Available online at www.sciencedirect.com
ScienceDirect
Neurocomputing 70 (2006) 93–104
NEUROCOMPUTING
www.elsevier.com/locate/neucom

Developing parallel sequential minimal optimization for fast training support vector machine ☆

L.J. Cao[a,*], S.S. Keerthi[b], C.J. Ong[b], P. Uvaraj[c], X.J. Fu[c], H.P. Lee[c]

[a]Financial Studies of Fudan University, HanDan Road, ShangHai, 200433, P.R. China
[b]Department Of Mechanical Engineering, National University of Singapore, 10 Kent Ridge Crescent, 119260, Singapore, Singapore
[c]Institute of High Performance Computing, 1 Science Park Road, #01-01 the Capricorn, Science Park II, 117528 Singapore, Singapore

Received 22 June 2005; received in revised form 16 March 2006; accepted 17 May 2006
Communicated by J. Tin-Yau Kwok
Available online 7 July 2006

Abstract

A parallel version of sequential minimal optimization (SMO) is developed in this paper for fast training support vector machine (SVM). Up to now, SMO is one popular algorithm for training SVM, but it still requires a large amount of computation time for solving large size problems. The parallel SMO is developed based on message passing interface (MPI). Unlike the sequential SMO which handle all the training data points using one CPU processor, the parallel SMO first partitions the entire training data set into smaller subsets and then simultaneously runs multiple CPU processors to deal with each of the partitioned data sets. Experiments show that there is great speedup on the adult data set, the MNIST data set and IDEVAL data set when many processors are used. There are also satisfactory results on the Web data set. This work is very useful for the research where multiple CPU processors machine is available.

Keywords: Support vector machine (SVM); Sequential minimal optimization (SMO); Message passing interface (MPI); Parallel algorithm

1. Introduction

Recently, a lot of research work has been done on support vector machines (SVMs), mainly due to their impressive generalization performance in solving various machine learning problems [1,2,15]. Given a set of data points $\{(X_i,y_i)\}_i^l$ ($X_i \in R^d$ is the input vector of ith training data pattern; $y_i \in \{-1,1\}$ is its class label; l is the total number of training data patterns), training an SVM in classification is equivalent to solving the following linearly constrained convex quadratic programming (QP) problem.

$$\text{maximize}: R(\alpha_i) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l} \alpha_i\alpha_j y_i y_j K(X_i, X_j) \quad (1)$$

$$\text{subject to}: \sum_{i=1}^{l} \alpha_i y_i = 0, \quad (2)$$

$$0 \leqslant \alpha_i \leqslant C, i = 1, \cdots, l,$$

where $K(X_i,X_j)$ is the kernel function. The mostly widely used kernel function is the Gaussian function $e - ||X_i - X_j||^2/\sigma^2$, where σ^2 is the width of the Gaussian kernel. α_i is the Lagrange multiplier to be optimized. For each of training data patterns, one α_i is associated. C is the regularization constant pre-determined by users. After solving the QP problem (1), the following decision function is used to determine the class label for a new data pattern:

$$f(X) = \sum_{i=1}^{l} \alpha_i y_i K(X_i, X) - b, \quad (3)$$

where b is obtained from the solution of (1). A detailed description of Eq. (3) can be referred to our previous work [7].

So the main problem in SVM is reduced to solving the QP problem (1), where the number of variables α_i to be

☆The research work is funded by National Natural Science Research Fund No. 70501008 and sponsored by Shanghai Pujiang program.
*Corresponding author.
E-mail address: caolj@fudan.edu.cn (L.J. Cao).

0925-2312/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.neucom.2006.05.007

optimized is equal to the number of training data patterns l. For small size problems, standard QP techniques such as the projected conjugate gradient can be directly applied. But for large size problems, standard QP techniques are not useful as they require a large amount of computer memory to store the kernel matrix K as the number of elements of K is equal to the square of the number of training data patterns.

For making SVM more practical, special algorithms are developed, such as Vapnik's chunking [14], Osuna's decomposition [10] and Joachims's SVM^{light} [8]. They make the training of SVM possible by breaking the large QP problem (1) into a series of smaller QP problems and optimizing only a subset of training data patterns at each step. The subset of training data patterns optimized at each step is called the working set. Thus, these approaches are categorized as the working set methods.

Based on the idea of the working set methods, Platt [12] proposed the sequential minimal optimization (SMO) algorithm which selects the size of the working set as two and uses a simple analytical approach to solve the reduced smaller QP problems. There are some heuristics used for choosing two α_i to optimize at each step. As pointed out by Platt, SMO scales only quadratically in the number of training data patterns, while other algorithms scales cubically or more in the number of training data patterns. Later, Keerthi et al. [9] ascertained inefficiency associated with Platt's SMO and suggested two modified versions of SMO that are much more efficient than Platt's original SMO. The second modification is particular good and used in popular SVM packages such as LIBSVM [3]. We will refer to this modification as the modified SMO algorithm.

Recently, there are few works on developing parallel implementation of training SVMs [4,6,16]. In [4], a mixture of SVMs are trained in parallel using the subsets of a training data set. The results of each SVM are then combined by training another multi-layer perceptron. The experiment shows that the proposed parallel algorithm can provide much efficiency than using a single SVM. In the algorithm proposed by Dong et al. [6], multiple SVMs are also developed using subsets of a training data set. The support vectors in each SVM are then collected to train another new SVM. The experiment demonstrates much efficiency of the algorithm. Zanghirati and Zanni [16] also proposed a parallel implementation of SVM^{light} where the whole quadratic programming problem is split into smaller subproblems. The subproblems are then solved by a variable projection method. The results show that the approach is comparable on scalar machines with a widely used technique and can achieve good efficiency and scalability on a multiprocessor system.

