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

Yong Zhuang*
Carnegie Mellon University
yong.zhuang22@gmail.com

Yuchin Juan†
Criteo Research
yc.juan@criteo.com

Guo-Xun Yuan
Facebook, Inc.
gxyzuan@gmail.com

Chih-Jen Lin
National Taiwan University
cjlin@csie.ntu.edu.tw

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. We point out that this common understanding may not be true if the algorithm sequentially accesses the data in a feature-wise manner. For almost all real-world sparse sets we have examined, some features are much denser than others. Thus 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 coordinate descent (CD) methods, which are effective single-thread technique for L1-regularized classification. 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 a highly competitive and robust multi-core implementation for L1-regularized classification.

CCS CONCEPTS

• **Computing methodologies** → *Shared memory algorithms*;

KEYWORDS

Parallelization; Coordinate Descent Methods; Multi-core; L1-regularized Classification

*Most of the work was done during the internship in Criteo Research.
†This author contributes equally with Yong Zhuang.

CIKM '18, October 22–26, 2018, Torino, Italy

ACM ISBN 978-1-4503-6014-2/18/10…$15.00
https://doi.org/10.1145/3269206.3271687

ACM Reference Format:
Yong Zhuang, Yuchin Juan, Guo-Xun Yuan, and Chih-Jen Lin. 2018. Naive Parallelization of Coordinate Descent Methods and an Application on Multi-core L1-regularized Classification. In *The 27th ACM International Conference on Information and Knowledge Management (CIKM '18), October 22–26, 2018, Torino, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3269206.3271687

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, parallelizing few operations is not useful. For example, at each iteration of batch methods we often need to 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 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. Note that CD-type methods are now considered the best single-thread technique for L1-regularized classification. Our work effectively extends them to multi-core scenarios. 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. Supplementary materials are at http://www.csie.ntu.edu.tw/~cjlin/papers/l1_parallel_supp_cikm.pdf.

2 NAIVE PARALLELIZATION AND DISTRIBUTION OF NON-ZEROS

The running time of a multi-threading task is roughly

$$\frac{\#\text{operations} \times \text{time per operation}}{\#\text{threads}} + \text{overhead}. \quad (1)$$

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 $\geq$ 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
- nnz_a: number of non-zero entries in $\boldsymbol{x}_a$
- 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 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-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

[1] 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	nnz_a	nnz_b	$\overline{\text{nnz}}_a$	$\overline{\text{nnz}}_b$	a/n	b/n	nnz_a	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.t	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, nnz_a, nnz_b, $\overline{\text{nnz}}_a$, $\overline{\text{nnz}}_b$ in selected data sets under instance-wise and feature-wise perspectives. For splice.t, a 10% subset is considered.

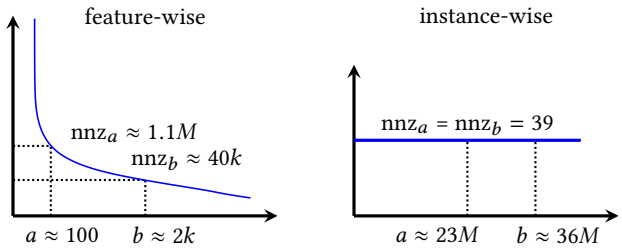


Figure 1: The distribution of non-zero elements in criteo. x-axis of left: $\{1, \ldots, n\}$, right: $\{1, \ldots, l\}$; y-axis of left: log scaled, right: linear scaled.

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. However, from an instance-wise perspective, for documents collected from the same or similar sources their numbers of words may not vary significantly.

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. However, 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.t, it seems that even with feature-wise data accesses, direct parallelization may not be effective because nnz_a and 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.

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 $\{(y_i, \boldsymbol{x}_i)\}$, $y_i \in \{-1, +1\}$, $\boldsymbol{x}_i \in \boldsymbol{R}^n$, $i = 1, \ldots, l$, a binary linear classifier obtains a model vector $\boldsymbol{w} \in \boldsymbol{R}^n$ by solving a convex optimization problem:

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

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

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

To apply a CD method to solve (2), a coordinate w_j is updated at a time. If

$$w_j \leftarrow w_j + d \tag{3}$$

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

$$\boldsymbol{w}^T\boldsymbol{x}_i \leftarrow \boldsymbol{w}^T\boldsymbol{x}_i + d(\boldsymbol{x}_i)_j, \ \forall i. \tag{4}$$

Examples of CD to solve (2) include [3, 22] for L2-regularized problems and [7, 16, 18] for L1 problems.

