Fast Matrix-vector Multiplications for Large-scale Logistic Regression on Shared-memory Systems

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

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

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

***Abstract*—Shared-memory systems such as regular desktops now possess enough memory to store large data. However, the training process for data classification can still be slow if we do not fully utilize the power of multi-core CPUs. Many existing works proposed parallel machine learning algorithms by modifying serial ones, but convergence analysis may be complicated. Instead, we do not modify machine learning algorithms, but consider those that can take the advantage of parallel matrix operations. We particularly investigate the use of parallel sparse matrix-vector multiplications in a Newton method for large-scale logistic regression. Various implementations from easy to sophisticated ones are analyzed and compared. Results indicate that under suitable settings excellent speedup can be achieved.**

***Keywords*-sparse matrix; parallel matrix-vector multiplication; classification; Newton method**

I. Introduction

Classification algorithms are now widely used in many machine learning and data mining applications, but training large-scale data remains a time-consuming process. To reduce the training time, many parallel classification algorithms have been proposed for shared-memory or distributed systems. In this work, we target at shared-memory systems because nowadays a typical server possesses enough memory to store reasonably large data and multi-core CPUs are widely used.

Existing parallel machine learning algorithms for classification include, for example, [1], [2], [3], [4], [5], [6]. Some target at distributed systems, but some others are for shared-memory systems. These algorithms are often designed by extending existing machine learning algorithms to parallel settings. For example, stochastic gradient (SG) and coordinate descent (CD) methods are efficient methods for data classification [7], [8], but they process one data instance at a time. The algorithm is inherently sequential because the next iteration relies on the result of the current one. For parallelization, researchers have proposed modifications so that each node or thread independently conducts SG or CD iterations on a subset of data. To avoid synchronization between threads, the machine learning model may be updated by one thread without considering the progress in others [1], [9]. An issue of the existing works is that convergence must be proved and analyzed following the modification from the serial setting.

Instead of modifying algorithms to achieve parallelism, it may be simpler to consider machine learning algorithms that can take the advantage of fast and parallel linear algebra modules. The core computational tasks of some popular machine learning algorithms are sparse matrix operations. For example, if Newton methods are used to train logistic regression for large-scale document data, the data matrix is sparse and sparse matrix-vector multiplications are the main computational bottleneck [10], [11]. If we can apply fast implementations of matrix operations, then machine learning algorithms can be parallelized without any modification. Such an approach possesses the following advantages.

1) Because the machine learning algorithm is not modified, the convergence analysis still holds.
2) The training speed keeps improving along with the advent on fast matrix operations.
3) By considering the whole data matrix rather than some instances, it is easier to improve the memory access. In shared-memory systems, moving data from lower-level memory (e.g., main memory) to upper level (e.g., cache) may be a more serious bottleneck than the synchronization or communication between threads.

Although our goal is to employ fast sparse matrix operations for machine learning algorithms, we must admit that unlike dense matrix operations, the development has not been as successful. It is known that sparse matrix operations may be memory bound [12], [13], [14]. That is, CPU waits for data being read from lower-level to upper-level memory. The same situation has appeared for dense matrix operations, but optimized implementations have been well studied by the numerical analysis community. The result is the successful optimized BLAS in the past several decades. BLAS (Basic Linear Algebra Subprograms) is a standard set of functions defined for vector and dense matrix operations [15], [16], [17]. By carefully designing the algorithm to reduce cache/memory accesses, an optimized matrix-matrix or matrix-vector multiplication can be much faster than a naive implementation. However, because of the complication on the quantity and positions of non-zero values, such huge success has not applied to sparse matrices. Fortunately, intensive research has been conducted recently so some tools for fast sparse matrix-vector multiplications are available (e.g., [12], [13]).

With the recent progress on the fast implementation of matrix-vector multiplications, it is time to investigate if applying them to some machine learning algorithms can reduce the running time. In this paper, we focus on Newton methods for large-scale logistic regression. In past works [10], [11] and the widely used software LIBLINEAR [18] for linear classification, it is known that the main computational bottleneck in a Newton method is matrix-vector multiplications. We investigate various implementations of matrix-vector multiplications when they are applied to the Newton method. Experiments show that we can achieve significant speedup on a typical server. In particular, a carefully designed implementation using OpenMP [19] is simple and efficient.

In Section II, we point out that the computational bottleneck of Newton methods for logistic regression is on matrix-vector multiplications. We then investigate several state-of-the-art strategies for sparse matrix-vector multiplications. Thorough experiments are in Section III to illustrate how we achieve good speedup. An extension of LIBLINEAR based on this work is available at http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/multicore-liblinear. Supplementary materials including the connection to [1] can be found at the same page.