This paper proposes a parallel implementation of the modified SMO based on the multiprocessor system for speeding up the training of SVM, especially with the aim of solving large size problems. In this paper, the parallel SMO is developed using message passing interface (MPI) [11]. Unlike the sequential SMO which handles the entire training data set using a single CPU processor, the parallel SMO first partitions the entire training data set into smaller subsets and then simultaneously runs multiple CPU processors to deal with each of the partitioned data sets. On the adult data set the parallell SMO using 32 CPU processors is more than 21 times faster than the sequential SMO. On the web data set,the parallel SMO using 30 CPU processors is more than 10 times faster than the sequential SMO. On the MNIST data set the parallel SMO using 30 CPU processors on the averaged time of "one-against-all" SVM classifiers is more than 21 times faster than the sequential SMO.

This paper is organized as follows. Section 2 gives an overview of the modified SMO. Section 3 describes the parallel SMO developed using MPI. Section 4 presents the experiment indicating the efficiency of the parallel SMO. A short conclusion then follows.

2. A brief overview of Keerthi's modified SMO

We begin the description of the modified SMO by giving the notation used. Let $I_0 = \{i : y_i = 1, 0 < \alpha_i < C\} \cup \{i : y_i = -1, 0 < \alpha_i < C\}$, $I_1 = \{i : y_i = 1, \alpha_i = 0\}$, $I_2 = \{i : y_i = -1, \alpha_i = C\}$, $I_3 = \{i : y_i = 1, \alpha_i = C\}$ and $I_4 = \{i : y_i = -1, \alpha_i = 0\}$. $I = \cup I_i, i = 0, \cdots, 4$ denotes the index of training data patterns. $F_i = \sum_{j=1}^{l} \alpha_j y_j K(X_j, X_i) - y_i$. $b_{\text{up}} = \min\{F_i : i \in I_0 \cup I_1 \cup I_2\}$, $I_{\text{up}} = \arg\min F_i$. $b_{\text{low}} = \max\{F_i : i \in I_0 \cup I_3 \cup I_4\}$, $I_{\text{low}} = \arg\max F_i$. $\tau = 10^{-6}$.

The idea of the modified SMO is to optimize the two α_i associated with b_{up} and b_{low} according to (4) and (5) at each step. Their associated index are I_{up} and I_{low}.

$$\alpha_2^{\text{new}} = \alpha_2^{\text{old}} - \frac{y_2(F_1^{\text{old}} - F_2^{\text{old}})}{\eta}, \tag{4}$$

$$\alpha_1^{\text{new}} = \alpha_1^{\text{old}} + s(\alpha_2^{\text{old}} - \alpha_2^{\text{new}}), \tag{5}$$

where the variables associated with the two α_i are represented using the subscripts "1" and "2". $s = y_1 y_2$. $\eta = 2K(X_1, X_2) - K(X_1, X_1) - K(X_2, X_2)$. α_1^{new} and α_2^{new} need to be clipped to $[0, C]$. That is, $0 \leqslant \alpha_1^{\text{new}} \leqslant C$ and $0 \leqslant \alpha_2^{\text{new}} \leqslant C$.

After optimizing α_1 and α_2, F_i, denoting the error on the ith training data pattern, is updated according to the following:

$$F_i^{\text{new}} = F_i^{\text{old}} + (\alpha_1^{\text{new}} - \alpha_1^{\text{old}}) y_1 K(X_1, X_i) + (\alpha_2^{\text{new}} - \alpha_2^{\text{old}}) y_2 K(X_2, X_i). \tag{6}$$

Based on the updated values of F_i, b_{up} and b_{low} and the associated index I_{up} and I_{low} are updated again according to their definitions. The updated values are then used to choose another two new α_i to optimize at the next step.

In addition, the value of Eq. (1), represented by *Dual*, is updated at each step by

$$Dual^{\text{new}} = Dual^{\text{old}} - \frac{\alpha_1^{\text{new}} - \alpha_1^{\text{old}}}{y_1}(F_1^{\text{old}} - F_2^{\text{old}}) + \frac{1}{2}\eta\left(\frac{\alpha_1^{\text{new}} - \alpha_1^{\text{old}}}{y_1}\right)^2 \tag{7}$$

and *DualityGap*, representing the difference between the primal and the dual objective function in SVM, is calculated by (8).

$$DualityGap = \sum_{i=0}^{l} \alpha_i y_i F_i + \sum_{i=o}^{l} \varepsilon_i, \tag{8}$$

$$\text{where} \begin{cases} \varepsilon_i = C\max(0, b - F_i) & \text{if } y_i = 1, \\ \varepsilon_i = C\max(0, -b + F_i) & \text{if } y_i = -1. \end{cases}$$

A more detailed description of *Dual* and *DualityGap* can be referred to the paper [9]. *Dual* and *DualityGap* are used for checking the convergence of the program. A simple description of the modified SMO in the sequential form can be summarized as:

2.1. Sequential SMO Algorithm

Initialize $\alpha_i = 0$, $F_i = -y_i$, $Dual = 0$, $i = 1, \cdots, l$
Calculate b_{up}, I_{up}, b_{low}, I_{low}, *DualityGap*
Until $DualityGap \leqslant \tau |Dual|$

(1) Optimize $\alpha_{I_{up}}$, $\alpha_{I_{low}}$ based on Eqs. (4) and (5).
(2) Update F_i, $i = 1, \cdots, l$ based on Eq. (6).
(3) Calculate b_{up}, I_{up}, b_{low}, I_{low}, *DualityGap* and update *Dual*.
Repeat

3. The parallel SMO

MPI is not a new programming language, but a library of functions that can be used in C, C++ and FORTRAN [11]. MPI allows one to easily implement an algorithm in parallel by running multiple CPU processors for improving efficiency. The "single program multiple data (SPMD)" mode where different processors execute the same program but different data is generally used in MPI for developing parallel programs.

In the sequential SMO algorithm, most of computation time is dominated by updating F_i array at the iteration (2), as it includes the kernel evaluations and is also required for every training data pattern. As shown in our experiment, over 90% of the total computation time of the sequential SMO is used for updating F_i array. So the first idea for us to improve the efficiency of SMO is to develop the parallel program for updating F_i array. According to (6), updating F_i array is independently evaluated one training data pattern at a time, so the "SPMD" mode can be used to execute this program in parallel. That is, the entire training data set is firstly equally partitioned into smaller subsets according to the number of processors used. Then each of the partitioned subsets is distributed into one CPU processor. By executing the program of updating F_i array using all the processors, each processor will update a different subset of F_i array based on its assigned training data patterns. In such a way, much computation time could be saved. Let p denote the total number of processors used, t_f is the amount of computation time used for updating F_i array in the sequential SMO. By using the parallel program of updating F_i array, the amount of computation time used to update F_i array is almost reduced to $(1/p)t_f$.

Besides updating F_i array, calculating b_{up}, b_{low}, I_{up} and I_{low} can also be performed in parallel as the calculation involves examining all the training data points. By executing the program of calculating b_{up}, b_{low}, I_{up} and I_{low} using all the processors, each processor could obtain one b_{up} and one b_{low} as well as the associated I_{up} and I_{low} based on its assigned training data patterns. The b_{up}, I_{up}, b_{low} and I_{low} of each processor are not global in the sense they are obtained only based on a subset of all the training data patterns. The global b_{up} and global b_{low} are, respectively, the minimum value of b_{up} of each processor and the maximum value of b_{low} of each processor, as described in Section 2. By determining the global b_{up} and the global b_{low}, the associated I_{up} and I_{low} can thus be found out. The corresponding two α_i are then optimized by using any one CPU processor.

According to (8), calculating *DualityGap* is also independently evaluated one training data pattern at a time. So this program can also be executed in parallel using the "SPMD" mode. By running the program of Eq. (8) using multiple CPU processors, each processor will calculate a different subset of *DualityGap* based on its assigned training data patterns. The value of *DualityGap* on the entire training data patterns is the sum of the *DualityGap* of all the processors.

In summary, based on the "SPMD" parallel mode, the parallel SMO update F_i array and calculate b_{up}, b_{low}, I_{up}, I_{low}, and *DualityGap* at each step in parallel using multiple CPU processor. The calculation of other parts of SMO which take little time is done using one CPU processor, which is the same as used in the sequential SMO. Due to the use of multiple processors, communication among processors is also required in the parallel SMO, such as getting global b_{up}, I_{up}, b_{low} and I_{low} from b_{up}, I_{up}, b_{low} and I_{low} of each processor. For making the parallel SMO efficient, the communication time should be kept small. A brief description of executing the parallel SMO can be summarized as follows.

3.1. Parallel SMO algorithm

Notation: p is the total number of processors used. $\{l^k\}_{k=1}^{p}$, $\cup_{k=1\to p} l^k = l$ is a subset of all the training data patterns and assigned to processor k. F_i^k, b_{up}^k, I_{up}^k, b_{low}^k, I_{low}^k, $DualityGap^k$, α_i^k, $i \in l^k$ denote the variables associated with processor k. $F_i^k = \sum_{j=1}^{l} \alpha_j^k y_j K(X_j, X_i) - y_i$. $b_{up}^k = \min\{F_i^k : i \in I_0 \cup I_1 \cup I_2 \cup l^k\}$, $I_{up}^k = \arg\min_i F_i^k$. $b_{low}^k = \max\{F_i^k : i \in I_0 \cup I_3 \cup I_4 \cup l^k\}$, $I_{low}^k = \arg\max_i F_i^k$. b_{up}, I_{up}, b_{low}, b_{low}, and *DualityGap* still denote the variables on the entire training data patterns. $b_{up} = \max\{b_{up}^k\}$, $I_{up} =$

$\arg_{I_{\text{up}}^k} b_{\text{up}} = b_{\text{up}}^k$, $b_{\text{low}} = \max\{b_{\text{low}}^k\}$, $I_{\text{low}} = \arg_{I_{\text{low}}^k} b_{\text{low}} = b_{\text{low}}^k$, *Duality Gap* $= \sum_{k=1}^{p} DualityGap^k$.