On the other hand, instead of (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 (2) is

$$\min_{\boldsymbol{\alpha}} \ \frac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l} y_i y_j \boldsymbol{x}_i^T\boldsymbol{x}_j\alpha_i\alpha_j + \frac{1}{4C}\sum_{i=1}^{l}\alpha_i^2 - \sum_{i=1}^{l}\alpha_i$$
$$\text{subject to } \ \alpha_i \geq 0 \ \forall i. \tag{5}$$

Existing CD works to solve the dual problem (5) include [8, 17]. Each time a coordinate α_i is updated, and while we do not give details, the corresponding 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 (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 (2) can achieve a better speedup than a dual CD method for solving (5). We will experimentally confirm this conjecture. In the rest of this section we discuss primal CD in detail.

Primal CD methods have been developed for both L1 and L2 regularization. We focus on L1 problems because primal CD is

among the most efficient single-thread training methods for L1-regularized classification. In contrast, primal CD does not enjoy the same status for L2 problems since under L2 regularization, dual CD is generally considered more efficient than primal CD [8]. Note that 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_∞-ball constraint. Take the L1-regularized problem with the squared hinge loss as an example. The dual is

$$\begin{aligned} &\min_{\boldsymbol{\alpha}} \ \frac{1}{4C}\sum_{i=1}^{l}\alpha_i^2 - \sum_{i=1}^{l}\alpha_i \\ &\text{subject to } \ \alpha_i \geq 0 \ \forall i \\ &\qquad \|\alpha_1 y_1 \boldsymbol{x}_1 + \ldots + \alpha_l y_l \boldsymbol{x}_l\|_\infty \leq 1. \end{aligned} \tag{6}$$

For (5), if an α_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 (6), to update an α_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 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 state-of-the-art algorithms. For the logistic loss, later [19] proposed a new and more efficient 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.

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

$$\min_z \ f(\boldsymbol{w} + z\boldsymbol{e}_j) - f(\boldsymbol{w}), \tag{7}$$

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

$$\boldsymbol{e}_j \equiv [\underbrace{0, \ldots, 0}_{j-1}, 1, 0, \ldots, 0]^T \in \boldsymbol{R}^n. \tag{8}$$

Regardless of using the logistic loss or the squared hinge loss, (7) does not have a closed-form solution. Thus [18] develops a Newton method with line search to approximately solve (7), and their method is referred to as CDN (CD Newton). Specifically, we consider the second-order approximation of the loss term at w_j.

$$\begin{aligned} &f(\boldsymbol{w} + z\boldsymbol{e}_j) - f(\boldsymbol{w}) = |w_j + z| + L_j(z; \boldsymbol{w}) \\ &\approx |w_j + z| + L'_j(0; \boldsymbol{w})z + \frac{1}{2}L''_j(0; \boldsymbol{w})z^2, \end{aligned} \tag{9}$$

where

$$L_j(z; \boldsymbol{w}) \equiv C\sum_{i=1}^{l}\xi((\boldsymbol{w} + z\boldsymbol{e}_j)^T\boldsymbol{x}_i, y_i), \tag{10}$$

$$L'_j(0; \boldsymbol{w}) = C\sum_{i:(\boldsymbol{x}_i)_j \neq 0}(\boldsymbol{x}_i)_j \cdot \partial_{\boldsymbol{w}^T\boldsymbol{x}}\xi(\boldsymbol{w}^T\boldsymbol{x}_i, y_i), \tag{11}$$

$$L''_j(0; \boldsymbol{w}) = C\sum_{i:(\boldsymbol{x}_i)_j \neq 0}(\boldsymbol{x}_i)_j^2 \cdot \partial^2_{\boldsymbol{w}^T\boldsymbol{x}}\xi(\boldsymbol{w}^T\boldsymbol{x}_i, y_i). \tag{12}$$

Algorithm 1 The CDN procedure for L1-regularized classification

Given $\boldsymbol{w}$; set $\boldsymbol{b} = [\boldsymbol{w}^T\boldsymbol{x}_1, \ldots, \boldsymbol{w}^T\boldsymbol{x}_l]^T$
while `true` **do**
  for $j = 1, 2, \ldots, n$ **do**
    Find d by solving (9); let $t \leftarrow 0$
    while (13) fails **do**
      Update $\boldsymbol{b}$ by (14)
      $t \leftarrow t + 1$
    $w_j \leftarrow w_j + \beta^t d$

The work [18] takes a direction d by solving (9), which has a closed-form solution (details not shown). To ensure the convergence, [18] conducts a line search process to check if $d, \beta d, \beta^2 d, \ldots$ satisfy

$$\begin{aligned} &f(\boldsymbol{w} + \beta^t d\boldsymbol{e}_j) - f(\boldsymbol{w}) \\ &= |w_j + \beta^t d| - |w_j| + C\sum_{i:(\boldsymbol{x}_i)_j \neq 0}\left(\xi((\boldsymbol{w} + \beta^t d\boldsymbol{e}_j)^T\boldsymbol{x}_i, y_i)\right. \\ &\left.-\xi(\boldsymbol{w}^T\boldsymbol{x}_i, y_i)\right) \leq \sigma\beta^t\left(L'_j(0; \boldsymbol{w})d + |w_j + d| - |w_j|\right), \end{aligned} \tag{13}$$

where $t = 0, 1, 2, \ldots$, and $\beta \in (0, 1)$ and $\sigma \in (0, 1)$ are given constants. 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 (11) and (12).
(2) The function-value evaluation in line search; see the summation in (13).

