Parallel Dual Coordinate Descent Method for Large-scale Linear Classification in Multi-core Environments

Wei-Lin Chiang
Dept. of Computer Science
National Taiwan Univ., Taiwan
b02902056@ntu.edu.tw

Mu-Chu Lee
Dept. of Computer Science
National Taiwan Univ., Taiwan
b01902082@ntu.edu.tw

Chih-Jen Lin
Dept. of Computer Science
National Taiwan Univ., Taiwan
cjlin@csie.ntu.edu.tw

ABSTRACT

Dual coordinate descent method is one of the most effective approaches for large-scale linear classification. However, its sequential design makes the parallelization difficult. In this work, we target at the parallelization in a multi-core environment. After pointing out difficulties faced in some existing approaches, we propose a new framework to parallelize the dual coordinate descent method. The key idea is to make the majority of all operations (gradient calculation here) parallelizable. The proposed framework is shown to be theoretically sound. Further, we demonstrate through experiments that the new framework is robust and efficient in a multi-core environment.

Keywords

dual coordinate descent, linear classification, multi-core computing

1. INTRODUCTION

Linear classification such as linear SVM and logistic regression is one of the most used machine learning methods. However, training large-scale data may be time-consuming, so the parallelization has been an important research issue. In this work, we consider multi-core environments and study parallel dual coordinate descent methods, which are an important class of optimization methods to train large-scale linear classifiers.

Existing optimization methods for linear classification can be roughly categorized to the following two types:

1. Low-order optimization methods such as stochastic gradient or coordinate descent (CD) methods. By using only the gradient information, this type of methods runs many cheap iterations.
2. High-order optimization methods such as quasi Newton or Newton methods. By using, for example, second-order information, each iteration is expensive but fewer iterations are needed to approach the final solution.

KDD '16, August 13 - 17, 2016, San Francisco, CA, USA

ISBN 978-1-4503-4232-2/16/08...$15.00

DOI: http://dx.doi.org/10.1145/2939672.2939826

These methods, useful in different circumstances, have been parallelized in some past works. To be focused here, we restrict our discussion to those that are suitable for multi-core environments. Therefore, some that are mainly applicable in distributed environments are out of our interests.

For Newton methods, recently we have shown that with careful implementations, excellent speedup can be achieved in a multi-core environment [11]. Its success relies on parallelizable operations that involve all data together. In contrast, stochastic gradient or CD methods are inherently sequential because each time only one instance is used to update the model. Among approaches of using low-order information, we are particularly interested in the CD method to solve the dual optimization problem. Although such techniques can be traced back to works such as [4], after the recent introduction to linear classification [5], dual CD has become one of the most efficient methods. Further, in contrast to primal-based methods (e.g., Newton or primal CD) that often require the differentiability of the loss function, a dual-based method can easily handle some non-differentiable losses such as the $l1$ hinge loss (i.e., linear SVM).

Several works have proposed parallel extensions of dual CD methods (e.g., [6, 10, 13, 14, 15]), in which [6, 13, 14] focus more on multi-core environments. We can further categorize them to two types:

1. Mini-batch CD [13]. Each time a batch of instances are selected and CD updates are parallelly applied to them.
2. Asynchronous CD [6][14]. Threads independently update different coordinates in parallel. The convergence is often faster than synchronous algorithms, but sometimes the algorithm fails to converge.

In Section 2, we detailedly discuss the above approaches for parallel dual CD, and explain why they may be either inefficient or not robust. Indeed, except the experiment code in [6], so far no publicly available packages have supported parallel dual CD in multi-core environments. In Section 3, we propose a new and simple framework that can effectively take the advantage of multi-core computation. Theoretical properties such as asymptotic convergence and finite termination under given stopping tolerances are provided in Section 4. In Section 5, we conduct thorough experiments and comparisons. Results show that our proposed method is robust and efficient.

Based on this work, parallel dual CD is now publicly available in the multi-core extension of our LIBLINEAR package: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/multicore-liblinear/. Because of space limitation, proofs and some additional experimental results are left in supplementary materials at the same address. Code for experiments is also available there.

2. DUAL COORDINATE DESCENT AND DIFFICULTIES OF ITS PARALLELIZATION

In this section, we begin with introducing optimization problems for linear classification and the basic concepts of dual CD methods. Then we discuss difficulties of the parallelization in multi-core environments.

2.1 Linear Classification and Dual CD Methods

Assume the classification task involves a training set of instance-label pairs $(\boldsymbol{x}_i, y_i), i = 1, \ldots, l$, $\boldsymbol{x}_i \in R^n$, $y_i \in \{-1, +1\}$, a linear classifier obtains its model vector $\boldsymbol{w}$ by solving the following optimization problem.

$$\min_{\boldsymbol{w}} \quad \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{l}\xi(\boldsymbol{w}; \boldsymbol{x}_i, y_i), \tag{1}$$

where $\xi(\boldsymbol{w}; \boldsymbol{x}_i, y_i)$ is a loss function, and $C > 0$ is a penalty parameter. Commonly used loss functions include

$$\xi(\boldsymbol{w}; \boldsymbol{x}, y) \equiv \begin{cases} \max(0, 1 - y\boldsymbol{w}^T\boldsymbol{x}) & l1 \text{ loss}, \\ \max(0, 1 - y\boldsymbol{w}^T\boldsymbol{x})^2 & l2 \text{ loss}, \\ \log(1 + e^{-y\boldsymbol{w}^T\boldsymbol{x}}) & \text{logistic (LR) loss}. \end{cases}$$

In this work, we focus on $l1$ and $l2$ losses (i.e., linear SVM), though results can be easily applied to logistic regression. Following the notation in [5], if (1) is referred to as the primal problem, then a dual CD method solves the following dual problem:

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & f(\boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\alpha}^T\bar{Q}\boldsymbol{\alpha} - \boldsymbol{e}^T\boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \le \alpha_i \le U, \forall i, \end{aligned} \tag{2}$$

where $\bar{Q} = Q + D$, D is a diagonal matrix, and $Q_{ij} = y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j$. For the $l1$ loss, $U = C$ and $D_{ii} = 0$, $\forall i$, while for the $l2$ loss, $U = \infty$ and $D_{ii} = 1/(2C)$, $\forall i$. Notice that $l1$ loss is not differentiable, so solving the dual problem is generally easier than the primal.

We briefly review dual CD methods by following the description in [5]. Each time a variable α_i is updated while others are fixed. Specifically, if the current $\boldsymbol{\alpha}$ is feasible for (2), we solve the following one-variable sub-problem:

$$\min_{d} f(\boldsymbol{\alpha} + d\boldsymbol{e}_i) \text{ subject to } 0 \le \alpha_i + d \le U, \tag{3}$$

where $\boldsymbol{e}_i = [\underbrace{0, \ldots, 0}_{i-1}, 1, 0, \ldots, 0]^T$. Clearly,

$$f(\boldsymbol{\alpha} + d\boldsymbol{e}_i) = \frac{1}{2}\bar{Q}_{ii}d^2 + \nabla_i f(\boldsymbol{\alpha})d + \text{constant}, \tag{4}$$

where

$$\nabla_i f(\boldsymbol{\alpha}) = (\bar{Q}\boldsymbol{\alpha})_i - 1 = \sum_{j=1}^{l}\bar{Q}_{ij}\alpha_j - 1.$$

If $\bar{Q}_{ii} > 0$,[1] the solution of (3) can be easily seen as

$$d = \min\left(\max\left(\alpha_i - \frac{\nabla_i f(\boldsymbol{\alpha})}{\bar{Q}_{ii}}, 0\right), U\right) - \alpha_i. \tag{5}$$

[1]It has been pointed out in [5] that $\bar{Q}_{ii} = 0$ occurs only when $\boldsymbol{x}_i = \boldsymbol{0}$ and the $l1$ loss is used. Then $\bar{Q}_{ij} = 0, \forall j$ and the optimal $\alpha_i = C$. This variable can thus be easily removed before running CD.