Initialize $\alpha_i^k = 0$, $F_i^k = -y_i$, $Dual = 0$, $i \in l^k$, $k = 1, \cdots, p$
Calculate b_{up}^k, I_{up}^k, b_{low}^k, I_{low}^k, $DualityGap^k$
Obtain b_{up}, I_{up}, b_{low}, I_{low}, and *DualityGap*
Until $DualityGap \leqslant \tau|Dual|$

(1) Optimize $\alpha_{I_{\text{up}}}$, $\alpha_{I_{\text{low}}}$.
(2) Update F_i^k, $i \in l^k$.
(3) Calculate b_{up}^k, I_{up}^k, b_{low}^k, I_{low}^k, $DualityGap^k$.
(4) Obtain b_{up}, I_{up}, b_{low}, I_{low}, *DualityGap* and update *Dual*.

Repeat.

A more detailed description of the parallel SMO can be referred to the pseudo-code in Appendix A. An extension of the parallel SMO for regression estimation is described in Appendix B.

4. Experiment

The parallel SMO is tested against the sequential SMO using three benchmarks: the adult data set, the web data set and the MNIST data set. Both algorithms are written in C. Both algorithms are run on IBM p690 Regata Super-Computer which has a total of seven nodes, with each node having 32 power PC_POWER4 1.3 GHz processors. For ensuring the same accuracy in the sequential SMO and the parallel SMO, the stop criteria used in both algorithms such as the value of τ are all the same.

4.1. Adult data set

The first data set used to test the parallel SMO's speed is the UCI adult data set [12]. The task is to predict whether the household has an income larger than \$50,000 based on a total of 123 binary attributes. For each input vector, only an average of 14 binary attributes are true, represented by the value of "1". Other attributes are all false, represented by the value of "0". There are a total of 28,956 data patterns in the training data set.

The Gaussian kernel is used for both the sequential SMO and the parallel SMO. The values of Gaussian variance σ^2 and C are arbitrarily used as 100 and 1. These values are not necessarily ones that give the best generalization performance of SVM, as the purpose of this experiment is only for evaluating the computation time of two algorithms. Moreover, the LIBSVM[version2.8] proposed by Chang and Lin [9] is also investigated using a single processor on the experiment. The aim is to see whether the kernel cache used in LIBSVM can provide efficiency in comparison with the sequential SMO without kernel cache.

The elapsed time (measured in seconds) with different number of processors in the sequential SMO, the parallel SMO and LIBSVM is given in Table 1, as well as the number of converged support vectors (denoted as SVs) and bounded support vectors with $\alpha_i = C$ (denoted as BSVs). From the table, it can be observed that the elapsed time of the parallel SMO gradually reduces with an increase in the number of processors. It can be reduced by almost half with the use of two processors and almost three-quarters with the use of four processors, etc. This result demonstrates that the parallel SMO is efficient in reducing the training time of SVM. Moreover, the parallel SMO using one CPU processor takes slightly more time than the sequential SMO, due to the use of MPI programs. The table also shows that LIBSVM running on the single processor requires less time than that of the sequential SMO. This demonstrates that the kernel caching is effective in reducing the computation time of the kernel evaluation.

For evaluating the performance of the parallel SMO, the following two criteria are used: speedup and efficiency. They are, respectively, defined by

$$\text{speedup} = \frac{\text{the elapsed time of the sequential SMO}}{\text{the elapsed time of the parallel SMO}}, \tag{9}$$

$$\text{efficieny} = \frac{\text{speedup}}{\text{number of processors}}. \tag{10}$$

The speedup of the parallel SMO with respect to different number of processors is illustrated in Fig. 1. The figure shows that up to 16 processors the parallel SMO scales almost linearly with the number of processors. After that, the scalability of the parallel SMO is slightly reduced. The maximum value of the speedup is more than 21, corresponding to the use of 32 processors. The result means that the training time of the parallel SMO by running 32 processors is only about $\frac{1}{21}$ of that of the sequential SMO, which is acceptable.

The efficiency of the parallel SMO with different number of processors is illustrated in Fig. 2. As shown in the figure,

Table 1
The elapsed time (seconds) used in the sequential SMO and the parallel SMO on the adult data set

	LIBSVM	Sequential SMO	Parallel SMO					
			1P	2P	4P	8P	16P	32P
Time(s)	1132.1	2026.1	2048.1	1026.8	521.9	275.8	145.1	93.8
SVs	10563	10593	10593	10593	10593	10593	10593	10593
BSVs	8949	8931	8931	8931	8931	8931	8931	8931

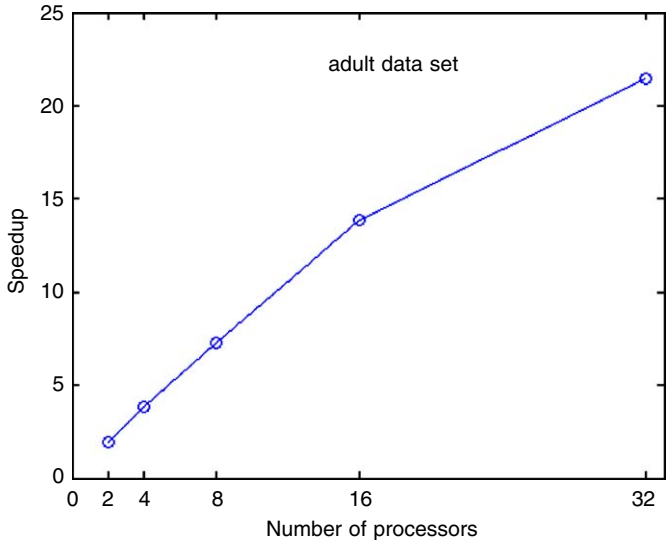

Fig. 1. The speedup of the parallel SMO on the adult data set.