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 (13) is much more expensive. CDN considers a cost-saving technique by maintaining $\boldsymbol{b} \equiv [\boldsymbol{w}^T\boldsymbol{x}_1, \ldots, \boldsymbol{w}^T\boldsymbol{x}_l]^T$. That is, at the line search we update $\boldsymbol{b}$ by

$$\begin{aligned} &b_i \leftarrow b_i + d(\boldsymbol{x}_i)_j, \text{ if } t = 0, \\ &b_i \leftarrow b_i - (\beta^{t-1}d - \beta^t d)(\boldsymbol{x}_i)_j, \text{ otherwise.} \end{aligned} \tag{14}$$

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 (11)–(14), 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 (11) and (12), we need a reduce operation to sum up values obtained from different threads, while (14) 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. 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.

[2] In [18], a technique is developed to further 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 (9); let $t \leftarrow 0$
    while (13) fails **do**
      atomic: update $\boldsymbol{b}$ by (14)
      $t \leftarrow t + 1$
    $w_j \leftarrow w_j + \beta^t d$

3.3 Multi-core Implementation for newGLMNET

For problem (2) with the logistic loss (i.e., L1-regularized logistic regression), [19] pointed out an issue of CDN on the relatively high portion of expensive log(·)/exp(·) operations in the entire procedure. Thus they propose newGLMNET, which is now considered the state-of-the-art. Although newGLMNET is a Newton method, at each iteration it finds a Newton direction by applying a CD procedure to solve the subproblem

$$\min_{\boldsymbol{d}} \quad \|\boldsymbol{w} + \boldsymbol{d}\|_1 + \nabla L(\boldsymbol{w})^T\boldsymbol{d} + \frac{1}{2}\boldsymbol{d}^T H\boldsymbol{d}. \tag{15}$$

The subproblem is in a form of L1-regularized least squares. To perform the CD subroutine, the training data is accessed/used in a feature-wise manner, which is the same as how data is used in CDN for problem (2). Thus, the applicability of the naive parallelization to the CD subroutine should hold here and the speedup is predicted to be at a comparable level. In Section 5.4, we demonstrate the effectiveness of naive parallelization in newGLMNET.

4 EXISTING PARALLEL CD METHODS FOR L1-REGULARIZED CLASSIFICATION

Several works have modified the CD method to solve (2) in parallel, e.g., [1, 2, 13–15]. They mostly change the sequential update rule to parallel updates so that multiple threads can work simultaneously. We focus on two CDN-based 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 diverge. To have the convergence, a conservative update (e.g., gradient direction with fixed size rather than Newton direction with line search) is often needed, but such a setting is impractical [2]. Some approaches consider semi-asynchronous settings to fix the convergence issue, but they are problem dependent. For example, [21] considers CD for the dual problem and assumes that each one-variable sub-problem can be exactly solved. It is not obvious how their method and convergence analysis can be applied to our primal problem, for which the single-variable sub-problem has no closed-form solution.

Bundle CDN: [1] proposed a method to address the divergence issue of Shotgun. 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

$$\min_{\boldsymbol{d}_B} \ \|\boldsymbol{w}_B + \boldsymbol{d}_B\|_1 + \nabla_B L(\boldsymbol{w})^T\boldsymbol{d}_B + \frac{1}{2}\boldsymbol{d}_B^T H\boldsymbol{d}_B \quad \text{subject to } \ d_j = 0, \ j \notin B, \tag{16}$$

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

$$H = \text{diag}(\nabla^2_{BB}L(\boldsymbol{w})) \tag{17}$$

to include all diagonal entries of $\nabla^2_{BB}L(\boldsymbol{w})$, (16) becomes $|B|$ independent one-variable sub-problems, each of which is the same as (9) 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 is needed to ensure the convergence. Unfortunately, it is not easy to select a suitable bundle size $|B|$, which is discussed in Section A.1.

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.t	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.

5 EXPERIMENTS

In this section, we first demonstrate that a naive parallelization of CDN is very effective for L1-regularized classification. Then, we investigate the role of shrinking technique and the impact of the threshold. At last, we conduct experiments to demonstrate the speedup of parallelizing newGLMNET for L1-regularized classification and dual CD for L2-regularized classification.