II. Sparse Matrix-vector Multiplications in Newton Methods for Logistic Regression

Logistic regression is widely used for classification. Given training instances $\{(y_i, \boldsymbol{x}_i)\}_{i=1}^{l}$ with label $y_i = \pm 1$ and feature vector $\boldsymbol{x}_i \in R^n$, logistic regression solves the following convex optimization problem to obtain a model vector $\boldsymbol{w}$.

$$\min_{\boldsymbol{w}} \quad f(\boldsymbol{w}) \equiv \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum\nolimits_{i=1}^{l} \log(1 + e^{-y_i \boldsymbol{w}^T \boldsymbol{x}_i}), \quad (1)$$

where $C > 0$ is the regularization parameter, l is the number of instances, n is the number of features. Many optimization methods have been proposed for training large-scale logistic regression. Among them, Newton methods are reliable approaches [20], [10], [11]. Assume the data matrix is

$$X = \begin{bmatrix}\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l\end{bmatrix}^T \in R^{l \times n}.$$

It is known that at each Newton iteration the main computational task is on a sequence of Hessian-vector products.

$$\nabla^2 f(\boldsymbol{w})\boldsymbol{d} = (\mathcal{I} + CX^T DX)\boldsymbol{d}, \quad (2)$$

where $\nabla^2 f(\boldsymbol{w})$ is the Hessian, $\mathcal{I}$ is the identity matrix, D is a diagonal matrix associated with the current $\boldsymbol{w}$, and $\boldsymbol{d}$ is an intermediate vector. Because of space limit, we leave detailed derivations in supplementary materials. Our focus in this paper is thus on how to speedup the calculation of (2).

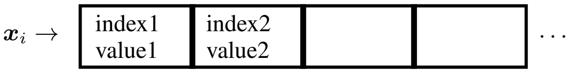
From (2), the Hessian-vector multiplication involves

$$X\boldsymbol{d}, \quad D(X\boldsymbol{d}), \quad X^T(DX\boldsymbol{d}).$$

The second step takes only $O(l)$ because D is diagonal. Subsequently we investigate efficient methods for the first and the third steps.

A. Baseline: Single-core Implementation in LIBLINEAR

By the nature of data classification, currently LIBLINEAR implements an instance-based storage so that non-zero entries in each $\boldsymbol{x}_i$ can be easily accessed. Specifically, $\boldsymbol{x}_i$ is an array of nodes and each node contains feature index and value:



Note that we modify a figure in [21] here. This type of row-based storage is common for sparse matrices, where the most used one is the CSR (Compressed Sparse Row) format [22]. However, we note that the CSR format separately saves feature indices (i.e., column indices of X) and feature values in two arrays, so it is slightly different from the above implementation in LIBLINEAR.

For the matrix-vector multiplication,

$$\boldsymbol{u} = X\boldsymbol{d} \qquad ,$$

LIBLINEAR implements the following simple loop

```
for $i = 1, \ldots, l$ do
    $u_i = \boldsymbol{x}_i^T \boldsymbol{d}$
```

For the other matrix-vector multiplication

$$\bar{\boldsymbol{u}} = X^T\boldsymbol{u}, \text{ where } \boldsymbol{u} = DX\boldsymbol{d},$$

we can use the following loop

```
for $i = 1, \ldots, l$ do
    $\bar{\boldsymbol{u}} \leftarrow \bar{\boldsymbol{u}} + u_i \boldsymbol{x}_i$
```

because $\bar{\boldsymbol{u}} = u_1\boldsymbol{x}_1 + \cdots + u_l\boldsymbol{x}_l$. By this setting there is no need to calculate and store X^T.

B. Parallel Matrix-vector Multiplication by OpenMP

We aim at parallelizing the two loops in Section II-A by OpenMP [19], which is simple and common in shared-memory systems. For the first loop, because $\boldsymbol{x}_i^T\boldsymbol{d}, \forall i$ are independent, we can easily parallelize it by

```
for $i = 1, \ldots, l$ do in parallel
    $u_i = \boldsymbol{x}_i^T \boldsymbol{d}$
```

It is known [12] that parallelizing the second loop is more difficult because threads cannot update $\bar{\boldsymbol{u}}$ at the same time. A common solution is that, after $u_i(x_i)_s$ is calculated, we update $\bar{u}_s$ by an atomic operation that avoids other threads to write $\bar{u}_s$ at the same time. Specifically, we have the following loop.

```
for $i = 1, \ldots, l$ do in parallel
    for $(x_i)_s \neq 0$ do
        atomic: $\bar{u}_s \leftarrow \bar{u}_s + u_i(x_i)_s$
```

This loop of calculating $X^T\boldsymbol{u}$ by atomic operations is related to the recent asynchronous parallel coordinate descent methods for linear classification [1]. In supplementary materials, we discuss the relationship between our results and theirs.