Algorithm 1 A dual CD method for linear SVM

1: Specify a feasible $\boldsymbol{\alpha}$ and calculate $\boldsymbol{w} = \sum_j y_j\alpha_j\boldsymbol{x}_j$
2: **while** $\boldsymbol{\alpha}$ is not optimal **do**
3: **for** $i = 1, \ldots, l$ **do**
4: $G \leftarrow y_i\boldsymbol{w}^T\boldsymbol{x}_i - 1 + D_{ii}\alpha_i$
5: $d \leftarrow \min(\max(\alpha_i - G/\bar{Q}_{ii}, 0), U) - \alpha_i$
6: $\alpha_i \leftarrow \alpha_i + d$
7: $\boldsymbol{w} \leftarrow \boldsymbol{w} + dy_i\boldsymbol{x}_i$

Algorithm 2 Mini-batch dual CD in [13]

1: Specify $\boldsymbol{\alpha} = \boldsymbol{0}$, batch size b, and a value $\beta_b > 1$.
2: **while** α is not optimal **do**
3: Get a set B with $|B| = b$ under uniform distribution
4: $\boldsymbol{w} = \sum_j y_j\alpha_j\boldsymbol{x}_j$
5: **for all** $i \in B$ **do in parallel**
6: $G = y_i\boldsymbol{w}^T\boldsymbol{x}_i - 1 + D_{ii}\alpha_i$
7: $\alpha_i \leftarrow \min(\max(\alpha_i - G/(\beta_b \times \bar{Q}_{ii}), 0), U)$

The main computation in (5) is on calculating $\nabla_i f(\boldsymbol{\alpha})$. One crucial observation in [5] is that

$$\sum_{j=1}^{l}\bar{Q}_{ij}\alpha_j - 1 = y_i(\sum_{j=1}^{l} y_j\alpha_j\boldsymbol{x}_j)^T\boldsymbol{x}_i - 1 + D_{ii}\alpha_i.$$

If

$$\boldsymbol{w} \equiv \sum_{j=1}^{l} y_j\alpha_j\boldsymbol{x}_j \tag{6}$$

is maintained, then $\nabla_i f(\boldsymbol{\alpha})$ can be easily calculated by

$$\nabla_i f(\boldsymbol{\alpha}) = y_i\boldsymbol{w}^T\boldsymbol{x}_i - 1 + D_{ii}\alpha_i. \tag{7}$$

Note that we slightly abuse the notation by using the same symbol $\boldsymbol{w}$ of the primal variable in (1). The reason is that $\boldsymbol{w}$ in (6) will become the primal optimum if $\boldsymbol{\alpha}$ converges to a dual optimal solution. We can then update $\boldsymbol{\alpha}$ and maintain the weighted sum in (6) by

$$\alpha_i \leftarrow \alpha_i + d \text{ and } \boldsymbol{w} \leftarrow \boldsymbol{w} + dy_i\boldsymbol{x}_i. \tag{8}$$

This is much cheaper than calculating the sum of l vectors in (6). The simple CD procedure of cyclically updating $\alpha_i, i = 1, \ldots, l$ is presented in Algorithm 1. We call each iteration of the **while** loop as an outer iteration. Thus each outer iteration contains l inner iterations to sequentially update all $\boldsymbol{\alpha}$'s components. Further, the main computation at each inner iteration includes two $O(n)$ operations in (7) and (8).

The above $O(n)$ operations are by assuming that the data set is dense. For sparse data, any $O(n)$ term in the complexity discussion in this paper should be replaced by $O(\bar{n})$, where $\bar{n}$ is the average number of non-zero feature values per instance.

2.2 Difficulties in Parallelizing Dual CD

We point out difficulties to parallelize dual CD methods by discussing two types of existing approaches.

2.2.1 Mini-batch Dual CD

Algorithm 1 is inherently sequential. Further, it contains many cheap inner iterations, each of which cost $O(n)$ operations. Some [13] thus propose applying CD updates on a batch of data simultaneously. Their procedure is summarized in Algorithm 2

Algorithms 1 and 2 differ in several places. First, in Algorithm 2 we must select a set B. In [13], this set is randomly

selected under a distribution, so the algorithm is a stochastic dual CD. If we would like a cyclic setting similar to that in Algorithm 1, a simple way is to split all data $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l\}$ to blocks and then update variables associated with each block in parallel. The second and also the main difference from Algorithm 1 is that (5) cannot be used to update $\alpha_i, \forall i \in B$. The reason is that we no longer have the property that all but one variable are fixed. To update all $\alpha_i, i \in B$ in parallel but maintain the convergence, the change on each coordinate must be conservative. Therefore, they consider an approximation of the one-variable problem (3) by replacing $\bar{Q}_{ii}$ in (4) with a larger value $\beta_b \times \bar{Q}_{ii}$; see line 7 of Algorithm 2. By choosing a suitable β_b that is data dependent, [13] proved the expected convergence. One disadvantage of using conservative steps is the slower convergence. Therefore, asynchronous CD methods that will be discussed later aim to address this problem by still using the sub-problem (3).

An important practical issue not discussed in [13] is the calculation of $\boldsymbol{w}$. In Algorithm 2, we can see that they recalculate $\boldsymbol{w}$ at every iteration. This operation becomes the bottleneck because it is much more time-consuming than the update of $\alpha_i, \forall i \in B$. Following the setting in Algorithm 1, what we should do is to maintain $\boldsymbol{w}$ according to the change of $\boldsymbol{\alpha}$. Therefore, lines 5-7 in Algorithm 2 can be changed to

1: **for all** $i \in B$ **do in parallel**
2: $\quad G \leftarrow y_i \boldsymbol{w}^T \boldsymbol{x}_i - 1 + D_{ii}\alpha_i$
3: $\quad d_i \leftarrow \min(\max(\alpha_i - G/(\beta_b \times \bar{Q}_{ii}), 0), U) - \alpha_i$
4: $\quad \alpha_i \leftarrow \alpha_i + d_i$
5: $\boldsymbol{w} \leftarrow \boldsymbol{w} + \sum_{j:j\in B} y_j d_j \boldsymbol{x}_j$

We notice that both the **for** loop (line 1) and the update of $\boldsymbol{w}$ (line 5) take $O(|B|\, n)$ operations. Thus parallelizing the **for** loop can at best half the running time. Updating $\boldsymbol{w}$ in parallel is possible, but we explain that it is much more difficult than the parallel calculation of $d_i, \forall i \in B$. The main issue is that two threads may want to update the same component of $\boldsymbol{w}$ simultaneously. The following example shows that one thread for $\boldsymbol{x}_i$ and another thread for $\boldsymbol{x}_j$ both would like to update w_s:

$$w_s \leftarrow w_s + y_i d_i (\boldsymbol{x}_i)_s \text{ and } w_s \leftarrow w_s + y_j d_j (\boldsymbol{x}_j)_s.$$

The recent work [11] has detailedly studied this issue. One way to avoid the race condition is by atomic operations, so each w_s is updated by only one thread at a time:

1: **for all** $i \in B$ **do in parallel**
2: $\quad$ Calculate G, obtain d_i and update α_i
3: $\quad$ **for** $(\boldsymbol{x}_i)_s \neq 0$ **do**
4: $\quad\quad$ atomic: $w_s \leftarrow w_s + y_i d_i (\boldsymbol{x}_i)_s$

Unfortunately, in some situations (e.g., number of features is small) atomic operations cause significant waiting time so that no speedup is observed [11]. Instead, for calculating the sum of some vectors

$$u_1 \boldsymbol{x}_1 + \cdots + u_l \boldsymbol{x}_l,$$

the study in [11] shows better speedup by storing temporary results of each thread in the following vector

$$\hat{\boldsymbol{u}}^p = \sum \{u_i \boldsymbol{x}_i \mid \boldsymbol{x}_i \text{ handled by thread } p\} \tag{9}$$

and parallelly summing these vectors in the end. This approach essentially implements a reduce operation in parallel computation. However, it is only effective when enough vectors are summed because otherwise the overhead of maintaining all $\hat{\boldsymbol{u}}^p$ vectors leads to no speedup. Unfortunately, B is now a small set, so this approach of implementing a reduce operation may not be useful.

In summary, through the discussion we point out that the update of $\boldsymbol{w}$ may be a bottleneck in parallelzing dual CD.

