.2** | **3.0** | x | x | x | 0.7 | 1.2 | 1.9 |
| yahoo-korea | **1.9** | **3.5** | **5.9** | 0.2 | 0.3 | 0.5 | 1.3 | 2.4 | 4.4 | **1.6** | **2.2** | 2.8 | x | x | 0.1 | 1.0 | 1.7 | **3.1** |

Table 3: Speedup by using 2, 4, and 8 cores. Left: L1-regularized LR. Right: L1-regularized SVM (with squared hinge loss). The symbol "x" means the approach fails to achieve the desired function value or the speedup is smaller than 0.1. Under the same number of cores, the best approach is bold-faced.

| Data set \ #threads | LR | | | | | | SVM | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Shrinking | | | No shrinking | | | Shrinking | | | No shrinking | | |
| | 2 | 4 | 8 | 2 | 4 | 8 | 2 | 4 | 8 | 2 | 4 | 8 |
| avazu-app | 1.9 | 3.4 | 5.6 | 1.9 | 3.4 | 5.3 | 1.5 | 2.4 | 4.3 | 1.8 | 2.8 | 3.7 |
| criteo | 1.8 | 3.3 | 5.5 | 1.9 | 3.1 | 5.0 | 1.6 | 2.6 | 4.2 | 2.0 | 3.3 | 4.2 |
| kdd2010-a | 1.7 | 2.4 | 3.1 | 1.3 | 1.6 | 1.8 | 1.5 | 1.9 | 2.2 | 1.2 | 1.4 | 1.4 |
| kdd2012 | 1.9 | 2.8 | 3.9 | 1.4 | 1.6 | 2.3 | 1.5 | 2.4 | 3.0 | 1.2 | 1.4 | 1.7 |
| rcv1_test | 1.9 | 3.4 | 5.9 | 1.8 | 3.3 | 5.5 | 1.6 | 2.4 | 3.2 | 1.6 | 2.3 | 3.0 |
| splice_site.t.10% | 1.9 | 3.6 | 6.2 | 1.4 | 2.0 | 2.4 | 1.8 | 3.1 | 4.4 | 1.1 | 1.2 | 1.1 |
| url_combined | 2.0 | 3.5 | 6.2 | 1.8 | 3.3 | 5.1 | 1.7 | 2.6 | 3.7 | 1.5 | 2.1 | 2.5 |
| webspam | 1.8 | 3.2 | 4.8 | 1.7 | 2.6 | 3.7 | 1.5 | 2.2 | 3.0 | 1.3 | 1.7 | 2.0 |
| yahoo-korea | 1.9 | 3.5 | 5.9 | 1.7 | 2.8 | 4.0 | 1.6 | 2.2 | 2.8 | 1.3 | 1.7 | 1.9 |

Table 4: Speedup of naive parallelization of CDN with and without shrinking.

threshold is decreased to zero). The resulting speedup is very poor for sparse sets because few operations are conducted per instance and the overhead in (2.1) accounts for a significant portion of the running time. This experiment confirms the conventional thinking that the naive parallelization of CD is in general not useful. Interestingly and surprisingly, our finding of the extremely skewed distribution of features' non-zeros in real-world sparse sets comes to the rescue in making the naive parallelization of primal CD highly effective.

## 6 Conclusions

As the developers of the popular package LIBLINEAR, we have long wanted to provide multi-core extensions for users. After some efforts we have had successful implementations for L2-regularized classification [10, 4]. However, for L1-regularized problems we struggled to develop an effective solution. Existing approaches either have convergence issues (e.g., Shotgun) or do not give good speedup. Surprisingly, a solution turns out to be the direct parallelization of loops in CD. We show that this strategy is effective because first, many sparse data sets have skewed feature-wise non-zero distributions, and second, the shrinking technique, if applied to L1-regularized problems, effectively helps the parallelization. We retain the same convergence property and achieve excellent speedup without modifying the CD algorithm at all. For data sets that do not have skewed distributions, we are developing some novel techniques to achieve better speedup than the current naive parallelization.

## Acknowledgments

This work was supported by 2017 Criteo Faculty Research Award. We also thank Criteo research team members for giving us feedback.

## References

[1] Y. Bian, X. Li, M. Cao, and Y. Liu. Bundle CDN: a highly parallelized approach for large-scale l1-regularized logistic regression. In *ECML/PKDD*, 2013.

[2] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel coordinate descent for l1-regularized loss minimization. In *ICML*, 2011.