An alternative approach to calculate $X^T\boldsymbol{u}$ without using atomic operations is to store

$$\hat{\boldsymbol{u}}^p = \sum\{u_i\boldsymbol{x}_i \mid i \text{ run by thread } p\}$$

during the computation, and sum up these vectors in the end. This approach essentially simulates a reduction operation in parallel computation: each thread has a local copy of the results and these local copies are combined for the final result. Currently, OpenMP supports reduction for scalars rather than arrays for C/C++, so we must handle these local arrays by ourself. The extra storage is in general not a concern because if the data set is not extremely sparse and the number of cores is not super high, then the size of $\hat{\boldsymbol{u}}^p, \forall p$ is relatively smaller than the data size. Similarly, the extra cost for summing $\hat{\boldsymbol{u}}^p$, $\forall p$ should be minor, but we will show in Section III-D that appropriate settings are needed to ensure fast computation.

TABLE I

LEFT: DATA STATISTICS. DENSITY IS THE AVERAGE RATIO OF NON-ZERO FEATURES PER INSTANCE. RIGHT: NUMBER OF ITERATIONS AND PERCENTAGE OF RUNNING TIME SPENT ON MATRIX-VECTOR MULTIPLICATIONS IN THE NEWTON METHOD. WE USE THE BASELINE LIBLINEAR WITH ONE THREAD.

Data set	#instances	#features	#nonzeros	density	#class	C	# iters	# and ratio of matrix-vector multiplications	
KDD2010-b	19,264,097	29,890,095	566,345,668	0.000%	2	0.0625	77	1,221	3,902.65 / 4,752.93 = 82.11%
url_combined	2,396,130	3,231,961	277,058,644	0.004%	2	1	8	44	87.82 / 92.61 = 94.83%
webspam	350,000	16,609,143	1,304,697,446	0.022%	2	1	6	50	530.66 / 541.80 = 97.95%
rcv1_binary	677,399	47,236	49,556,258	0.155%	2	1	5	43	14.70 / 15.02 = 97.88%
covtype_binary	581,012	54	6,940,438	22.12%	2	1	4	60	1.80 / 2.02 = 89.20%
epsilon	400,000	2,000	800,000,000	100.00%	2	1	4	46	124.70 / 124.85 = 99.88%
rcv1_multiclass	518,571	47,236	33,486,015	0.137%	53	1	451	1,696	563.65 / 580.85 = 97.04%
covtype_multiclass	581,012	54	6,972,144	22.22%	7	1	51	630	19.62 / 22.02 = 89.06%

C. Combining Matrix-vector Operations

Although in Section II-B we have two separate loops for $X\boldsymbol{d}$ and $X^T(DX\boldsymbol{d})$, from

$$X^T DX\boldsymbol{d} = \sum\nolimits_{i=1}^{l} \boldsymbol{x}_i D_{ii} \boldsymbol{x}_i^T \boldsymbol{d}, \tag{3}$$

the two loops can be combined together. For example, if we take the approach of temporarily storing arrays $\hat{\boldsymbol{u}}^p$, $p = 1, \ldots, P$, where P is the total number of threads, then the implementation becomes

```
for $i = 1, \ldots, l$ do in parallel
    $u_i = D_{ii}\boldsymbol{x}_i^T\boldsymbol{d}$
    $\hat{\boldsymbol{u}}^p = \hat{\boldsymbol{u}}^p + u_i\boldsymbol{x}_i$, where $p$ is the thread ID
```

Although the number of operations is the same, because of accessing $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l$ only once rather than twice, this approach should be better than the two-loop setting in Section II-B. However, currently this is not what implemented in LIBLINEAR.

Regarding the implementation efforts and the availability, this approach can be easily implemented by OpenMP. However, for existing sparse matrix packages such as those we will discuss in Sections II-D and II-E, currently they do not support $X^TX\boldsymbol{d}$ type of operations. Therefore, the two operations involving X and X^T must be separately conducted.

The formulation (3) is the foundation of distributed Newton methods in [2], [21], where data are split to disjoint blocks, each node considers its local block and calculates

$$\hat{\boldsymbol{u}}^p = \sum\{\boldsymbol{u}_i\boldsymbol{x}_i \mid i \text{ run by node } p\}$$

and then a reduce operation sums all $\hat{\boldsymbol{u}}^p$ up to get $\bar{\boldsymbol{u}}$. In a shared-memory system, a core does not have its local subset, but we show in Section III-E that letting each thread work on a block of instances is very essential to achieve better speedup.