2.2.2 Asynchronous Dual CD

To address the conservative updates in parallel mini-batch CD, a recent direction is by asynchronous updates [6], [14]. Under a stochastic setting to choose variables, each thread *independently* updates an α_i by the rule in (5):

1: **while** $\boldsymbol{\alpha}$ is not optimal **do**
2: $\quad$ Select a set B
3: $\quad$ **for all** $i \in B$ **do in parallel**
4: $\quad\quad G \leftarrow y_i \boldsymbol{w}^T \boldsymbol{x}_i - 1 + D_{ii}\alpha_i$
5: $\quad\quad d_i \leftarrow \min(\max(\alpha_i - G/\bar{Q}_{ii}, 0), U) - \alpha_i$
6: $\quad\quad \alpha_i \leftarrow \alpha_i + d_i$
7: $\quad\quad$ **for** $(\boldsymbol{x}_i)_s \neq 0$ **do**
8: $\quad\quad\quad$ atomic: $\boldsymbol{w}_s \leftarrow \boldsymbol{w}_s + d_i y_i (\boldsymbol{x}_i)_s$

To avoid the conflicts in updating $\boldsymbol{w}$, they consider atomic operations. From the discussion in Section 2.2.1, one may worry that such operations cause serious waiting time, but [6], [14] report good speedup. A detailed analysis on the use of atomic operations here was in [11, supplement], where we point out that practically each thread updates $\boldsymbol{w}$ (line 8 of the above algorithm) by the following setting:

1: **if** $d_i \neq 0$ **then**
2: $\quad$ **for** $(\boldsymbol{x}_i)_s \neq 0$ **do**
3: $\quad\quad$ atomic: $\boldsymbol{w}_s \leftarrow \boldsymbol{w}_s + d_i y_i (\boldsymbol{x}_i)_s$

For linear SVM, some $\boldsymbol{\alpha}$ elements may quickly reach bounds (0 or C for $l1$ loss and 0 for $l2$ loss) and remain the same. The corresponding $d_i = 0$ so the atomic operation is not needed after calculating $G = \nabla_i f(\boldsymbol{\alpha})$. Therefore, the atomic operations that may cause troubles occupy a relatively small portion of the total computation. However, for dense problems because most $\boldsymbol{x}_i$'s elements are non-zero, the race situation more frequently occurs. Hence experiments in Section 5 show worse scalability.

The major issue of using an asynchronous setting is that the convergence may not hold. Both works [6], [14] assume that the lag between the start (i.e., reading $\boldsymbol{x}_i$) and the end (i.e., updating $\boldsymbol{w}$) of one CD step is bounded. Specifically, if we denote the update by a thread as an iteration and order these iterations according to their finished time, then the resulting sequence $\{\boldsymbol{\alpha}^k\}$ should satisfy that

$$k \leq \bar{k} + \tau,$$

where $\bar{k}$ is the iteration index when iteration k starts, and τ is a positive constant.

Both works require τ to satisfy some conditions for the convergence analysis. Unfortunately, as indicated in Figure 2 of [14], these conditions may not always hold, so the asynchronous dual CD method may not converge. In our experiment, this situation easily occurs for dense data (i.e., most feature values are non-zeros) if more cores are used. To avoid the divergence situation, [14] further proposes a semi-asynchronous dual CD method by having a separate thread to calculate (6) once after a fixed number of CD updates. However, they do not prove the convergence under such a semi-asynchronous setting.

Algorithm 3 A practical implementation of Algorithm 1 considered by LIBLINEAR, where new statements are marked by "◁ new"

1: Specify a feasible $\boldsymbol{\alpha}$ and calculate $\boldsymbol{w} = \sum_j y_j \alpha_j \boldsymbol{x}_j$
2: **while** true **do**
3: $M \leftarrow -\infty$
4: **for** $i = 1, \ldots, l$ **do**
5: $G \leftarrow y_i \boldsymbol{w}^T \boldsymbol{x}_i - 1 + D_{ii}\alpha_i$
6: Calculate PG by (11) ◁ new
7: $M \leftarrow \max(M, |PG|)$ ◁ new
8: **if** $|PG| \geq 10^{-12}$ **then** ◁ new
9: $d \leftarrow \min(\max(\alpha_i - G/\bar{Q}_{ii}, 0), U) - \alpha_i$
10: $\alpha_i \leftarrow \alpha_i + d$
11: $\boldsymbol{w} \leftarrow \boldsymbol{w} + d y_i \boldsymbol{x}_i$
12: **if** $M < \varepsilon$ **then** ◁ new
13: break

3. A FRAMEWORK FOR PARALLEL DUAL CD

Based on the discussion in Section 2, we set the following design goals for a new framework.

1. To ensure the convergence in all circumstances, we do not consider asynchronous updates.
2. Because of the difficulty to parallelly update $\boldsymbol{w}$ (see Section 2.2.1), we run this operation only in a serial setting. Instead, we design the algorithm so that this $\boldsymbol{w}$ update takes a small portion of the total computation. Further, we ensure that the most computationally intensive part is parallelizable.

3.1 Our Idea for Parallelization

To begin, we present Algorithm 3, which is the practical version of Algorithm 1 implemented in the popular linear classifier LIBLINEAR [2]. A difference is that a stopping condition is introduced. If we assume that one outer iteration contains the following inner iterates,

$$\boldsymbol{\alpha}^{k,1}, \boldsymbol{\alpha}^{k,2}, \ldots, \boldsymbol{\alpha}^{k,l},$$

then the stopping condition[2] is

$$\max_i |\nabla_i^P f(\boldsymbol{\alpha}^{k,i})| < \varepsilon, \tag{10}$$

where ε is a given tolerance and $\nabla_i^P f(\boldsymbol{\alpha})$ is the projected gradient defined as

$$\nabla_i^P f(\boldsymbol{\alpha}) = \begin{cases} \nabla_i f(\boldsymbol{\alpha}) & \text{if } 0 < \alpha_i < U, \\ \min(0, \nabla_i f(\boldsymbol{\alpha})) & \text{if } \alpha_i = 0, \\ \max(0, \nabla_i f(\boldsymbol{\alpha})) & \text{if } \alpha_i = U. \end{cases} \tag{11}$$

Notice that for problem (2), $\boldsymbol{\alpha}$ is optimal if and only if

$$\nabla^P f(\boldsymbol{\alpha}) = \mathbf{0}.$$

Another important change made in Algorithm 3 is that at line 8, we check whether $\nabla_i f^P(\boldsymbol{\alpha}) \approx 0$ to see if the current α_i is close to the optimum of the single-variable optimization problem (3). If that is the case, then we update neither α_i nor $\boldsymbol{w}$. Note that updating α_i is cheap, but the check at line 8 may significantly save the $O(n)$ cost to update $\boldsymbol{w}$. Therefore, in practice we may have the following situation

$$\underbrace{\boldsymbol{\alpha}^{k,1}, \ldots, \boldsymbol{\alpha}^{k,s-1}}_{\text{unchanged}}, \boldsymbol{\alpha}^{k,s}, \underbrace{\boldsymbol{\alpha}^{k,s+1}, \ldots, \boldsymbol{\alpha}^{k,s'-1}}_{\text{unchanged}}, \boldsymbol{\alpha}^{k,s'}, \ldots \tag{12}$$

Clearly, the calculation of

$$\nabla_1^P f(\boldsymbol{\alpha}^{k,1}), \ldots, \nabla_{s-1}^P f(\boldsymbol{\alpha}^{k,s-1})$$

is wasted. However, we know these values are close to zero only if we have calculated them.