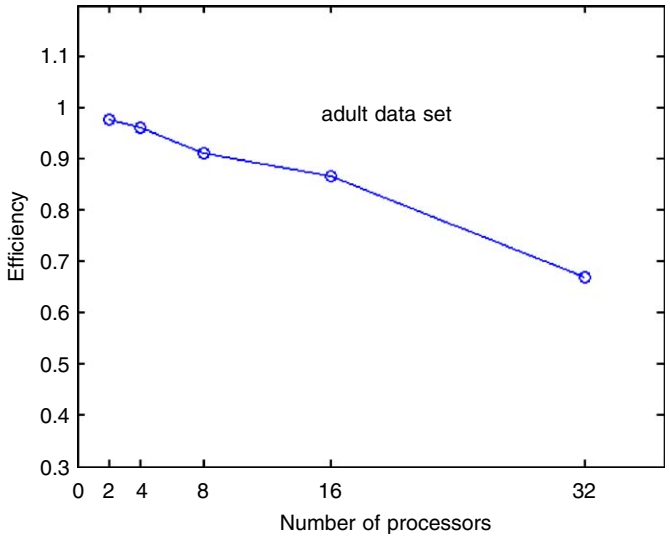

Fig. 2. The efficiency of the parallel SMO on the adult data set.

the value of the efficiency of the parallel SMO is 0.9788 when two processors are used. It gradually reduces as the number of processor increases. The reason may lie in that the use of more processors will lead to more communication time, thus reducing the efficiency of the parallel SMO.

For a better understanding of the cost of various subparts in the parallel SMO, the computation time in different components (I/O; initialization; optimizing $\alpha_{I_{up}}$ and $\alpha_{I_{low}}$; updating F_i^k and calculating b_{up}^k, I_{up}^k, b_{low}^k, I_{low}^k, $DualityGap^k$; and obtaining b_{up}, I_{up}, b_{low}, I_{low}, $DualityGap$) is reported in Table 2. The time for updating F_i^k and calculating b_{up}^k, I_{up}^k, b_{low}^k, I_{low}^k, $DualityGap^k$ is called as the parallel time as the involved calculations are done in parallel and the time for obtaining b_{up}, I_{up}, b_{low}, I_{low}, $DualityGap$ is called as the communication time as many

Table 2
The computation time in different components of the parallel SMO on the adult data set

Components	Number of processors					
	1P	2P	4P	8P	16P	32P
I/O	1	1	1	1	1	1
Initialization	0	0	0	0	0	0
a_{I_up}, a_{I_low}	0	0	0	0	0	0
b_{up}, I_{up}, b_{low}, I_{low}, $DualityGap$	0	2	6	8	8	18
F^k, b_{up}^k, I_{up}^k, b_{low}^k, I_{low}^k, $DualityGap^k$	2041	1017	507	261	129	66

Table 3
The elapsed time (s) used in the sequential SMO and the parallel SMO on the web data set

	LIBSVM	Sequential SMO	Parallel SMO					
			1P	2P	4P	8P	16P	30P
Time(s)	104.3	172.8	191.3	95.7	52.4	31.6	23.1	16.0
SVs	632	624	624	624	624	624	624	624
BSVs	205	232	232	232	232	232	232	232

processors are included in the calculation. The table shows that the time for I/O, initialization, and optimizing $\alpha_{I_{up}}$ and $\alpha_{I_{low}}$ is little and irrelevant to the number of processor, while a large amount of time is used in the parallel time, which means that the updating of F_i^k and the calculating of b_{up}^k, I_{up}^k, b_{low}^k, I_{low}^k, $DualityGap^k$ had better be performed in parallel using multiple processors. As expected, the parallel time decreases with the increase of the number of processors. In contrast, the communication time slightly increases with the increase of the number of processors. This exactly explains why the efficiency of the parallel SMO decreases as the number of processors increases.

4.2. Web data set

The web data set is examined in the second experiment [12]. This problem is to classify whether a web page belongs to a certain category or not. There are a total of 24,692 data patterns in the training data set, with each data pattern composed of 300 spare binary keyword attributes extracted from each web page.

For this data set, the Gaussian function is still used as the kernel function of the sequential SMO and the parallel SMO. The values of Gaussian variance σ^2 and C are respectively used as 0.064 and 64.

The elapsed time with different number of processors used in the sequential SMO, the parallel SMO and LIBSVM is given in Table 3, as well as the total number of support vectors and bounded support vectors. Same as in the adult data set, the elapsed time of the parallel SMO gradually reduces with the increase of the number of processors, by almost half using two processors and almost

three-quarters using four processors, so on and so for. The parallel SMO using one CPU processor also takes slightly more time than the sequential SMO, due to the use of MPI program. The LIBSVM requires less time than that of the sequential SMO, due to the use of the kernel cache.