D. Sparse Matrix-vector Multiplications by Intel MKL

Intel Math Kernel Library (MKL) is a commercial library including optimized routines for linear algebra. It supports fast matrix-vector multiplications for different sparse formats. We consider the commonly used CSR format to store X. Note that MKL provides subroutines for $X^T\boldsymbol{u}$ with X as the input.

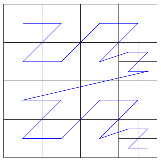
E. Sparse Matrix-vector Multiplications with the Recursive Sparse Blocks Format

Following the concept of splitting the matrix to blocks for dense BLAS implementations, [13] applies a similar idea for sparse matrix-vector multiplications. For example,

$$X = \begin{bmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{bmatrix} \Rightarrow \boldsymbol{u} = X\boldsymbol{d} = \begin{bmatrix} X_{1,1}\boldsymbol{d}_1 + X_{1,2}\boldsymbol{d}_2 \\ X_{2,1}\boldsymbol{d}_1 + X_{2,2}\boldsymbol{d}_2 \end{bmatrix}.$$

This setting possesses the following advantages. First, $X_{1,1}\boldsymbol{d}_1$ and $X_{1,2}\boldsymbol{d}_2$ are independent to each other, so two threads can be used to compute them simultaneously. Second, Each sub-matrix is smaller, so it occupies a smaller part of the higher-level memory (e.g., cache). Then the chance that $\boldsymbol{u}$ and $\boldsymbol{d}$ are thrown out of cache during the computation becomes smaller. We illustrate this point by an extreme example. Suppose the cache is not enough to store $\boldsymbol{x}_i$ and $\boldsymbol{d}$ together. In calculating $\boldsymbol{x}_1^T\boldsymbol{d}$, we sequentially load elements in $\boldsymbol{x}_1$ and $\boldsymbol{d}$ to cache. By the time the inner product is done, $\boldsymbol{d}$'s first several entries have been removed from the cache. Then for $\boldsymbol{x}_2^T\boldsymbol{d}$, the vector $\boldsymbol{d}$ must be loaded again. In contrast, by using blocks, each time only part of $\boldsymbol{x}_i$ and $\boldsymbol{d}$ are used, so the above situation may not occur. In any program, if data are accessed from a lower-level memory frequently, then CPUs must wait until they are available.

While the concept of using blocks is the same as that for dense matrices, the design as well as the implementation for sparse matrices are more sophisticated. Recently, RSB (Recursive Sparse Blocks) format [23] has been proposed as an effective format for fast sparse matrix-vector multiplications. It recursively partitions a matrix in quadrants. In the end a tree structure with sub-matrices in the leaves is generated. The recursive partition terminates according to some conditions on the sub-matrix (e.g., number of non-zeros). The following figure shows an example of an 8×8 matrix in the RSB format.