The above discussion shows that between any two updated $\boldsymbol{\alpha}$ components, several unchanged elements may exist. In fact we may deliberately have more unchanged elements. For example, if at line 8 of Algorithm 3 we instead use the following condition

$$\nabla_i^P f(\boldsymbol{\alpha}^{k,i}) \geq \delta\varepsilon, \text{ where } \delta \in (0, 1) \text{ and } \delta\varepsilon \gg 10^{-12},$$

then many elements may be unchanged between two updated ones. Note that ε is typically larger than 0.001 (0.1 is the default stopping tolerance used in LIBLINEAR) and $\delta \in (0, 1)$ can be chosen not too small (e.g., 0.5).[3]

A crucial observation from (12) is that because

$$\boldsymbol{\alpha}^{k,1} = \cdots = \boldsymbol{\alpha}^{k,s-1},$$

we can calculate their projected gradient values in parallel. Unfortunately, the number s is not known in advance. One solution is to conjecture an interval $\{1, \ldots, I\}$ so we parallelly calculate all corresponding gradient values,

$$\nabla_i f(\boldsymbol{\alpha}^k), i = 1, \ldots, I.$$

This approach ends up with the following situation

$$\underbrace{\nabla_1 f(\boldsymbol{\alpha}^k), \ldots, \overset{\text{selected}}{\overset{\downarrow}{\nabla_s f(\boldsymbol{\alpha}^k)}}}_{\text{checked}}, \underbrace{\nabla_{s+1} f(\boldsymbol{\alpha}^k), \ldots, \nabla_I f(\boldsymbol{\alpha}^k)}_{\text{unchecked \& wasted}} \tag{13}$$

After α_s^k is updated, gradient values become different and hence the calculation for $\nabla_i f(\boldsymbol{\alpha}^k), i = s+1, \ldots, I$ is wasted. Because guessing the size of the interval is extremely difficult, we propose a two-stage approach. We still calculate gradient values of I elements, but select a subset of candidates rather than one single element for CD updates:

Stage 1: We calculate $\nabla_i f(\boldsymbol{\alpha}^k), i = 1, \ldots, I$ *in parallel* and then select some elements for update. The following example shows that after checking all I elements, three of them, $\{s_1, s_2, s_3\}$, are selected; see the difference from (13).

$$\underbrace{\alpha_1^k, \ldots, \overset{\downarrow}{\alpha_{s_1}^k}, \ldots, \overset{\downarrow}{\alpha_{s_2}^k}, \ldots, \overset{\downarrow}{\alpha_{s_3}^k}, \ldots, \alpha_I^k}_{\text{all checked}}$$

Stage 2: We *sequentially* update selected elements (e.g., $\alpha_{s_1}, \alpha_{s_2}$, and α_{s_3} in the above example) by regular CD updates.

The standard CD greedily uses the latest $\nabla_i f(\boldsymbol{\alpha})$ to check if α_i should be updated. In contrast, our setting here relies on the current $\nabla_i f(\boldsymbol{\alpha}), i = 1, \ldots, I$ to check if the next I elements should be updated. When $\boldsymbol{\alpha}$ is close to the optimum and is not changed much, the selection should be as good as the standard CD. Algorithm 4 shows the details of

[2]Note that LIBLINEAR actually uses $\max_i \nabla f(\boldsymbol{\alpha}^{k,i}) - \min_i \nabla f(\boldsymbol{\alpha}^{k,i}) < \varepsilon$, though for simplicity in this paper we consider (10).

[3]Note that we need $\delta \in (0, 1)$ to ensure from the stopping condition (10) that at each outer iteration at least one α_i is updated.

Algorithm 4 A parallel dual CD method

1: Specify a feasible $\boldsymbol{\alpha}$ and calculate $\boldsymbol{w} = \sum_j y_j \alpha_j \boldsymbol{x}_j$
2: Specify a tolerance ε and a small value $0 < \bar{\varepsilon} \ll \varepsilon$
3: **while** true **do**
4: $M \leftarrow -\infty$
5: Split $\{1, \ldots, l\}$ to $\bar{B}_1, \ldots, \bar{B}_T$
6: $\bar{t} \leftarrow 0$
7: **for** $\bar{B}$ in $\bar{B}_1, \ldots, \bar{B}_T$ **do**
8: Calculate $\nabla f_{\bar{B}}(\boldsymbol{\alpha})$ in parallel
9: $M \leftarrow \max(M, \max_{i \in \bar{B}} |\nabla_i^P f(\boldsymbol{\alpha})|)$
10: $B \leftarrow \{i \mid i \in \bar{B}, |\nabla_i^P f(\boldsymbol{\alpha})| \geq \delta\varepsilon\}$
11: **for** $i \in B$ **do**
12: $G \leftarrow y_i \boldsymbol{w}^T \boldsymbol{x}_i - 1 + D_{ii}\alpha_i$
13: $d \leftarrow \min(\max(\alpha_i - G/\bar{Q}_{ii}, 0), U) - \alpha_i$
14: **if** $|d| \geq \bar{\varepsilon}$ **then**
15: $\alpha_i \leftarrow \alpha_i + d$
16: $\boldsymbol{w} \leftarrow \boldsymbol{w} + d y_i \boldsymbol{x}_i$
17: $\bar{t} \leftarrow \bar{t} + 1$
18: **if** $M \leq \varepsilon$ or $\bar{t} = 0$ **then**
19: break

Algorithm 5 A framework of parallel dual CD methods, where Algorithms 4 and 6 are special cases

1: Specify a feasible $\boldsymbol{\alpha}$
2: **while** true **do**
3: Select a set $\bar{B}$
4: Calculate $\nabla_{\bar{B}} f(\boldsymbol{\alpha})$ in parallel
5: Select $B \subset \bar{B}$ with $|B| \ll |\bar{B}|$
6: Update $\alpha_i, i \in B$

our approach. Like the cyclic setting in Algorithm 1, here we split $\{1, \ldots, l\}$ to several blocks. Each time we parallely calculate $\nabla_i f(\boldsymbol{\alpha})$ of elements in a block $\bar{B}$ and then select a subset $B \subset \bar{B}$ for sequential CD updates. Note that line 14 is similar to line 8 in Algorithm 3 for checking if the change of α_i is too small and $\boldsymbol{w}$ needs not be updated.

A practical issue in Algorithm 4 is that the selection of B depends on the given ε. That is, the stopping tolerance specified by users may affect the behavior of the algorithm. We resolve this issue in Section 4 for discussing practical implementations.

3.2 A General Framework for Parallel Dual CD

The idea in Section 3.1 motivates us to have a general framework for parallel dual CD in Algorithm 5, where Algorithm 4 is a special case. The key properties of this framework are:

1. We select a set $\bar{B}$ and calculate the corresponding gradient values *in parallel*.
2. We then get a much smaller set $B \subset \bar{B}$ and update $\boldsymbol{\alpha}_B$.

Assume that updating $\boldsymbol{\alpha}_B$ costs $O(|B|n)$ operations as in Algorithm 4. Then the complexity of Algorithm 5 is

$$O\left(\frac{|\bar{B}|n}{P} + |B|n\right) \times \#\text{iterations},$$

where P is the number of threads. If $|B| \ll |\bar{B}|$, we can see that parallel computation can significantly reduce the running time.

One may argue that Algorithm 5 is no more than a typical block CD method and question why we come a long way to get it. A common block CD method selects a set $\bar{B}$ at a time and solve a sub-problem of the variable $\boldsymbol{\alpha}_{\bar{B}}$. If we consider Algorithm 5 as a block CD method, then it has a very special setting in solving the sub-problem of $\boldsymbol{\alpha}_{\bar{B}}$: Algorithm 5 spends most efforts on further selecting a much smaller subset B and then (approximately or accurately) solving a smaller sub-problem of $\boldsymbol{\alpha}_B$. Therefore, we can say that Algorithm 5 is a specially tweaked block CD that aims for multi-core environments.

3.3 Relation with Decomposition Methods for Kernel SVM

In Algorithm 4, while the second stage is to cyclically update elements in the set B, the first stage is a gradient-based selection of B from a larger set $\bar{B}$. Interestingly, cyclic and gradient-based settings are the two major ways in CD to select variables for update. The use of gradient motivates us to link to the popular decomposition methods for kernel SVM (e.g., [3, 8, 12]), which calculate the gradient and select a small subset of variables for update. It has been explained in [5, Section 4] why a gradient-based rather than a cyclic variable selection is useful for kernel classifiers, so we do not repeat the discussion here. Instead, we would like to discuss the BSVM package [7] that has recognized the importance of maintaining $\boldsymbol{w}$ for the linear kernel;[4] see also [9, Section 4]. After calculating $\nabla f(\boldsymbol{\alpha})$, BSVM selects a small set B (by default $|B| = 10$) by the following procedure. Let r be the number of $\boldsymbol{\alpha}$'s free components (i.e., $0 < \alpha_i < C$), $|B|$ be the number of elements to be selected, and

$$\boldsymbol{v} = -\nabla^P f(\boldsymbol{\alpha}).$$

The set B includes the following indices.

1. The largest $\min(|B|/2, r)$ elements in $\boldsymbol{v}$ that correspond to $\boldsymbol{\alpha}$'s free elements.
2. The smallest $(|B| - \min(|B|/2, r))$ elements in $\boldsymbol{v}$.