The detailed statistics of the 11 public sets are in Table 2. For webspam, the tri-gram version is used. For splice.t, it is a 10% random subset because the original set is too large for our machine. Experiments were conducted on an Amazon EC2 r4.8xlarge machine with an 16-core Intel Xeon E5-2686 v4 Processor.[3] 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 Section 5.3, we show this threshold can be easily selected.

5.1 Comparison Between State-of-the-art Methods

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

[3]We disable hyperthreading in the machine.

- **Naive** CDN: We parallelize loops in the CDN implementation of LIBLINEAR [6], a popular linear classification package. For LR we use an earlier version 1.7 because in the current LIBLINEAR the solver has been changed to newGLMNET. Results of parallelizing newGLMNET are in Section 5.4.
- **Shotgun** CDN [2]: Although the code is publicly available at https://github.com/akyrola/shotgun, 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 (https://github.com/bianan/ParallelCDN) 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.

Following [2], we measure parallel algorithms by the speedup defined as

$$\frac{\text{Time to reach } 1.005\times f^* \text{ with 1 thread}}{\text{Time to reach } 1.005\times f^* \text{ with multiple threads}}, \tag{18}$$

where f^* is the objective value 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 (18).

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 and 16 threads are used. This result is consistent with earlier findings in asynchronous CD experiments [4] (for dual L2-regularized problems), where they showed 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 appendix and 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, $\exp(\cdot)/\log(\cdot)$ operations are involved in (12)-(14). Because each $\exp(\cdot)/\log(\cdot)$ 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 (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 The Impact of the Threshold

For naive parallelization, a parameter is the threshold to decide if operations associated with the current feature should be parallelized or not. We investigate its selection by an experiment on the data set `url_combined`.

From the discussion in Section 2, the threshold should not be too small. Nor should it be too large as otherwise very few features' operations are parallelized and the resulting speedup is not good. We present in Table 6 the speedup by varying the threshold from 0 to 100,000. Clearly, the selection of the threshold is not difficult. We observe similar speedup when the threshold ranges from a few hundreds to a few thousands.

5.4 Speedup of newGLMNET for L1-regularized Logistic Regression

In Section 3.3, we have mentioned that newGLMNET is currently the state-of-the-art for L1-regularized logistic regression under a single-core setting. It is thus essential to check if a naive parallelization on the CD procedure in solving the sub-problem (15) can give good speedup.

We show in Table 7 that naive parallelization can achieve a good speedup. However, the speedup of newGLMNET is not as good as CDN for logistic regression. Instead, the speedup is closer to that of CDN for SVM with the squared hinge loss. The reason should be that $\log(\cdot)/\exp(\cdot)$ operations occupy a smaller percentage of the total running time. More precisely, newGLMNET conducts $\log(\cdot)/\exp(\cdot)$ operations only in generating the sub-problem (15), which is closer to (2) with the squared hinge loss.

5.5 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.

We present in Table 8 the 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 consider the setting of always using multiple threads

Data set \ #threads	LR									SVM								
	Naive			Bundle			Shotgun			Naive			Bundle			Shotgun		
	4	8	16	4	8	16	4	8	16	4	8	16	4	8	16	4	8	16
avazu-app	**3.4**	**5.5**	**7.5**	0.7	1.0	1.3	2.5	3.2	2.9	**3.1**	**4.2**	**4.6**	0.3	0.5	0.6	2.5	3.4	2.6
criteo	**3.5**	**5.8**	**7.5**	1.2	1.9	2.6	2.9	4.9	7.5	**3.2**	**4.2**	4.4	1.1	1.5	1.8	2.5	3.9	**4.8**
epsilon	**3.9**	**7.8**	**14.9**	x	x	x	2.0	x	x	**3.9**	**7.8**	**14.8**	x	x	x	2.1	x	x
HIGGS	**3.9**	**7.2**	**12.9**	0.8	0.9	1.0	1.6	x	x	**3.5**	**4.9**	**5.1**	x	x	x	2.0	x	x
kdd2010-a	2.4	3.2	3.5	1.4	2.4	3.4	**2.8**	**5.0**	**8.0**	2.0	2.3	2.2	1.2	1.9	2.3	**2.2**	**3.7**	**5.8**
kdd2012	2.9	4.1	4.8	0.4	0.6	0.7	**4.2**	**5.9**	**7.8**	2.4	3.0	3.0	0.1	0.1	0.2	**3.6**	**4.9**	**5.1**
rcv1_test	**3.4**	**6.0**	**8.5**	x	x	x	2.5	4.5	7.8	**2.4**	**3.3**	3.0	0.1	0.3	0.4	1.3	2.4	**3.6**
splice.t	**3.5**	**6.1**	**9.2**	x	x	x	3.0	4.2	5.3	**3.2**	**5.1**	**6.2**	x	x	x	1.4	2.5	3.6
url_combined	**3.4**	**5.9**	**8.9**	0.8	1.2	1.3	1.4	1.3	2.0	**2.5**	**3.5**	**3.9**	0.1	0.2	0.3	1.1	1.0	1.6
webspam	**3.2**	**5.1**	**6.9**	0.3	0.5	0.9	2.5	3.6	5.7	**2.3**	**3.4**	**4.1**	x	x	x	1.1	2.0	2.7
yahoo-korea	**3.5**	**5.8**	**8.0**	0.3	0.5	0.9	2.5	4.4	7.7	**2.3**	2.8	2.7	x	0.1	0.2	1.8	**3.1**	**5.2**