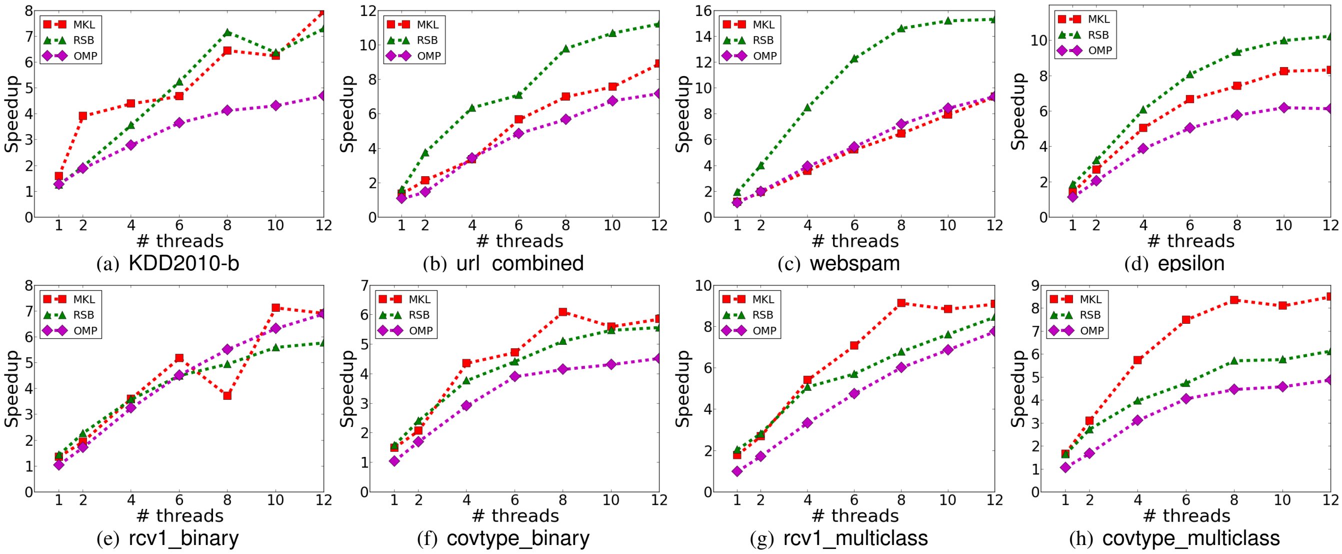


An important concern for using the RSB format is the cost of initial construction, where details are in [24]. Practically, it is observed in [13] that when single thread is used for initial construction as well as matrix-vector multiplication, the construction cost is about 20 to 30 multiplications. This cost is not negligible, but for Newton methods which need quite a few multiplications, we will check in Section III if using RSB is cost-effective.

Like MKL, the package librsb that we will use includes $X^T\boldsymbol{u}$ subroutines by taking X as the input. We do not compare with another block-based approach CSB [12] because [13] has shown that RSB is often competitive with CSB.

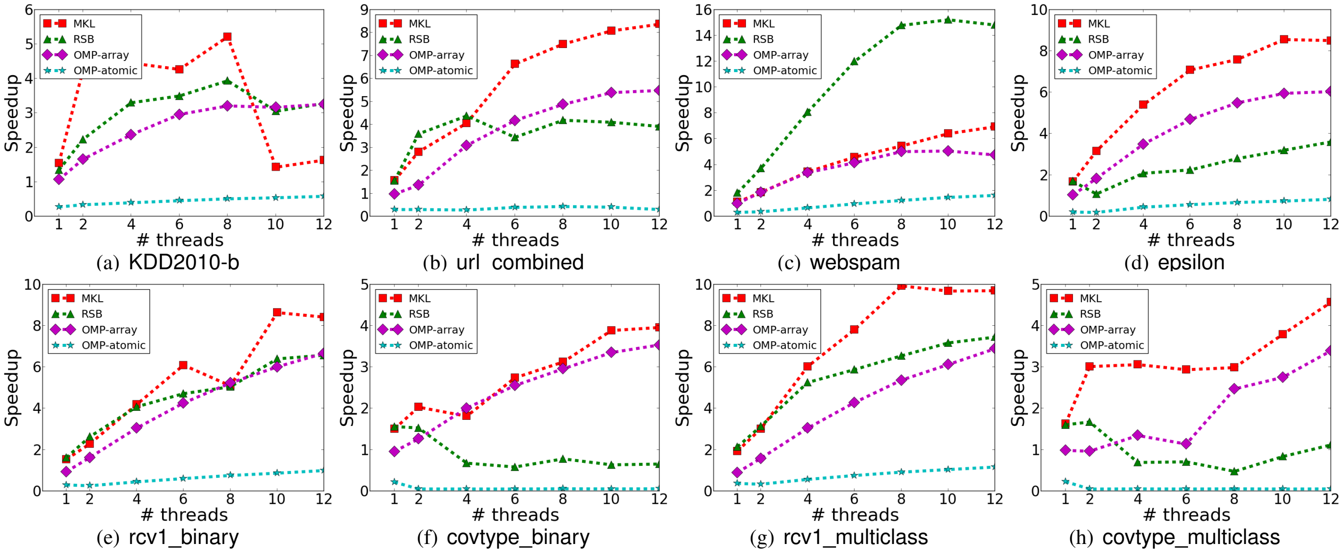
III. EXPERIMENTS

We compare various implementations of sparse matrix-vector multiplications when they are used in a Newton method.



(a) KDD2010-b (b) url_combined (c) webspam (d) epsilon (e) rcv1_binary (f) covtype_binary (g) rcv1_multiclass (h) covtype_multiclass

Fig. 1. Speedup of calculating $X\boldsymbol{d}$



(a) KDD2010-b (b) url_combined (c) webspam (d) epsilon (e) rcv1_binary (f) covtype_binary (g) rcv1_multiclass (h) covtype_multiclass

Fig. 2. Speedup of calculating $X^T\boldsymbol{u}$, where $\boldsymbol{u} = DX\boldsymbol{d}$.

