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

Wei-Lin Chiang
Department of Computer Science
National Taiwan University

Joint work with Mu-Chu Lee and Chih-Jen Lin

Outline

Outline

Linear classification

- Linear classification such as logistic regression is popular and efficient for some problems (e.g., document classification)
- But training on large-scale (terabyte level) data is still a time-consuming process

Speed up linear classification

- We focus on parallel computing on shared memory systems in this work
- Difficulty: Some algorithms such as stochastic gradient, coordinate descent are sequential because the next iteration relies on the current one
- Many works modify serial algorithms to parallel settings, but proving the convergence may be difficult

How to speed up?

- We aim not to modify the algorithms, but employ parallel matrix operations
- The main advantage is that the same method can be used and the convergence still holds

Outline

Newton method for logistic regression

- Newton method is a popular optimization method for logistic regression
- It is known that at each Newton iteration, the main computational bottleneck is computing a sequence of Hessian-vector products
- For logistic regression, the Hessian-vector product can be simplified as the following,

$$\nabla^2 f(\boldsymbol{w})\boldsymbol{d} = \boldsymbol{d} + C \cdot X^T(D(X\boldsymbol{d})), X \text{ is data matrix}$$

- The main computations are $X\boldsymbol{d}$ and $X^T(DX\boldsymbol{d})$

Matrix-vector operations generally takes more than 90% of the training time

Data set	#instances	#features	ratio
kddb	19,264,097	29,890,095	82.11%
url_combined	2,396,130	3,231,961	94.83%
webspam	350,000	16,609,143	97.95%
rcv1_binary	677,399	47,236	97.88%
covtype_binary	581,012	54	89.20%
epsilon_normalized	400,000	2,000	99.88%
rcv1_multiclass	518,571	47,236	97.04%
covtype_multiclass	581,012	54	89.06%

Fast matrix-vector multiplications

- For dense matrices, optimized BLAS can be significantly faster than a naive implementation
- However, the data matrix X is usually sparse
- Fortunately, some recent progress has been made for sparse matrix-vector multiplications (e.g. Bordes et al., 2009; Martone, 2014)
- Intel Math Kernel Library (MKL) started supporting sparse BLAS recently

Outline

Proposed OpenMP implementation

In $\nabla^2 f(\boldsymbol{w}) \cdot \boldsymbol{d}$ product, need to speed up $X\boldsymbol{d}$ and $X^T\boldsymbol{u}$

- Assume that X is in a row-oriented sparse format

$$X = \begin{bmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_l^T \end{bmatrix} \text{ and } \boldsymbol{u} = X\boldsymbol{d} = \begin{bmatrix} \boldsymbol{x}_1^T\boldsymbol{d} \\ \vdots \\ \boldsymbol{x}_l^T\boldsymbol{d} \end{bmatrix}$$

- Because rows can be easily accessed, we can parallelize the l independent inner products
- Proper scheduling is needed (since the data matrix may not be balanced)

OpenMP implementation (Cont'd)

- For the other matrix-vector multiplication,

$$\bar{\boldsymbol{u}} = X^T \boldsymbol{u} = \left[\boldsymbol{x}_1 \ldots \boldsymbol{x}_l\right] \cdot \begin{bmatrix} u_1 \\ \vdots \\ u_l \end{bmatrix} = u_1 \boldsymbol{x}_1 + \cdots + u_l \boldsymbol{x}_l$$

- Because matrix X is row-oriented, accessing columns in X^T is much easier than rows
- We can use the following loop to calculate $X^T \boldsymbol{u}$

1: **for** $i = 1, \ldots, l$ **do**

2: $\qquad \bar{\boldsymbol{u}} \leftarrow \bar{\boldsymbol{u}} + u_i \boldsymbol{x}_i$

Atomic operations

- However, for parallelization, different threads may update the same component at the same time

  for $i = 1, \ldots, l$ **do** in parallel
      for $(\boldsymbol{x}_i)_s \neq 0$ **do**
          $\bar{u}_s \leftarrow \bar{u}_s + u_i(\boldsymbol{x}_i)_s$