Table 3: Speedup of CDN-based methods by using 4, 8, and 16 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	16	2	4	8	16	2	4	8	16	2	4	8	16
avazu-app	1.9	3.4	5.5	7.5	1.8	3.3	5.3	7.0	1.8	3.1	4.2	4.6	1.8	3.0	3.9	4.1
criteo	1.9	3.5	5.8	7.5	1.9	3.4	5.3	6.9	1.9	3.2	4.2	4.4	1.8	3.1	4.0	4.1
kdd2010-a	1.7	2.4	3.2	3.5	1.3	1.6	1.8	1.8	1.5	2.0	2.3	2.2	1.2	1.4	1.4	1.4
kdd2012	1.8	2.9	4.1	4.8	1.4	1.9	2.3	2.4	1.6	2.4	3.0	3.0	1.3	1.6	1.8	1.8
rcv1_test	1.9	3.4	6.0	8.5	1.8	3.3	5.6	7.7	1.6	2.4	3.3	3.0	1.5	2.3	3.1	2.8
splice.t	1.9	3.5	6.1	9.2	1.2	1.3	1.3	1.3	1.8	3.2	5.1	6.2	1.1	1.2	1.2	1.1
url_combined	1.9	3.4	5.9	8.9	1.8	3.2	5.2	7.2	1.7	2.5	3.5	3.9	1.5	2.0	2.5	2.6
webspam	1.8	3.2	5.1	6.9	1.7	2.7	3.9	4.5	1.5	2.3	3.4	4.1	1.4	1.9	2.3	2.3
yahoo-korea	1.9	3.5	5.8	8.0	1.7	2.8	4.0	4.5	1.6	2.3	2.8	2.7	1.3	1.7	1.9	1.8

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

(i.e., 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 (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. An exception is the primal CD discussed in Section 3, where the extremely skewed distribution of features' non-zeros in real-world sparse sets makes the naive parallelization highly effective.

6 CONCLUSIONS

CD has been a state-of-the-art single-thread algorithm for L1-regularized linear classification, but because of its sequential nature, effective multi-core parallelization is not easy. Surprisingly, a solution is 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, helps the parallelization. We retain the same convergence property and achieve excellent speedup without modifying the CD algorithm at all. For future works, we are developing techniques to cover the few data sets that have less skewed distributions.

REFERENCES

[1] Yatao Bian, Xiong Li, Mingqi Cao, and Yuncai Liu. 2013. Bundle CDN: a highly parallelized approach for large-scale l1-regularized logistic regression. In *Proceedings of European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML/ PKDD)*.

[2] Joseph K. Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. 2011. Parallel coordinate descent for l1-regularized loss minimization. In *Proceedings of the Twenty Eighth International Conference on Machine Learning (ICML)*. 321–328.

[3] Kai-Wei Chang, Cho-Jui Hsieh, and Chih-Jen Lin. 2008. Coordinate Descent Method for Large-scale L2-loss Linear SVM. *Journal of Machine Learning Research* 9 (2008), 1369–1398. http://www.csie.ntu.edu.tw/~cjlin/papers/cdl2.pdf

[4] Wei-Lin Chiang, Mu-Chu Lee, and Chih-Jen Lin. 2016. Parallel dual coordinate descent method for large-scale linear classification in multi-core environments. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. http://www.csie.ntu.edu.tw/~cjlin/papers/multicore_cddual.pdf

[5] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5 (1998), 46–55.

[6] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: a library for large linear classification. *Journal of Machine Learning Research* 9 (2008), 1871–1874. http://www.csie.ntu.edu.tw/~cjlin/papers/liblinear.pdf

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

[8] Cho-Jui Hsieh, Kai-Wei Chang, Chih-Jen Lin, S. Sathiya Keerthi, and Sellamanickam Sundararajan. 2008. A dual coordinate descent method for large-scale linear SVM. In *Proceedings of the Twenty Fifth International Conference on Machine Learning (ICML)*. http://www.csie.ntu.edu.tw/~cjlin/papers/cddual.pdf

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.t	#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.