A. Data Sets and Experimental Settings

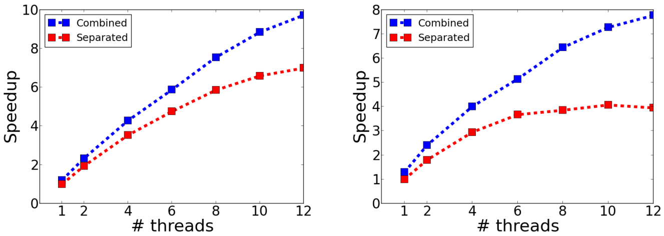
We carefully choose data sets to cover different density and different relationships between numbers of instances and features. All data sets listed in Table I are available at the LIBSVM data set page. Note that we use the tri-gram version of webspam in our experiments.

Besides two-class data sets, we consider several multi-class problems. In LIBLINEAR, a K-class problem is decomposed to K two-class problems by the one-versus-the-rest strategy. Because each two-class problem treats one class as positive and the rest as negative, all the K problems share the same $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l$. Their matrix-vector multiplications involve the same X and X^T. We are interested in such cases because approaches like RSB need the initial construction only once.

We choose $C = 1$ as the regularization parameter in (1). An exception is $C = 0.0625$ for KDD2010-b for shorter training time. Our code is based on LIBLINEAR [18], which implements a trust region Newton method [11]. We compare the following approaches to conduct matrix-vector multiplications in the Newton method.

- Baseline: The one-core implementation in LIBLINEAR (Section II-A) serves as the baseline in our investigation of the speedup. We use version 1.96.
- OpenMP: We employ OpenMP to parallelize the loops for matrix-vector multiplications. The scheduling we used is `schedule(dynamic,256)`, where details will be discussed in Section III-D. For $X^T\boldsymbol{u}$ we have two settings OpenMP-atomic and OpenMP-array. The first one considers atomic operations, while the second stores results of each thread to an array; see Section II-B.
- MKL: We use Intel MKL version 11.2; see Section II-D.
- RSB: the approach described in Section II-E by using the recursive sparse block format. We use version 1.2.0 of the package librsb at http://librsb.sourceforge.net.

Experiments are conducted on machines with 12 cores of Intel Xeon E5-2620 CPUs which has 32K L1-cache, 256K L2-cache and 15360K shared L3-cache. Because past comparisons



(a) rcv1_binary (b) covtype_binary

Fig. 3. Speedup of calculating $X\boldsymbol{d}$ and $X^T\boldsymbol{u}$ separately and together

(e.g., [13]) involving MKL often use the Intel icc compiler, we use it for all our programs with the option `-O3 -xAVX -fPIC -openmp`. Our experiment uses 1, 2, 4, 6, 8, 10, 12 threads because the machine we used has up to 12 cores.

B. Running Time of Sparse Matrix-vector Multiplications in Newton Methods

Although past works have pointed out the importance of sparse matrix-vector multiplications in Newton methods for data classification, no numerical results have been reported to show their cost in the entire optimization procedure. In Table I, we show the percentage of running time spent on matrix-vector multiplications. Clearly matrix-vector multiplications are the computational bottleneck because for almost all problems, more than 90% of the running time is spent on them. Therefore, it is very essential to parallelize the sparse matrix-vector multiplications in a Newton method.

C. Comparison on Matrix-vector Multiplications

Although past works from numerical analysis and parallel processing communities have compared different strategies for matrix-vector multiplications, they did not use matrices from machine learning problems. It is interesting to see if our results are consistent with theirs. We present the result of

$$\text{speedup} = \frac{\text{running time of LIBLINEAR}}{\text{running time of the compared approach}}$$

versus the number of threads. Note that the speedup may be bigger than the number of threads because of the storage and algorithmic differences from the baseline. The speedup for $X\boldsymbol{d}$ and $X^T(DX\boldsymbol{d})$ may be different, so we separately present results in Figures 1 and 2.

Results indicate that all approaches give excellent speedup for calculating $X\boldsymbol{d}$. For some problems, the speedup is so good that it is much higher than the number of threads. For example, using eight threads, RSB achieves speedup higher than 12. This result is possible because while the baseline uses a row-based storage, RSB applies a block storage for better data locality. Between RSB and MKL there is no clear winner. This observation is slightly different from [13], which shows that RSB is better than MKL. One reason might be that [13] experimented with matrices close to squared ones, but ours here are not. Although OpenMP's implementation is the simplest, it enjoys good speedup.

For $X^T\boldsymbol{u}$ with $\boldsymbol{u} = DX\boldsymbol{d}$, in general the speedup becomes worse than that for $X\boldsymbol{d}$. For a few cases the difference is significant (e.g., RSB for covtype_binary). Like the situation for $X\boldsymbol{d}$, there is no clear winner between RSB and MKL. For the two OpenMP implementations for $X^T\boldsymbol{u}$, OpenMP-atomic completely fails to gain any speedup. Although we pointed out in Section II-B the possible delay of using atomic operations, such poor results are not what we expected. In contrast, OpenMP-array gives good speedup and is sometimes even better than RSB or MKL. From this experiment we see that even for the simple implementation by OpenMP, a careful design of algorithms for the loops can make significant differences.