- Atomic operation could be used

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

Reduce operations

- Another method is using temporary arrays maintained by each thread, and summing up them in the end
- That is, store

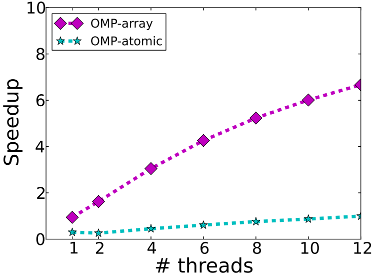
$$\hat{\boldsymbol{u}}^p = \sum_i \{u_i \boldsymbol{x}_i \mid i \text{ run by thread} \mathrm{p}\}$$

and then

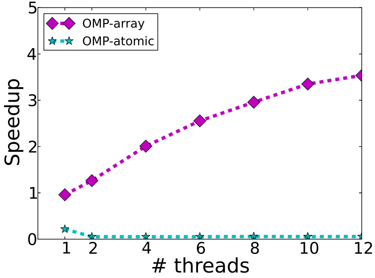
$$\bar{\boldsymbol{u}} = \sum_p \hat{\boldsymbol{u}}^p$$

Atomic operation has almost no speedup

- Reduce operations are superior to atomic operations



rcv1_binary

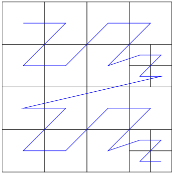


covtype_binary

- Subsequently we use the reduce operations

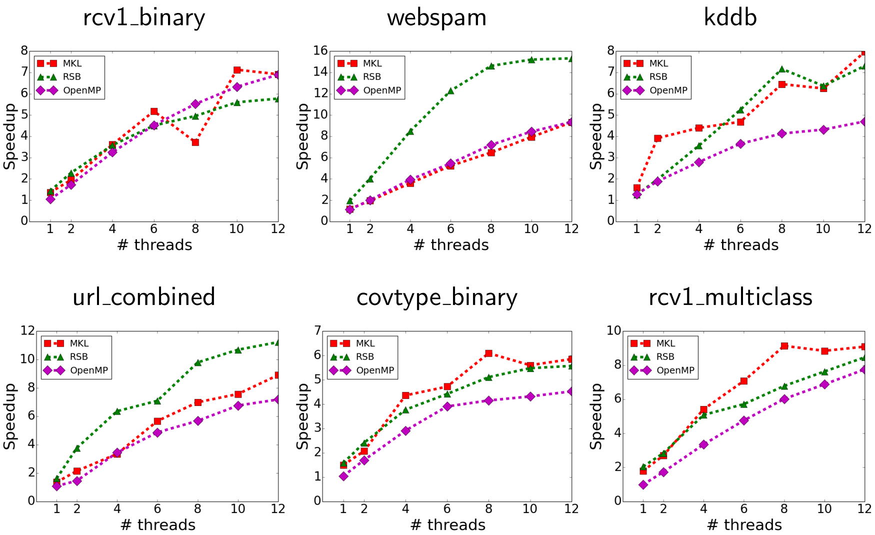
Recursive Sparse Blocks (Martone, 2014)

- RSB (Recursive Sparse Blocks) is an effective format for fast parallel sparse matrix-vector multiplications