Threshold	SVM			LR		
	4	8	16	4	8	16
0	2.9	2.8	1.7	1.0	0.6	0.3
50	3.5	6.0	8.4	2.5	3.2	2.7
100	3.5	6.1	8.8	2.5	3.3	3.0
200	3.5	6.1	9.0	2.5	3.7	3.5
500	3.5	6.0	8.8	2.6	3.5	3.9
1000	3.4	5.8	8.7	2.5	3.7	4.3
2000	3.3	5.5	8.1	2.5	3.6	4.1
3000	3.3	5.3	7.5	2.5	3.5	3.9
5000	3.2	5.0	6.8	2.5	3.3	4.0
10000	3.0	4.5	5.8	2.2	3.0	3.4
50000	2.5	3.4	4.1	1.9	2.5	2.8
100000	2.3	3.0	3.6	1.9	2.1	2.5

Table 6: The impact of the threshold on the speedup of naive parallelization of CDN**. The data set** `url_combined` **is considered.**

Data set \ #threads	4	8	16
avazu-app	3.1	4.1	4.4
criteo	3.1	4.1	4.3
epsilon	3.8	6.5	8.3
HIGGS	3.5	4.5	4.6
kdd2010-a	2.1	2.5	2.5
kdd2012	2.5	3.0	3.0
rcv1_test	2.2	3.1	3.6
splice.t	2.2	2.6	2.8
url_combined	2.9	4.0	4.5
webspam	2.5	3.7	4.5

Table 7: The speedup of naive parallelization on newGLMNET**.**

Data set \ #threads		4	8	16
sparse sets	avazu-app	0.3	0.2	0.1
	criteo	0.3	0.2	0.1
	url_combined	0.4	0.3	0.2
dense sets	epsilon	1.3	1.1	0.6
	splice.t	2.9	4.1	4.2
	webspam	2.5	2.9	2.6

Table 8: 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 consider the setting of always using multiple threads.**

[9] Fang-Lan Huang, Cho-Jui Hsieh, Kai-Wei Chang, and Chih-Jen Lin. 2010. Iterative Scaling and Coordinate Descent Methods for Maximum Entropy. *Journal of Machine Learning Research* 11 (2010), 815–848. http://www.csie.ntu.edu.tw/~cjlin/papers/maxent_journal.pdf

[10] Mu-Chu Lee, Wei-Lin Chiang, and Chih-Jen Lin. 2015. Fast Matrix-vector Multiplications for Large-scale Logistic Regression on Shared-memory Systems. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*. http://www.csie.ntu.edu.tw/~cjlin/papers/multicore_liblinear_icdm.pdf

[11] Chih-Jen Lin, Ruby C. Weng, and S. Sathiya Keerthi. 2008. Trust region Newton method for large-scale logistic regression. *Journal of Machine Learning Research* 9 (2008), 627–650. http://www.csie.ntu.edu.tw/~cjlin/papers/logistic.pdf

[12] Dong C. Liu and Jorge Nocedal. 1989. On the limited memory BFGS method for large scale optimization. *Mathematical Programming* 45, 1 (1989), 503–528.

[13] Ji Liu and Stephen J Wright. 2015. Asynchronous Stochastic Coordinate Descent: Parallelism and Convergence Properties. *SIAM Journal on Optimization* 25, 1 (2015), 351–376.

[14] Ji Liu, Stephen J Wright, Christopher Ré, Victor Bittorf, and Srikrishna Sridhar. 2015. An Asynchronous Parallel Stochastic Coordinate Descent Algorithm. *Journal of Machine Learning Research* 16, 1 (2015), 285–322.

[15] Peter Richtárik and Martin Takáč. 2016. Parallel coordinate descent methods for big data optimization. *Mathematical Programming* 156, 1-2 (2016), 433–484.