[3] K.-W. Chang, C.-J. Hsieh, and C.-J. Lin. Coordinate

| Data set | | First bin: densest features. Last bin: sparsest features. | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| `avazu-app` | #features | 2.0e+00 | 3.0e+00 | 7.0e+00 | 1.4e+01 | 2.5e+01 | 3.8e+01 | 7.3e+01 | 1.9e+02 | 6.0e+02 | 2.3e+04 |
| | #updates | 90 | 90 | 90 | 90 | 90 | 90 | 90 | 90 | 89 | 54 |
| `criteo` | #features | 6.0e+00 | 9.0e+00 | 1.5e+01 | 2.3e+01 | 4.5e+01 | 1.0e+02 | 2.6e+02 | 9.8e+02 | 6.3e+03 | 6.6e+05 |
| | #updates | 64 | 64 | 64 | 64 | 64 | 64 | 64 | 63 | 63 | 28 |
| `kdd2010-a` | #features | 9.7e+01 | 2.1e+02 | 4.3e+02 | 1.0e+03 | 3.1e+03 | 1.5e+04 | 8.6e+04 | 3.5e+05 | 1.5e+06 | 1.7e+07 |
| | #updates | 347 | 346 | 347 | 340 | 323 | 243 | 166 | 140 | 99 | 46 |
| `kdd2012` | #features | 2.0e+00 | 3.0e+00 | 1.0e+01 | 3.2e+02 | 2.3e+03 | 1.1e+04 | 4.4e+04 | 2.4e+05 | 2.8e+06 | 5.2e+07 |
| | #updates | 1,000 | 1,000 | 1,000 | 971 | 967 | 886 | 759 | 564 | 332 | 52 |
| `url_combined` | #features | 1.2e+01 | 1.2e+01 | 1.2e+01 | 1.2e+01 | 1.2e+01 | 1.5e+01 | 2.3e+01 | 1.0e+02 | 1.1e+03 | 3.2e+06 |
| | #updates | 24 | 24 | 24 | 24 | 24 | 24 | 24 | 23 | 22 | 5 |
| `webspam` | #features | 4.6e+02 | 5.8e+02 | 7.4e+02 | 9.5e+02 | 1.2e+03 | 1.7e+03 | 2.9e+03 | 5.5e+03 | 1.4e+04 | 6.5e+05 |
| | #updates | 31 | 24 | 17 | 12 | 8 | 9 | 8 | 6 | 5 | 5 |
| `rcv1_test` | #features | 3.1e+01 | 5.3e+01 | 8.0e+01 | 1.2e+02 | 1.8e+02 | 2.6e+02 | 4.3e+02 | 8.2e+02 | 2.2e+03 | 3.9e+04 |
| | #updates | 13 | 13 | 12 | 12 | 12 | 12 | 12 | 12 | 11 | 7 |
| `splice_site.t.10%` | #features | 1.0e+03 | 5.6e+03 | 2.9e+04 | 1.4e+05 | 5.6e+05 | 1.1e+06 | 1.3e+06 | 1.4e+06 | 1.5e+06 | 5.7e+06 |
| | #updates | 312 | 61 | 4 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| `yahoo-korea` | #features | 8.5e+01 | 1.9e+02 | 3.4e+02 | 5.9e+02 | 1.0e+03 | 1.8e+03 | 3.7e+03 | 9.0e+03 | 3.6e+04 | 3.0e+06 |
| | #updates | 24 | 24 | 23 | 23 | 23 | 22 | 20 | 18 | 12 | 5 |

Table 5: The number of features in each bin and the average number of updates of features in the same bin. Each bin contains roughly 10% of total non-zeros.

| Data set \ #threads | | 2 | 4 | 8 |
|---|---|---|---|---|
| sparse sets | `avazu-app` | 0.4 | 0.3 | 0.2 |
| | `criteo` | 0.5 | 0.3 | 0.2 |
| | `url_combined` | 0.6 | 0.4 | 0.3 |
| dense sets | `epsilon` | 1.3 | 1.3 | 1.1 |
| | `splice_site.t.10%` | 1.8 | 2.8 | 4.1 |
| | `webspam` | 1.6 | 2.4 | 2.9 |

Table 6: Speedup of the direct parallelization of CD on dual L2-regularized SVM. For problems in the upper part, none of the instances has more non-zeros than the threshold to choose between multiple- and single-thread tasks, so we always use multiple threads. Because of space limit, not all sparse sets are presented.

descent method for large-scale L2-loss linear SVM. *JMLR*, 9:1369–1398, 2008.

[4] W.-L. Chiang, M.-C. Lee, and C.-J. Lin. Parallel dual coordinate descent method for large-scale linear classification in multi-core environments. In *KDD*, 2016.

[5] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5:46–55, 1998.

[6] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: a library for large linear classification. *JMLR*, 9:1871–1874, 2008.

[7] A. Genkin, D. D. Lewis, and D. Madigan. Large-scale Bayesian logistic regression for text categorization. *Technometrics*, 49(3):291–304, 2007.

[8] C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan. A dual coordinate descent method for large-scale linear SVM. In *ICML*, 2008.

[9] F.-L. Huang, C.-J. Hsieh, K.-W. Chang, and C.-J. Lin. Iterative scaling and coordinate descent methods for maximum entropy. *JMLR*, 2010.

[10] M.-C. Lee, W.-L. Chiang, and C.-J. Lin. Fast matrix-vector multiplications for large-scale logistic regression on shared-memory systems. In *ICDM*, 2015.

[11] C.-J. Lin, R. C. Weng, and S. S. Keerthi. Trust region Newton method for large-scale logistic regression. *JMLR*, 9:627–650, 2008.

[12] D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Math. Program.*, 45(1):503–528, 1989.

[13] J. Liu and S. J. Wright. Asynchronous stochastic coordinate descent: Parallelism and convergence properties. *SIAM J. Optim.*, 25, 2015.

[14] J. Liu, S. J. Wright, C. Ré, V. Bittorf, and S. Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. *JMLR*, 16, 2015.

[15] P. Richtárik and M. Takáč. Parallel coordinate descent methods for big data optimization. *Math. Program.*, 156:433–484, 2016.

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

[17] H.-F. Yu, C.-J. Hsieh, K.-W. Chang, and C.-J. Lin. Large linear classification when data cannot fit in memory. *ACM TKDD*, 5, 2012.

[18] G.-X. Yuan, K.-W. Chang, C.-J. Hsieh, and C.-J. Lin. A comparison of optimization methods and software for large-scale l1-regularized linear classification. *JMLR*, 11:3183–3234, 2010.

[19] G.-X. Yuan, C.-H. Ho, and C.-J. Lin. An improved GLMNET for l1-regularized logistic regression. *JMLR*, 13, 2012.

[20] S. Yun and K.-C. Toh. A coordinate gradient descent method for l1-regularized convex minimization. *Computational Opt. and App.*, 48:273–307, 2011.

[21] T. Zhang and F. J. Oles. Text categorization based on regularized linear classification methods. *IR*, 4, 2001.