BSVM then updates $\boldsymbol{\alpha}_B$ by fixing all other elements and solving the following sub-problem.

$$\begin{aligned} \min_{\boldsymbol{d}_B} \quad & f(\left[\begin{smallmatrix}\boldsymbol{\alpha}_B \\ \boldsymbol{\alpha}_N\end{smallmatrix}\right] + \left[\begin{smallmatrix}\boldsymbol{d}_B \\ \boldsymbol{0}\end{smallmatrix}\right]) \\ \text{subject to} \quad & -\alpha_i \leq d_i \leq C - \alpha_i, \forall i \in B \\ & d_i = 0, \forall i \notin B, \end{aligned} \tag{14}$$

where $N = \{1, \ldots, l\} \setminus B$ and

$$f(\left[\begin{smallmatrix}\boldsymbol{\alpha}_B \\ \boldsymbol{\alpha}_N\end{smallmatrix}\right] + \left[\begin{smallmatrix}\boldsymbol{d}_B \\ \boldsymbol{0}\end{smallmatrix}\right]) = \frac{1}{2}\boldsymbol{d}_B^T \bar{Q}_{BB} \boldsymbol{d}_B + \nabla_B f(\boldsymbol{\alpha})^T \boldsymbol{d}_B + \text{constant}.$$

Note that $\bar{Q}_{BB}$ is a sub-matrix of the matrix $\bar{Q}$. If $|B| = 1$, (14) is reduced to the single-variable sub-problem in (3). We present a parallel implementation of the BSVM algorithm in Algorithm 6, which is the same as the current BSVM implementation except the parallel calculation of $\nabla f(\boldsymbol{\alpha})$ at line 3. Clearly, Algorithm 6 is a special case of the framework in Algorithm 5 with $\bar{B} = \{1, \ldots, l\}$.

While both Algorithms 4 and 6 are realizations of Algorithm 5, they significantly differ in how to update $\boldsymbol{\alpha}_B$ after selecting the working set B (line 6 of Algorithm 5). In [9], the sub-problem (14) is accurately solved by an optimization algorithm that costs

$$O(|B|^2 n + |B|^3)$$

[4]Maintaining $\boldsymbol{w}$ is not possible in the kernel case because it is too high dimensional to be stored.

Algorithm 6 A parallel implementation of the BSVM algorithm [7] for linear classification

1: Specify a feasible $\boldsymbol{\alpha}$ and calculate $\boldsymbol{w} = \sum_j y_j \alpha_j \boldsymbol{x}_j$
2: **while** true **do**
3: Calculate $\nabla_i f(\boldsymbol{\alpha}), \forall i = 1, \ldots, l$ in parallel
4: **if** $\boldsymbol{\alpha}$ is close to an optimum **then**
5: break
6: Select B by the procedure in Section 3.3
7: Find $\boldsymbol{d}_B$ by solving (14)
8: **for** $i \in B$ **do**
9: **if** $|d_i| \geq \bar{\varepsilon}$ **then**
10: $\alpha_i \leftarrow \alpha_i + d_i$
11: $\boldsymbol{w} \leftarrow \boldsymbol{w} + d_i y_i \boldsymbol{x}_i$

operations, where $|B|^2 n$ is for constructing the matrix $\bar{Q}_{BB}$ and $|B|^3$ is for factorizing $\bar{Q}_{BB}$ several times. In contrast, from line 11 to line 17 in Algorithm 4, we very loosely solve the sub-problem (3) by conducting $|B|$ number of CD updates. As a result, we can see the following difference on the two algorithms' complexity:

$$\begin{array}{l} \text{Algorithm 4: } O(\frac{ln}{PT} + |B|n) \times \#\text{inner iterations}, \\ \text{Algorithm 6: } O(\frac{ln}{P} + \frac{(|B|^2 n + |B|^3)}{P}) \times \#\text{iterations}. \end{array} \quad (15)$$

Here an inner iteration in Algorithm 4 means to handle one block $\bar{B}$ of $\bar{B}_1, \ldots, \bar{B}_T$. We let it be compared to an iteration in Algorithm 6 because they both update elements in a set B eventually. Note that in (15) we slightly favor Algorithm 6 by assuming that solving the sub-problem (14) can be fully parallelized.

The complexity comparison in (15) explains why in the serial setting the cyclic CD [5] is much more widely used than the BSVM implementation [7]. When a single thread is used, Algorithm 4 is reduced to Algorithm 1 with $P = 1, T = l$ and $|B| = 1$. Then (15) becomes

$$\begin{array}{c} \text{Algorithm 1: } O(n+n) \times \#\text{inner iterations}, \\ \text{Algorithm 6 (serial): } O(ln + (|B|^2 n + |B|^3)) \times \#\text{iterations}. \end{array}$$

Clearly the ln term causes each iteration of Algorithm 6 to be extremely expensive. Thus unless the number of iterations is significantly less, the total time of Algorithm 6 is more than that of Algorithm 1. Now for the multi-core environment, Algorithm 4 parallelizes the evaluation of $|\bar{B}| = l/T$ gradient components, and to use the latest gradient information, $|\bar{B}|$ cannot be too large (we used several hundreds or thousands in our experiments; see Section 4 for details.). In contrast, Algorithm 6 parallelizes the evaluation of all l components. Because the scalability is often better for the situation of a higher computational demand, we expect that Algorithm 6 benefits more from multi-core computation. Therefore, it is interesting to see if Algorithm 6 becomes practically viable. Unfortunately, in Section 5.2 we see that even with better scalability, Algorithm 6 is still slower than Algorithm 4.

4. THEORETICAL PROPERTIES AND IMPLEMENTATION ISSUES

In this section we investigate theoretical properties and implementation issues of Algorithm 4, which will be used for subsequent comparisons with existing approaches. First we show the finite termination.

Theorem 1 *Under any given stopping tolerance $\varepsilon > 0$, Algorithm 4 terminates after a finite number of iterations.*

Because of the space limitation, we leave the proof in Section I of supplementary materials.

Besides the finite termination under a tolerance ε, we hope that as $\varepsilon \to 0$, the resulting solution can approach an optimum. Then the asymptotic convergence is established. Note that Algorithm 4 has another parameter $\bar{\varepsilon} \ll \varepsilon$, so we also need $\bar{\varepsilon} \to 0$ as well. Now assume that $\boldsymbol{\alpha}^{\varepsilon,\bar{\varepsilon}}$ is the solution after running Algorithm 4 under ε and $\bar{\varepsilon}$, and

$$\boldsymbol{w}^{\varepsilon,\bar{\varepsilon}} = \sum y_j \alpha_j^{\varepsilon,\bar{\varepsilon}} \boldsymbol{x}_j.$$

The following theorem gives the asymptotic convergence.

Theorem 2 *Consider a sequence $\{\varepsilon_k, \bar{\varepsilon}_k\}$ with*

$$\lim_{k\to\infty} \varepsilon_k, \bar{\varepsilon}_k = 0, 0. \quad (16)$$

If $\boldsymbol{w}^*$ is the optimum of (1), then we have

$$\lim_{k\to\infty} \boldsymbol{w}^{\varepsilon_k,\bar{\varepsilon}_k} = \boldsymbol{w}^*.$$

Next we discuss several implementation issues.

4.1 Shrinking

An effective technique demonstrated in [5] to improve the efficiency of dual CD methods is shrinking. This technique, originated from training kernel classifiers, aims to remove some elements that are likely bounded (i.e., $\alpha_i = 0$ or U) in the end. For the proposed Algorithm 4, the shrinking technique can be easily adapted. Once $\nabla_{\bar{B}} f(\boldsymbol{\alpha})$ is calculated, we can apply conditions used in [5] to remove some elements in $\bar{B}$. After the stopping condition is satisfied on the remaining elements, we check if the whole set satisfies the same condition as well. A detailed pseudo code is given in Algorithm I of supplementary materials.

4.2 The Size of $|\bar{B}|$

In Algorithm 4, the set $\bar{B}$ is important because we parallelize the calculation of $\nabla_{\bar{B}} f(\boldsymbol{\alpha})$ and then select a set $B \subset \bar{B}$ for CD updates. Currently we cyclically get $\bar{B}$ after splitting $\{1, \ldots, l\}$ to T blocks, but the size of $\bar{B}$ needs to be decided. We list the following considerations.

- $|\bar{B}|$ cannot be too small because first the overhead in parallelizing the calculation of $\nabla_{\bar{B}} f(\boldsymbol{\alpha})$ becomes significant, and second the set B selected from $\bar{B}$ may be empty.
- $|\bar{B}|$ cannot be too large because the algorithm uses the current solution to select too many elements at a time for CD updates. Without using the latest gradient information, the convergence may be slower.