[16] Shai Shalev-Shwartz and Ambuj Tewari. 2011. Stochastic Methods for l1-Regularized Loss Minimization. *Journal of Machine Learning Research* 12 (2011),

1865–1892.
[17] Hsiang-Fu Yu, Cho-Jui Hsieh, Kai-Wei Chang, and Chih-Jen Lin. 2012. Large linear classification when data cannot fit in memory. *ACM Transactions on Knowledge Discovery from Data* 5, 4 (February 2012), 23:1–23:23. http://www.csie.ntu.edu.tw/~cjlin/papers/kdd_disk_decomposition.pdf
[18] Guo-Xun Yuan, Kai-Wei Chang, Cho-Jui Hsieh, and Chih-Jen Lin. 2010. A Comparison of Optimization Methods and software for Large-scale L1-regularized Linear Classification. *Journal of Machine Learning Research* 11 (2010), 3183–3234. http://www.csie.ntu.edu.tw/~cjlin/papers/l1.pdf
[19] Guo-Xun Yuan, Chia-Hua Ho, and Chih-Jen Lin. 2012. An Improved GLMNET for L1-regularized Logistic Regression. *Journal of Machine Learning Research* 13 (2012), 1999–2030. http://www.csie.ntu.edu.tw/~cjlin/papers/l1_glmnet/long-glmnet.pdf
[20] Sangwoon Yun and Kim-Chuan Toh. 2011. A Coordinate Gradient Descent Method for L1-regularized Convex Minimization. *Computational Optimizations and Applications* 48, 2 (2011), 273–307.
[21] Huan Zhang and Cho-Jui Hsieh. 2016. Fixing the Convergence Problems in Parallel Asynchronous Dual Coordinate Descent. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*. 619–628.
[22] Tong Zhang and Frank J. Oles. 2001. Text Categorization Based on Regularized Linear Classification Methods. *Information Retrieval* 4, 1 (2001), 5–31.

A MORE STUDY ON BUNDLE CDN

In Section A.1, we describe the difficulty of the selection of the bundle size in Bundle CDN [1]. We then extensively study the efficacy of Bundle CDN on sparse and dense data sets in Section A.2. In Section A.3, we study the `atomic` operation in Bundle CDN. In Section **??** of supplementary materials, we explain a line search trick that can be applied in Naive CDN but cannot be applied in Bundle CDN. Though our experiments are conducted on logistic regression, we believe the same conclusion should hold for SVMs.

A.1 The Difficulty of the Selection of the Bundle Size in Bundle CDN

The bundle size $|B|$ can be as large as n, but the diagonal approximation in (17) may result in a poor direction. In experiments in Section 5 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 convergence 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, a formulation similar to (15) by updating $|B|$ elements shows that the following vector indicating the change of $\boldsymbol{w}^T\boldsymbol{x}_i,\ \forall i$ must be obtained

$$\sum\nolimits_{j\in B} d_j\bar{\boldsymbol{x}}_j. \tag{19}$$

Because cores may simultaneously update the same entry in (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 (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.

A.2 Bundle CDN on Sparse and Dense Data Sets

In each epoch, Bundle CDN randomly splits n features to $n/|B|$ bundles, and for each bundle, Algorithm 3 summarizes that three operations `memset`, `find_d`, and `line_search` are performed. Following the discussion in Section A.1 on the line-search cost, the time complexity in an epoch is:

- `memset`: $\frac{n}{|B|}\times l$
- `find_d`: non-zeros in the data set
- `line_search`: $\frac{n}{|B|}\times l\times$ line-search steps

Clearly, the bundle size cannot be very small, otherwise $\frac{n}{|B|}\times l$ will be very close to $O(ln)$, which is much larger than $O(\text{nnz})$ for going over all features once in CDN. Therefore, a large bundle size is required to make this method feasible on sparse data sets. However, the convergence of Bundle CDN may get worse when we increase the bundle size, because the larger the bundle size is, the approximation H in (17) gets further from $\nabla^2_{BB}L(\boldsymbol{w})$. We verify this argument by conducting the following experiment. We run Bundle CDN on `kdd2010-a` and `url_combined` with different bundle sizes. Then we check the number of epochs needed to reach a manually specified objective value. Further, we record the time spent on `memset`, `find_d`, and `line_search`. The experimental result in Table 9 reveals the following observations when we increase the bundle size:

- More epochs are needed to reach the target objective value.
- The running time of `find_d` is proportional to the number of epochs.
- Because the running time of `memset` and `line_search` per epoch decreases when we increase the bundle size, their total time may decrease even if more epochs are needed.

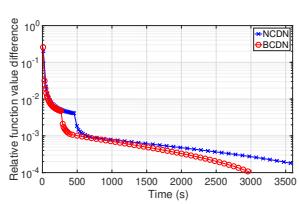
Therefore, to maximize the performance of Bundle CDN, the bundle size must be carefully selected.

The next question is that if the bundle size is carefully selected, can Bundle CDN outperform Naive CDN? To answer this question, we run Bundle CDN with the best bundle size obtained in Table 9, and compare with Naive CDN. For both methods, 16 threads are used. Note that because we are interested in the convergence of Bundle CDN in the later stage, we do not impose the default LIBLINEAR stopping criterion as we did in Section 5. Instead, we let the two methods run for a long time (3,600 seconds), and report the relative function value difference defined as follows.