Based on the obtained results, the speedup and the efficiency of the parallel SMO are calculated and, respectively, illustrated in Figs. 3 and 4. Fig. 3 shows that the speedup of the parallel SMO increases with the increase of the number of processors (up to 30 processors), demonstrating the efficiency of the parallel SMO. For this data set, the maximum value of the speedup is more than 10, corresponding to the use of 30 processors. As illustrated in Fig. 4, the efficiency of the parallel SMO decreases with the increase of the number of processors, due to the increase of the communication time.

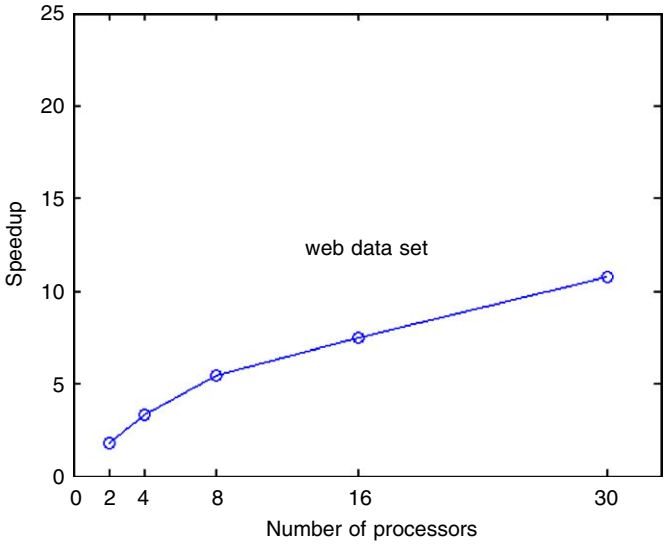

Fig. 3. The speedup of the parallel SMO on the web data set.

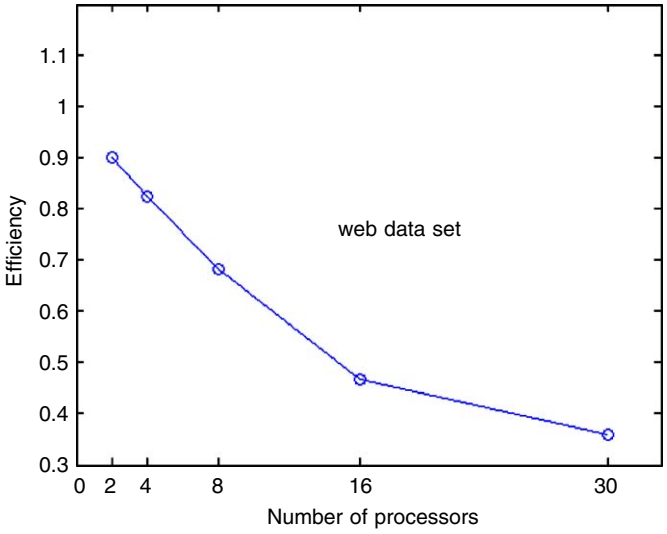

Fig. 4. The efficiency of the parallel SMO on the web data set.

The computation time in different components of the parallel SMO is reported in Table 4. The same conclusions are reached as in the adult data set. The time for I/O, initialization, and optimizing $\alpha_{I_{\text{up}}}$ and $\alpha_{I_{\text{low}}}$ is little and almost irrelevant to the number of processors. With the increase of the number of processors, the parallel time decreases, while the communication time slightly increases.

In terms of speedup and efficiency the result on the web data set is not as good as that in the adult data set. This can be analyzed as the ratio of the parallel time to the communication time in the web data set is much smaller than that of the adult data set, as illustrated in Tables 2 and 4. This also means that the advantage of using the parallel SMO is more obvious for large size problems.

4.3. MNIST data set

The MNIST handwritten digit data set is also examined in the experiment. This data set consists of 60,000 training samples and 10,000 testing samples. Each sample is composed of 576 features. This data set is available at http://www.cenparmi.concordia.ca/~people/jdong/HeroSvm/ and has also been used in Dong et al.'s work on speeding up the sequential SMO [5].

The MNIST data set is actually a 10-class classification problem. According to the "one against the rest" method, ten SVM classifiers are constructed by separating one class from the rest. In our experiment, the Gaussian kernel is used in the sequential SMO and the parallel SMO for each of ten SVM classifiers. The values of σ^2 and C are, respectively, used as 0.6 and 10, same as those used in [5].

The elapsed time with different number of processors in the sequential SMO and the parallel SMO and LIBSVM for each of 10 SVM classifiers is given in Table 5. The averaged value of the elapsed time in the 10 SVM classifiers is also listed in this table. The table shows that there is still benefit in the using of the kernel cache in LIBSVM in comparison with the sequential SMO. Figs. 5 and 6, respectively, illustrate the speedup and the efficiency of the parallel SMO. Fig. 5 shows that the speedup of the parallel SMO increases with the increase of the number of processors. The maximum values of the speedup in the 10

Table 4
The computation time in different components of the parallel SMO on the web data set

Components	Number of processors					
	1P	2P	4P	8P	16P	30P
I/O	2	2	2	2	2	2
initialization	0	0	0	0	0	0
$a_{\text{I\_up}}$, $a_{\text{I\_low}}$	0	0	0	0	0	0
b_{up}, I_{up}, b_{low}, I_{low}, *DualityGap*	0	1	1	2	3	3
F^k, b^k_{up}, I^k_{up}, b^k_{low}, I^k_{low}, *DualityGap*^{k}	183	87	43	20	9	5