D. Detailed Analysis on Implementations using OpenMP

Following the dramatic difference between OpenMP-atomic and OpenMP-array for $X^T(DX\boldsymbol{d})$, we learned that having a suitable implementation by OpenMP is non-trivial. Therefore, we devote this subsection to thoroughly investigate some implementation issues.

First we discuss the two implementations of computing $X\boldsymbol{d}$ and $X^T(DX\boldsymbol{d})$ separately (Section II-B) and together by (Section II-C). Note that OpenMP-array is used here because of the much better speedup than OpenMP-atomic. Figure 3 presents the speedup of the two settings. The approach of using (3) in Section II-C is better because data instances $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l$ are accessed once rather than twice. We already see improvement in the one-core situation, so (3) is what should have been implemented in LIBLINEAR. Interestingly, the gap between the two approaches widens as the number of threads increases. An explanation is that the doubled number of data accesses cause problems when more threads are accessing data at the same time. Based on this experiment, from now on when we refer to the OpenMP implementation, we mean (3) with the OpenMP-array setting.

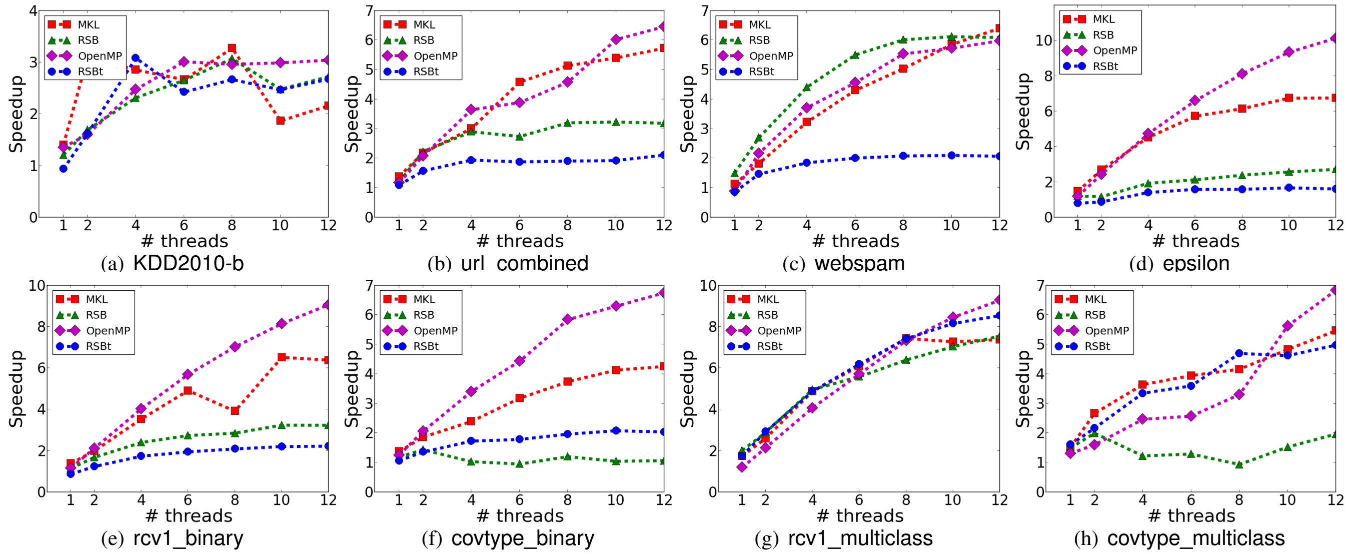
Next we investigate the implementation of OpenMP-array. We show in supplementary materials that without suitable settings (in particular, the scheduling of OpenMP loops), speedup can be poor.

E. Results on Newton Methods for Solving (1)

Figure 4 presents the speedup of the Newton method using various implementations for matrix-vector multiplications. Note that we consider end-to-end running time after excluding the initial data loading and the final model writing. Therefore, the construction time for transforming the row-based format to RSB is now included. This transformation does not occur for other approaches.

For RSB, results in Section III-C indicate that its speedup for $X\boldsymbol{d}$ is excellent, but is poor for $X^T\boldsymbol{u}$. Therefore, we consider a variant RSBt that generates and stores the RSB format of X^T in the beginning. Although more memory is used, we hope that the speedup for $X^T\boldsymbol{u}$ can be as good as that for $X\boldsymbol{d}$.