Fortunately, we find that the training time is about the same when $|\bar{B}|$ is set to be a few hundreds or a few thousands. Therefore, the selection of $|\bar{B}|$ is not too difficult; see experimental results in Section 5.1.2.

To avoid that $|B| = 0$ happens frequently, we further design a simple rule to adjust the size of $|\bar{B}|$:

if $|B| = 0$ **then**
$|\bar{B}| \leftarrow \min(|\bar{B}| \times 1.5, \text{maxB})$
else if $|B| \geq \text{init}\bar{\text{B}}$ **then**
$|\bar{B}| \leftarrow |\bar{B}|/2$

The idea is to check the size of B for deciding if $|\bar{B}|$ needs to be adjusted: If $|B| = 0$, to get some elements in B for

Table 1: Data statistics: Density is the average ratio of non-zero features per instance. Ratio is the percentage of running time spent on the gradient calculation (line 8 of Algorithm 4); we consider the $l1$ loss by using one core (see also the discussion in Section 5.1).

Data set	#data	#features	density	ratio
rcv1	677,399	47,236	0.15%	89%
yahoo-korea	368,444	3,052,939	0.01%	86%
yahoo-japan	176,203	832,026	0.02%	96%
webspam (trigram)	350,000	16,609,143	0.02%	91%
url_combined	2,396,130	3,231,961	0.004%	86%
KDD2010-b	19,264,097	29,890,095	0.0001%	86%
covtype	581,012	54	22.12%	66%
epsilon	400,000	2,000	100%	80%
HIGGS	11,000,000	28	92.11%	85%

CD updates, we should enlarge $\bar{B}$. In contrast, if too many elements are included in B, we should reduce the size of $\bar{B}$. Here init$\bar{\text{B}}$ is the initial size of $\bar{B}$, while max$\bar{\text{B}}$ is the upper bound. In our experiments, we set init$\bar{\text{B}} = 256$ and max$\bar{\text{B}} = 4,096$. Because in general $0 < |B| < \text{init}\bar{\text{B}}$, $|\bar{B}|$ is seldom changed in practice. Hence our rule mainly serves as a safeguard.

4.3 Adaptive Condition in Choosing B

Algorithm 4 is ε-dependent because of the condition

$$|\nabla_i^P f(\boldsymbol{\alpha})| \geq \delta\varepsilon$$

to select the set B. This property is undesired because if users pick a very small ε, then in the beginning of the algorithm almost all elements in $\bar{B}$ are included in B. To make Algorithm 4 independent of the stopping tolerance, we have a separate parameter ε_1, that starts with a constant not too close to zero and gradually decreases to zero. Specifically we make the following changes:

1. $\varepsilon_1 = 0.1$ in the beginning.
2. The set B is selected by

$$B \leftarrow \{i \mid i \in \bar{B}, |\nabla_i^P f(\boldsymbol{\alpha})| \geq \delta\varepsilon_1\}.$$

3. The stopping condition is changed to

if $M < \varepsilon_1$ or $\bar{t} = 0$ **then**
  if $\varepsilon_1 \leq \varepsilon$ **then**
    break
  else
    $\varepsilon_1 \leftarrow \max(\varepsilon, \varepsilon_1/10)$

Therefore, the algorithm relies on a fixed sequence of ε_1 values rather than a single value ε specified by users.

5. EXPERIMENTS

We consider nine data sets, each of which has a large number of instances.[5] Six of them are sparse sets with many features, while the others are dense sets with few features. Details are in Table 1.

In all experiments, the regularization parameter $C = 1$ is used. We have also considered the best C value selected by cross-validation. Results, presented in supplementary materials, are similar. All implementations, including the one in [5], are extended from the package LIBLINEAR version 2.1 [2] by using OpenMP [1], so the comparison is fair. The initial $\boldsymbol{\alpha} = \mathbf{0}$ is used in all algorithms. In Algorithm 4, we set $\bar{\varepsilon} = 10^{-15}, \delta = 0.1$, and the initial $|\bar{B}| = 256$. Experiments are conducted on Amazon EC2 m4.4xlarge machines, each of which is equivalent to 8 cores of an Intel Xeon E5-2676 v3 CPU.

5.1 Analysis of Algorithm 4

We investigate various aspects of Algorithm 4. Some more results are in supplementary materials.

5.1.1 Percentage of Parallelizable Operations

Our idea in Algorithm 4 is to make the gradient calculation the most computationally expensive yet parallelizable step of the procedure. In Table 1, we check the percentage of total training time spent on this operation by using a single core and the stopping tolerance $\varepsilon = 0.1$.[6] The $l1$ loss is considered. Results indicate that in general more than 80% of time is used for calculating the gradient. Hence the running time can be effectively reduced in a multi-core environment.

5.1.2 Size of the Set $\bar{B}$

An important parameter to be decided in Algorithm 4 is the size of the set $\bar{B}$; see the discussion in Section 4.2. To see how the set size $|\bar{B}|$ affects the running time, in Figures I and II of supplementary materials, we compare the running time of using $|\bar{B}| = 64, 256, 1024$. Note that we do not apply the adaptive rule in Section 4.2 in order to see the effect of different $|\bar{B}|$ sizes.

Results show that $|\bar{B}| = 64$ is slightly worse than 256 and 1,024. For a too small $|\bar{B}|$, the parallelization of $\nabla_{\bar{B}} f(\boldsymbol{\alpha})$ is less effective because the overhead to conduct parallel operations becomes significant. On the other hand, results of using $|\bar{B}| = 256$ and 1,024 are rather similar, so the selection of $|\bar{B}|$ is not difficult in practice.

5.2 Comparison of Algorithms 4 and 6

We briefly compare Algorithms 4 and 6 because they are two different realizations of the framework in Algorithm 5. We mentioned in Section 3.3 that Algorithm 6 use all gradient elements to greedily select a subset B, but Algorithm 4 is closer to the cyclic CD.

In Table 2, we compare them by using two sets. The subset size $|B| = 10$ is considered in Algorithm 6, while for Algorithm 4, the initial $|B| = 256$ is used for the adaptive rule in Section 4.2. A stopping tolerance $\varepsilon = 0.001$ is used for both algorithms, although in all cases Algorithm 4 reaches a smaller final objective value. Clearly, Table 2 indicates that increasing the number of cores from 1 to 8 leads to more significant improvement on Algorithm 6. However, the overall computational time is still much more than that of Algorithm 4. This result is consistent with our analysis in Section 3.3. Because Algorithm 4 is superior, subsequently we use it for other experiments.

5.3 Comparison of Parallel Dual CD Methods

We compare the following approaches.

- Mini-batch CD [13]: See Section 2.2.1 for details.
- Asynchronous CD [6]: We directly use the implementation in [6]. See details in Section 2.2.2.

[5] All sets except yahoo-japan and yahoo-korea are available at http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/. For covtype, both scaled and original versions are available; we use the scaled one.

[6] The value 0.1 is the default stopping tolerance used in LIBLINEAR.