Table 5
The elapsed time (s) used in the sequential SMO and the parallel SMO on the Minist data set

Class	LIBSVM	Sequential SMO	Parallel SMO					
			1P	2P	4P	8P	16P	30P
0	2931.7	3598.0	3948.8	1862.5	1006.5	483.5	283.2	210.1
1	2753.4	3717.9	3326.1	1845.3	895.5	462.5	266.7	196.1
2	5160.9	5644.2	5595.0	2781.2	1302.3	656.6	372.7	248.3
3	5738.0	6021.5	5404.2	2749.0	1330.9	703.1	399.2	272.0
4	5145.9	6044.6	6143.9	2771.7	1544.1	719.9	400.7	274.1
5	4825.6	5568.7	5529.6	2551.4	1408.7	655.1	378.6	267.6
6	3448.5	4232.7	4226.8	2099.8	973.8	491.4	294.3	194.8
7	5421.6	5788.9	5796.9	3124.4	1468.0	731.6	413.0	292.2
8	6565.8	7183.1	7243.1	3321.7	1800.3	822.4	468.5	314.7
9	7642.7	8033.8	7960.6	3645.5	1844.4	932.3	554.0	353.8
Averaged	4963.4	5583.3	5517.5	2675.2	1357.4	665.8	383.1	262.4

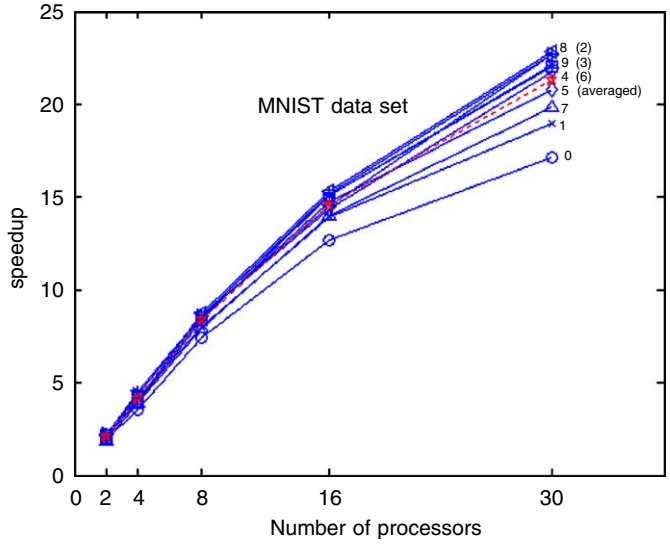

Fig. 5. The speedup of the parallel SMO on the MNIST data set.

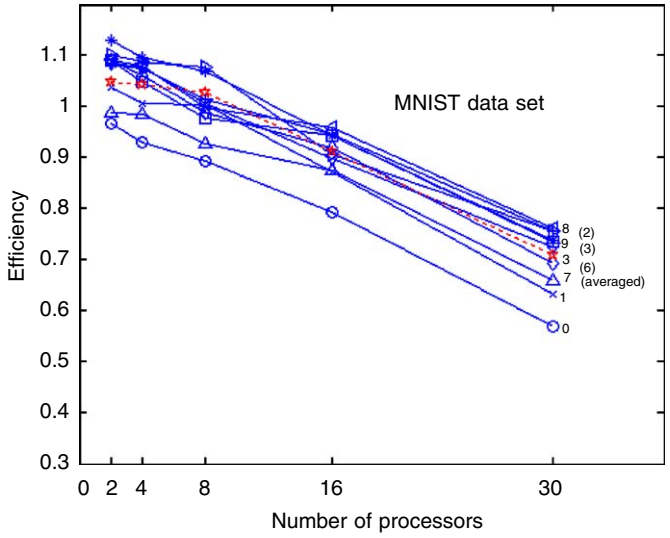

Fig. 6. The efficiency of the parallel SMO on the MNIST data set.

SVM classifiers range from 17.12 to 22.82. The averaged maximum value of speedup is equal to 21.27, corresponding to the use of 30 processors. Fig. 6 shows that the efficiency of the parallel SMO decreases with the increase of the number of processors, due to the use of more communication time.

4.4. IDEVAL data set

1999 DARPA intrusion detection evaluation data set used for the third international knowledge discovery and data mining tools competition is further examined in the experiment. The data set is available at http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html. It consists of 4,156,656 data samples. Each sample is composed of three categorical features and other 38 features with real values and one variable denoting whether the status is normal or attacked. It is actually a two-class classification problem, with one class denoting normal status and the other class denoting attacked status. In the training SVM, the three categorical features are transformed into binary features. In our experiment, the Gaussian kernel is used in the sequential SMO and the parallel SMO, with the values of σ^2 and C respectively used as 100 and 1.

The elapsed time with different number of processors in the sequential SMO and the parallel SMO and LIBSVM is given in Table 6. Same as in the other data sets, the table shows that there is benefit in the using of the kernel cache in LIBSVM in comparison with the sequential SMO. Figs. 7 and 8, respectively, illustrate the speedup and the efficiency of the parallel SMO. Fig. 7 shows that the speedup of the parallel SMO increases almost linearly with the increase of the number of processors. The maximum values of the speedup is 23.5, corresponding to the use of 30 processors.[1] Fig. 8 shows that the efficiency of the parallel SMO slightly decreases with the increase of the

[1]In the experiment, the maximal value of processors we can use is 30 due to the environment in IHPC.