Results in Figure 4 show that OpenMP (the version presented in Section II-C) is the best for almost all cases, while MKL comes the second. RSB is worse than MKL mainly because of the following reasons. First, from Figure 2, RSB is mostly worse than MKL for $X^T\boldsymbol{u}$. Second, the construction



(a) KDD2010-b (b) url combined (c) webspam (d) epsilon

(e) rcv1_binary (f) covtype_binary (g) rcv1_multiclass (h) covtype_multiclass

Fig. 4. Speedup of the Newton method for training logistic regression

time is not entirely negligible. Regarding RSBt, although we see better speedup for $X^T\boldsymbol{u}$, where details are not shown because of space limitation, the overall speed is worse than RSB for two-class problems. The reason is because that the construction time is doubled. However, for multi-class problems, RSBt may become better than RSB. Because several binary problems on the same data instances are solved, the construction time for the RSB format of X^T has less impact on the total training time.

The speedup for KDD2010-b is worse than others because it has the lowest percentage of running time on matrix-vector multiplications (see Table I).

We must point out that one factor to the superiority of OpenMP over others is that it implements (3) by considering $X\boldsymbol{d}$ and $X^T\boldsymbol{u}$ together. In this regard, future development and support of $XX^T(\cdot)$ type of operations in sparse matrix packages is a important direction.

F. Summary of Experiments

1) It is more difficult to gain speedup for $X^T\boldsymbol{u}$ than $X\boldsymbol{d}$. Improving the speedup of $X^T\boldsymbol{u}$ should be a focus for the future development of programs for sparse matrix-vector multiplications.
2) With settings discussed in Section III-D, simple implementations by OpenMP can achieve excellent speedup and beat existing state-of-the-art packages.

References

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

[2] Y. Zhuang, W.-S. Chin, Y.-C. Juan, and C.-J. Lin, "Distributed Newton method for regularized logistic regression," in *PAKDD*, 2015.

[3] T. Yang, "Trading computation for communication: Distributed stochastic dual coordinate ascent," in *NIPS*, 2013.

[4] M. Jaggi, V. Smith, M. Takáč, J. Terhorst, T. Hofmann, and M. I. Jordan, "Communication-efficient distributed dual coordinate ascent," in *NIPS*, 2014.

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

[6] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Found. Trends Mach. Learn.*, vol. 3, pp. 1–122, 2011.

[7] 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.

[8] A. Bordes, L. Bottou, and P. Gallinari, "SGD-QN: Careful quasi-Newton stochastic gradient descent," *JMLR*, vol. 10, pp. 1737–1754, 2009.

[9] F. Niu, B. Recht, C. Ré, and S. J. Wright, "HOGWILD!: a lock-free approach to parallelizing stochastic gradient descent," in *NIPS*, 2011.

[10] S. S. Keerthi and D. DeCoste, "A modified finite Newton method for fast solution of large scale linear SVMs," *JMLR*, vol. 6, pp. 341–361, 2005.

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

[12] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *SPAA*, 2009.

[13] M. Martone, "Efficient multithreaded untransposed, transposed or symmetric sparse matrix–vector multiplication with the recursive sparse blocks format," *Parallel Comput.*, vol. 40, pp. 251–270, 2014.

[14] J.-H. Byun, R. Lin, K. A. Yelick, and J. Demmel, "Autotuning sparse matrix-vector multiplication for multicore," EECS, UC Berkeley, Tech. Rep., 2012.

[15] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for Fortran usage," *ACM TOMS*, vol. 5, pp. 308–323, 1979.

[16] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of Fortran basic linear algebra subprograms," *ACM TOMS*, vol. 14, pp. 1–17, 1988.

[17] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM TOMS*, vol. 16, pp. 1–17, 1990.

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

[19] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, pp. 46–55, 1998.

[20] P. Komarek and A. W. Moore, "Making logistic regression a core data mining tool," Carnegie Mellon University, Tech. Rep., 2005.

[21] C.-Y. Lin, C.-H. Tsai, C.-P. Lee, and C.-J. Lin, "Large-scale logistic regression and linear support vector machines using Spark," in *IEEE BigData*, 2014.

[22] W. F. Tinney and J. W. Walker, "Direct solutions of sparse network equations by optimally ordered triangular factorization," *PIEEE*, vol. 55, pp. 1801–1809, 1967.

[23] M. Martone, S. Filippone, S. Tucci, M. Paprzycki, and M. Ganzha, "Utilizing recursive storage in sparse matrix-vector multiplication-preliminary considerations," in *CATA*, 2010.

[24] M. Martone, S. Filippone, S. Tucci, and M. Paprzycki, "Assembling recursively stored sparse matrices," in *IMCSIT*, 2010.