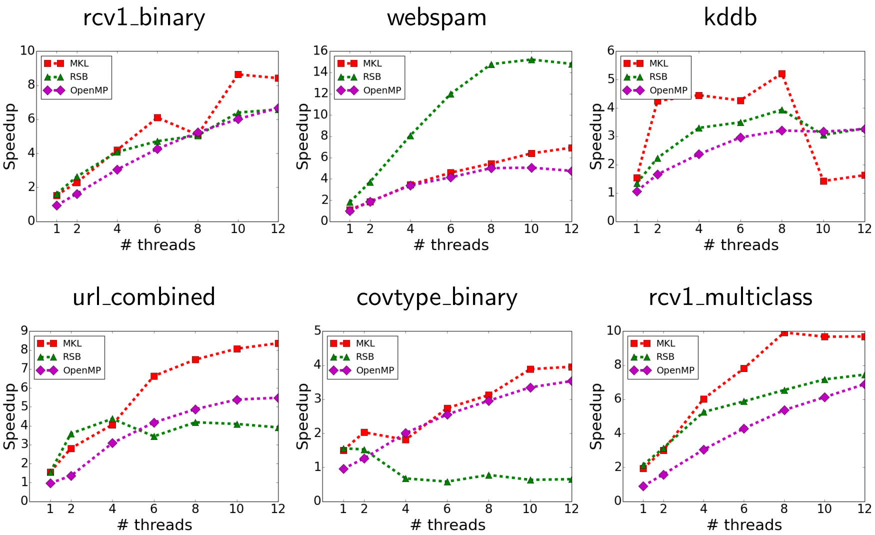


- It recursively partitions a matrix to be like the figure
- Locality of memory references improved, but the construction time is not negligible

Speedup of Xd: all are excellent



More difficult to speed up $X^T \boldsymbol{u}$

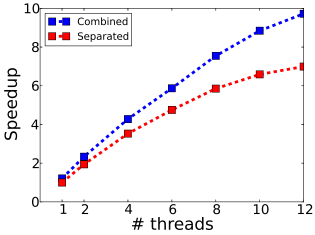


Improvement of OpenMP implementation

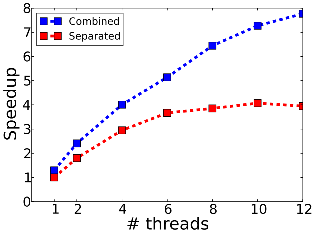
- Instead of computing $X\boldsymbol{d}$ and $X^T(DX\boldsymbol{d})$ separately, we combine them into a single loop

$$X^T DX\boldsymbol{d} = \sum_{i=1}^{l} \boldsymbol{x}_i D_{ii} \boldsymbol{x}_i^T \boldsymbol{d}$$

- Better speedup as memory accesses reduced

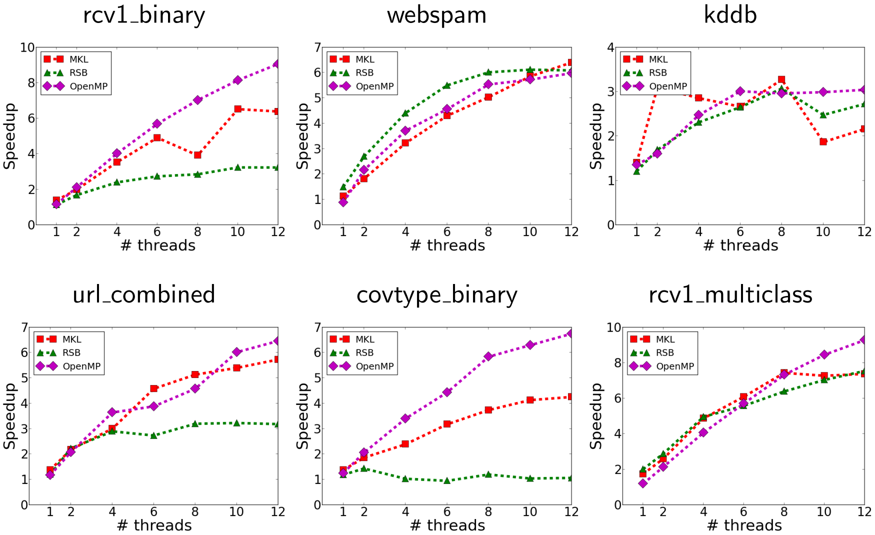


rcv1_binary



covtype_binary

Speedup of total training time



Outline

Conclusions

- With appropriate settings, simple implementations by OpenMP can achieve excellent speedup
- Based on this research, a multi-core extension of the popular package LIBLINEAR is available at: `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/multicore-liblinear`
- There are already many users. For example, one user from USC uses this tool to reduce his training time from over 30 hours to 5 hours