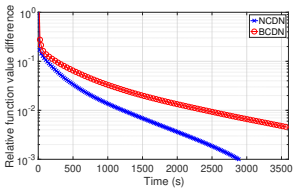
$$\frac{|f(\boldsymbol{w})-f(\boldsymbol{w}^*)|}{|f(\boldsymbol{w}^*)|},$$

where the reference optimal $\boldsymbol{w}^*$ is the minimal objective function value of Bundle CDN and Naive CDN. The experimental results are shown in Figures 2a and 2b. In `url_combined`, Naive CDN outperforms Bundle CDN all the time. For `kdd2010-a`, Bundle CDN with bundle size 29,500 is overall faster than Naive CDN. Note that this figure seems to be inconsistent with Table 3, where the speedup of Naive CDN is comparable to Bundle CDN. This is because Table 3 corresponds to only the situation in Figure 2a when the relative function-value difference is around 10^{-2}

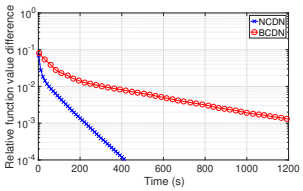
For dense data, because the number of non-zeros equals $n\times l$, `memset` and `line_search` may no longer be the bottleneck if a small bundle size is used. Table 10 is the analysis of different bundle sizes. We still see similar behavior as in the sparse data sets when the bundle size increases. The experiment comparing the convergence



(a) `kdd2010-a` (bundle size 29,500)



(b) `url_combined` (bundle size 5,000)



(c) `HIGGS` (bundle size 16)

Figure 2: Comparison between Naive CDN **(NCDN) and Bundle** CDN **(BCDN) on sparse data sets `kdd2010-a` and `url_combined`, and on a dense set `HIGGS`. We use 16 threads for sparse sets, while 8 for dense sets.**

Algorithm 3 One epoch (scan data once) of Bundle CDN.

Randomly split $\{1, \ldots, n\}$ to $n/|B|$ bundles
for each bundle **do**
  `memset`:
    Initialize $\boldsymbol{b}'$ as a zero vector to store (19)
  `find_d`:
    Solve (16)
  `line_search`:
    Perform a line search step

of Bundle CDN and Naive CDN is in Figure 2c, which shows that Bundle CDN does not out-perform Naive CDN.

A.3 Effect of the `Atomic` Operation

The implementation of `find_d` involves an `atomic` operation to ensure (19) can be safely calculated by multiple threads. However, a recent paper [10] shows that the `atomic` operation may hurt the speedup. In Table 11, we investigate this problem by checking the running time spent on `memset`, `find_d`, and `line_search` on a sparse data set `kdd2010-a` and a dense data set `HIGGS`. On the sparse data set, though the speedup of `find_d` is not as excellent as `line_search`, it is still pretty good. In contrast, on the dense data set, `find_d` has no speedup. Therefore, we expect that if we apply the technique developed in [10], then we may get a better speedup on dense data sets. We leave this as a future work.

		time			
$\|B\|$	epochs	total	memset	find_d	line_search
5000	32	942	97	194	650
10000	34	769	50	204	514
20000	39	691	26	230	433
29500	39	602	18	231	353
50000	52	760	12	305	442

(a) `kdd2010-a`

		time			
$\|B\|$	epochs	total	memset	find_d	line_search
1000	16	394	28	70	295
5000	28	389	8	120	261
10000	53	524	4	222	298
50000	112	751	1	464	285
100000	91	637	1	387	248
500000	194	1220	1	854	365

(b) `url_combined`

Table 9: Running time of Bundle CDN **with different bundle sizes ($|B|$) on two sparse data sets `kdd2010-a` and `url_combined`. We include the size 29,500 in `kdd2010-a` because it is the best bundle size reported in [1]. Time is in seconds.**

		time			
$\|B\|$	epochs	total	memset	find_d	line_search
1	16	316	4	57	253
2	16	197	2	58	136
4	20	180	1	72	106
8	23	184	1	82	100
16	43	313	0	154	157
24	37	240	0	134	105
28	100	634	1	360	272

Table 10: Running time of Bundle CDN **with different bundle size ($|B|$) on a dense data set `HIGGS`.**

	time			
#threads	total	memset	find_d	line_search
1	1865	34	765	1065
16	213	7	117	87
speedup	8.8	4.9	6.5	12.2

(a) `kdd2010-a`

	time			
#threads	total	memset	find_d	line_search
1	1842	6	1000	836
16	962	1	900	59
speedup	1.9	6.0	1.1	14.2

(b) `HIGGS`

Table 11: The time (in seconds) consumed and the speedup of `memset`, `find_d`, and `line_search` at a given epoch by using 1 and 16 threads.