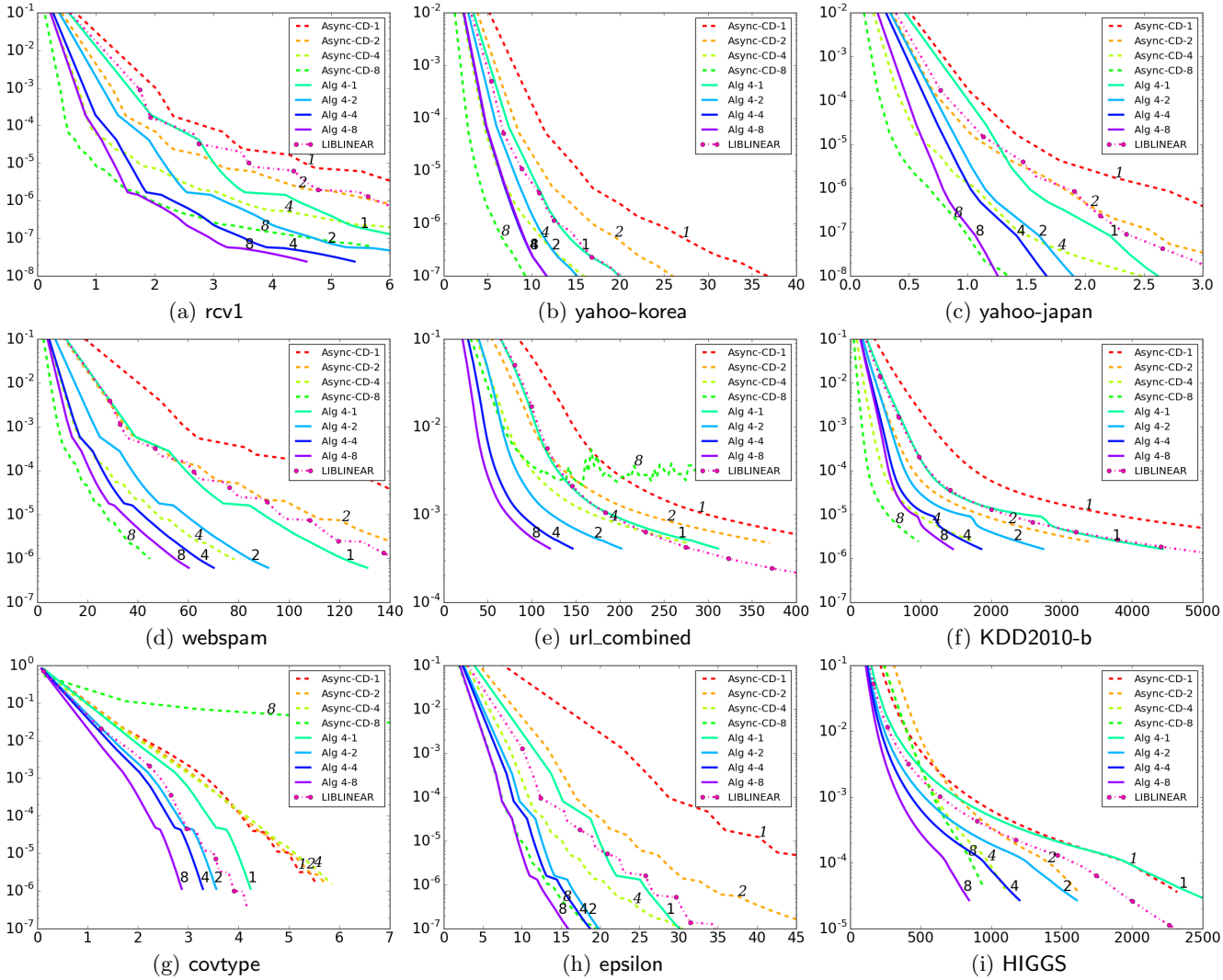


Figure 1: A comparison of two multi-core dual CD methods: asynchronous CD and Algorithm 4, and one single-core implementation: LIBLINEAR. We present the relation between training time in seconds (x-axis) and the relative difference to the optimal objective value (y-axis, log-scaled). The $l1$ loss is used.

Table 2: A comparison between Algorithms 4 and 6 on the training time (in seconds). A stopping tolerance $\varepsilon = 0.001$ is used.

	Algorithm 4		Algorithm 6	
Data	1 core	8 cores	1 core	8 cores
covtype	28.6	13.7	3,624.8	1,251.5
rcv1	12.0	4.4	2,114.8	406.8

- Algorithm 4: the proposed multi-core dual CD algorithm in this study.
- LIBLINEAR [2]: It implements Algorithm 1 with the shrinking technique [5]. This serial code serves as a reference to compare with the above multi-core algorithms.

To see how the algorithm behaves as training time increases, we carefully consider non-stop settings for these approaches; see details in Section VII of supplementary materials.

We check the relation between running time and the relative difference to the optimal objective value:

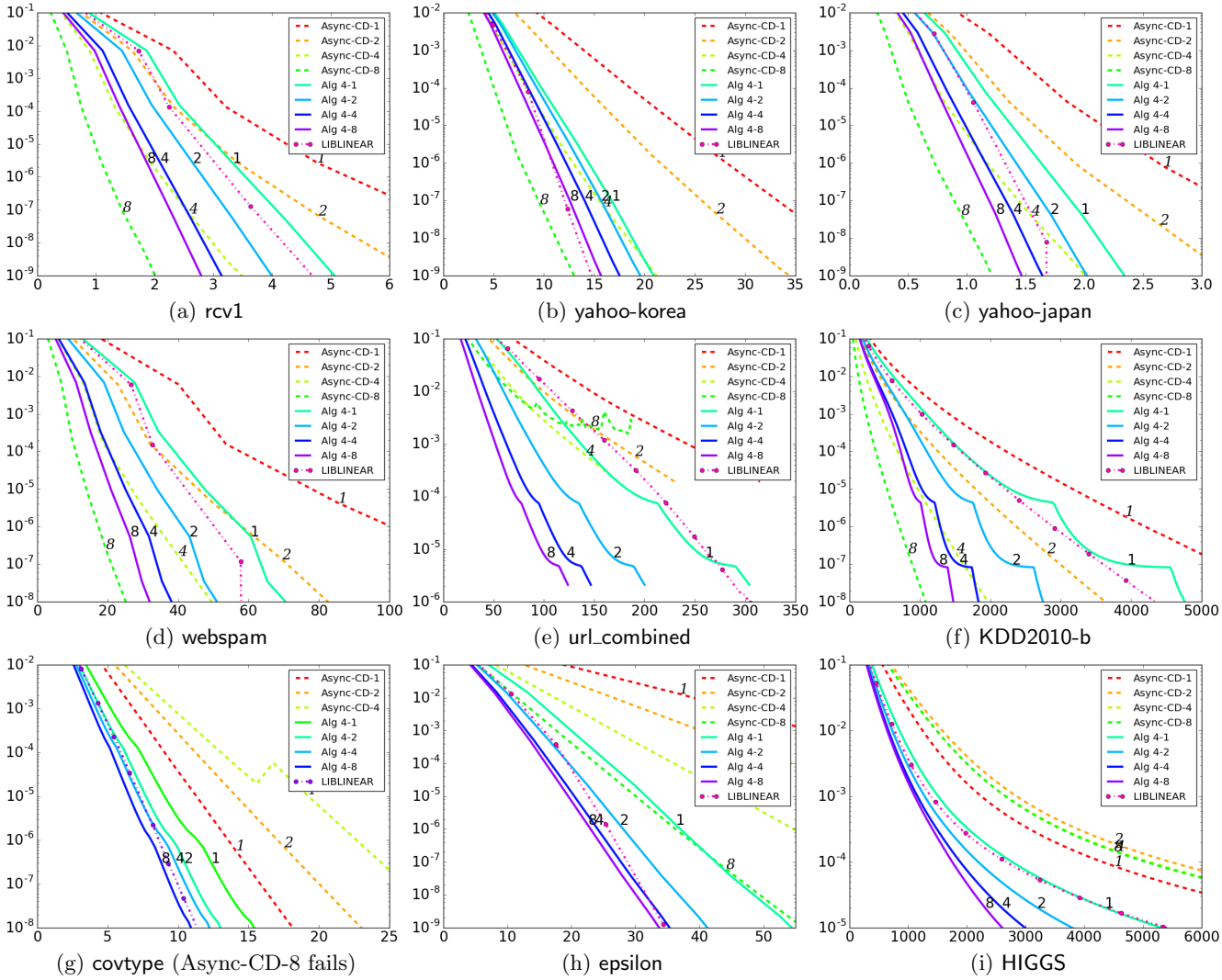
$$|f(\boldsymbol{\alpha}) - f(\boldsymbol{\alpha}^*)|/|f(\boldsymbol{\alpha}^*)|,$$

where $f(\boldsymbol{\alpha})$ is the objective function of (2). Because $\boldsymbol{\alpha}^*$ is not available, we obtain an approximate optimal $f(\boldsymbol{\alpha}^*)$ by running LIBLINEAR with a small tolerance $\varepsilon = 10^{-6}$.

Before presenting the main comparisons, by some experiments we rule out mini-batch CD because it is less efficient in compared with other methods. Details are left in Supplementary Section III.

We present the main results of using $l1$ and $l2$ losses in Figures 1 and 2, respectively. To check the scalability, 1, 2, 4, 8 cores are used. Note that our CPU has 8 cores and all three approaches apply the shrinking technique. Therefore, the result of asynchronous CD may be slightly different from that in [6], where shrinking is not applied. From Figures 1 and 2, the following observations can be made.

- For some sparse problems, asynchronous CD gives excellent speedup as the number of cores increases. However, it fails to converge in some situations (url_combined and covtype when 8 cores are used). For all three dense problems with $l1$ or $l2$ loss, it diverges if 16 cores are used.



(a) rcv1 (b) yahoo-korea (c) yahoo-japan

(d) webspam (e) url_combined (f) KDD2010-b

(g) covtype (Async-CD-8 fails) (h) epsilon (i) HIGGS

Figure 2: The same comparison as in Figure 1 except that the $l2$ loss is used.

- Algorithm 4 is robust because it always converges. Although the scalability may not be as good as asynchronous CD in the beginning, in Figure 1 it generally has faster final convergence. Further, Algorithm 4 achieves much better speedup for problems url_combined and covtype, where asynchronous CD may diverge.

- In compared with the serial algorithm in LIBLINEAR, we can see that Algorithm 4 using one core is slower. However, as the number of cores increases, Algorithm 4 often becomes much faster. This observation confirms the importance of modifying the serial algorithm to take the advantage of multi-core computation, where our discussion in Section 3 serves as a good illustration.

- Results for $l1$ and $l2$ losses are generally similar though we can see that for all approaches, the final convergence for the $l2$ loss is nicer. The curves of training time versus the objective value are sometimes close to straight lines.

- The resulting curve of Algorithm 4 may look like a piecewise combination of several curves. This situation comes from the reduction of the ε_1 parameter; see Section 4.3.

We have also compared these methods without applying the shrinking technique. Detailed results are in Section V of supplementary materials.

It is mentioned in Section 2.2.2 that to address the convergence issue of the asynchronous CD method, the study in [14] considers a semi-asynchronous setting. We modify the code in [6] to have that $\boldsymbol{w}$ is recalculated by (6) after each cycle of using all $\boldsymbol{x}_i, \forall i = 1, \ldots, l$. The computational time is significantly increased, but we observe similar behavior. For problems where the asynchronous CD method fails, so does the new semi-asynchronous implementation. Therefore, it is unclear to us yet how to effectively modify the asynchronous CD method so that the convergence is guaranteed.

6. DISCUSSION AND CONCLUSIONS

Before making conclusions we discuss issues including limitation and future challenges of the proposed approach.

6.1 Multi-CPU Environments and the Comparison with Parallel Newton Methods

Our current development is for the environment of a single CPU with multiple cores. We find that if multiple CPUs

are used (i.e., the NUMA architecture in multi-processing), then the scalability is slightly worse. The main reason is because of the communication between CPUs. Assume two CPUs are available: CPU-1 and CPU-2. When $\nabla_{\bar{B}} f(\boldsymbol{\alpha})$ is calculated in parallel (line 8 of Algorithm 4), an instance $\boldsymbol{x}_i$ may be loaded into the cache of CPU-2 for calculating $\nabla_i f(\boldsymbol{\alpha})$. Later if $\boldsymbol{x}_i$ is selected to the set B and CPU-1 is utilized to sequentially conduct CD updates on elements in B (line 11 of Algorithm 4), then $\boldsymbol{x}_i$ must be loaded from memory or transferred from the cache of CPU-2. How to design an effective parallel dual CD method for multi-CPU environments is an important future issue.

Our recent study [11] on parallel Newton methods for the primal problem with $l2$ and LR losses easily achieves excellent speedup in multi-CPU environments. In compared with Algorithm 4, a Newton method possesses the following advantages for parallelization.

- For every operation the Newton method uses the whole data set, so it is like that the set $\bar{B}$ in Algorithm 4 becomes much bigger. Then the overhead for parallelization is relatively smaller.
- In the previous paragraph we discussed that in a loop of Algorithm 4 an $\boldsymbol{x}_i$ may be accessed in two separate places. Such situations do not occur in the Newton method, so the issue of memory access or data movement between CPUs is less serious.

Nevertheless, parallel dual CD is still very useful because of the following reasons. First, in the serial setting, dual CD is in some cases much faster than other approaches including the primal Newton method, so even with less effective parallelization, it may still be faster. Second, for the $l1$ loss, the primal problem lacks differentiability, so solving the differentiable dual problem is more suitable. Because the dual problem possesses bound constraints $0 \le \alpha_i \le U, \forall i$, unconstrained optimization methods such as Newton or quasi-Newton cannot be directly applied. In contrast, CD methods are convenient choices for such problems.

6.2 Using $\nabla^P f(\boldsymbol{\alpha})$ or $\boldsymbol{\alpha} - P[\boldsymbol{\alpha} - \nabla f(\boldsymbol{\alpha})]$

It is known that both

$$\nabla^P f(\boldsymbol{\alpha}) = \mathbf{0} \text{ and } \boldsymbol{\alpha} - P[\boldsymbol{\alpha} - \nabla f(\boldsymbol{\alpha})] = \mathbf{0}$$

are optimality conditions of problem (2). Here $P[\cdot]$ is the projection operation defined as

$$P[\alpha_i] = \min(\max(\alpha_i, 0), U).$$

These two conditions are respectively used in lines 9-10 and lines 13-14 of Algorithm 4. An interesting question is why we do not just use one of the two.

In optimization, $\boldsymbol{\alpha} - P[\boldsymbol{\alpha} - \nabla f(\boldsymbol{\alpha})]$ is often considered more suitable because it gives a better measure about the optimality when α_i is close to a bound. For example,

$$\alpha_i = 10^{-5} \text{ and } \nabla_i f(\boldsymbol{\alpha}) = 5 \text{ imply that}$$
$$\nabla_i^P f(\boldsymbol{\alpha}) = 5 \text{ and } \alpha_i - P[\alpha_i - \nabla_i f(\boldsymbol{\alpha})] = 10^{-5}.$$

Clearly α_i cannot be moved much, so $\alpha_i - P[\alpha_i - \nabla_i f(\boldsymbol{\alpha})]$ rightly indicates this fact. Therefore, it seems that we should use $\alpha_i - P[\alpha_i - \nabla_i f(\boldsymbol{\alpha})]$ in lines 9-10 instead. We still use project gradient mainly because of historical reasons. The dual CD in LIBLINEAR currently relies on project gradient for implementing the shrinking technique and so does the asynchronous CD [6] used for comparison. Hence we follow them for a fair comparison. Modifying Algorithm 4 to use $\alpha_i - P[\alpha_i - \nabla_i f(\boldsymbol{\alpha})]$ is worth investigating in the future.

6.3 Conclusions

In this work we have proposed a general framework for parallel dual CD. For one specific implementation we establish the convergence properties and demonstrate the effectiveness in multi-core environments.

Acknowledgements This work was supported in part by MOST of Taiwan via the grant 104-2221-E-002-047-MY3.

References

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

[2] 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.

[3] R.-E. Fan, P.-H. Chen, and C.-J. Lin. Working set selection using second order information for training SVM. *JMLR*, 6:1889–1918, 2005.

[4] C. Hildreth. A quadratic programming procedure. *Naval Res. Logist.*, 4:79–85, 1957.

[5] 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.

[6] C.-J. Hsieh, H.-F. Yu, and I. S. Dhillon. PASSCoDe: Parallel asynchronous stochastic dual coordinate descent. In *ICML*, 2015.

[7] C.-W. Hsu and C.-J. Lin. A simple decomposition method for support vector machines. *Machine Learning*, 46:291–314, 2002.

[8] T. Joachims. Making large-scale SVM learning practical. In *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1998.

[9] W.-C. Kao, K.-M. Chung, C.-L. Sun, and C.-J. Lin. Decomposition methods for linear support vector machines. *Neural Comput.*, 16(8):1689–1704, 2004.

[10] C.-P. Lee and D. Roth. Distributed box-constrained quadratic optimization for dual linear SVM. In *ICML*, 2015.

[11] 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.

[12] J. C. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods - Support Vector Learning*, Cambridge, MA, 1998. MIT Press.

[13] M. Takáč, P. Richtárik, and N. Srebro. Distributed mini-batch SDCA, 2015. arXiv.

[14] K. Tran, S. Hosseini, L. Xiao, T. Finley, and M. Bilenko. Scaling up stochastic dual coordinate ascent. In *KDD*, 2015.

[15] T. Yang. Trading computation for communication: Distributed stochastic dual coordinate ascent. In *NIPS*. 2013.