Table 6
The elapsed time (s) used in the sequential SMO and the parallel SMO on the IDEVAL data set

	LIBSVM	Sequential SMO	Parallel SMO					
			1P	2P	4P	8P	16P	30P
Time(s)	17361.9	22987.3	23093.8	11549.3	5851.6	2956.0	1538.5	890.8
SVs	109294	110385	110382	110382	110382	110382	110382	110382
BSVs	92631	96480	96480	96480	96480	96480	96480	96480

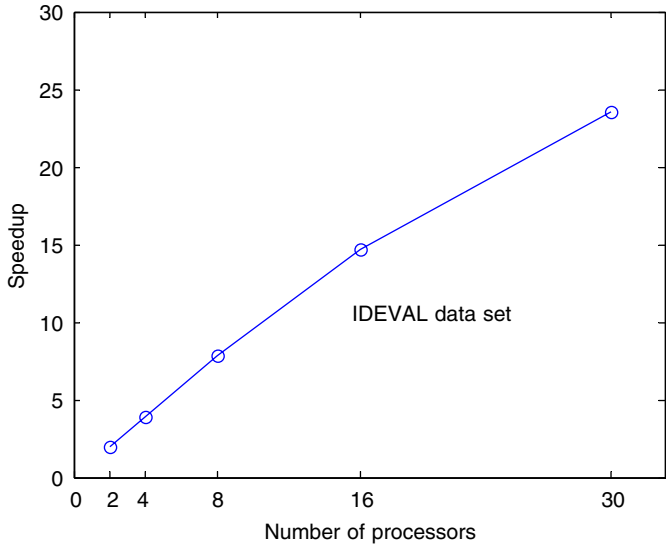

Fig. 7. The speedup of the parallel SMO on the IDEVAL data set.

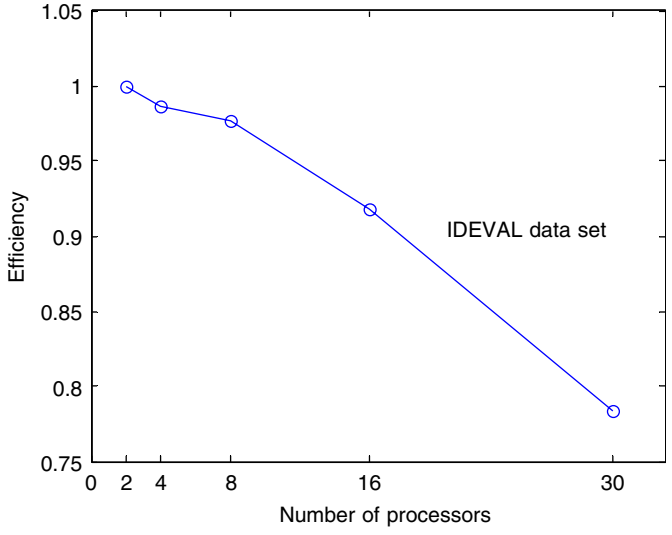

Fig. 8. The efficiency of the parallel SMO on the IDEVAL data set.

number of processors, due to the use of slightly increased communication time.

5. Conclusions

This paper proposes the parallel implementation of SMO using MPI. The parallel SMO uses multiple CPU processors to deal with the computation of SMO. By partitioning the entire training data set into smaller subsets and distributing each of the partitioned subsets into one CPU processor, the parallel SMO updates F_i array and calculates b_{up}, b_{low}, and *DualityGap* at each step in parallel using multiple CPU processors. This parallel mode is called the "SPMD" model in MPI. Experiment on three large data sets demonstrates the efficiency of the parallel SMO.

The experiment also shows that the efficiency of the parallel SMO decreases with the increase of the number of processors, as there is more communication time with the use of more processors. For this reason, the parallel SMO is more useful for large size problems.

The experiment also shows that LIBSVM with the using of the working set size as 2 is more efficient than the sequential SMO. This can be explained that the LIBSVM use the kernel cache, while the sequential and parallel SMO do not take it into account. Future work will exploit the kernel cache for further improving the current version of the parallel SMO.

In the current version of the parallel SMO, the multi-class classification problem is performed by considering one class by one class. In the future work, it is worthy to perform the multi-class classification problem in parallel by considering all the classes simultaneously for further improving the efficiency of the parallel SMO. In such an approach, it needs to develop a structural approach to consider the communication between processors. This work is very useful for the research where multiple CPU processors machine is available.

Appendix A. Pseudo-code for the parallel SMO for classification

(*Note*: If there is some process rank before the code, this means that only the processor associated with the rank executes the code. Otherwise, all the processors execute the code.)

```
n_sample = total number of training samples
p = total number of processors
local_nsample = n_sample/p
Procedure takeStep ( )
    if (i_up = = i_low&& Z1 = = Z2) return 0;
    s = y1*y2;
    if (y1 = = y2)
```

```
    gamma = alph1 + alph2;
else
    gamma = alph1-alph2;
if (s = = 1)
{
    if (y2 = = 1)
    {
        L = MAX(0,gamma-C);
        H = MIN(C, gamma);
    else
    {
        L = MAX(0,gamma-C);
        H = MIN(C, gamma);
    }
else
{
    L = MAX(0,-gamma);
    if (y2 = = 1)
        H = MIN(C, C-gamma);
else
    H = MIN(C, C-gamma);
}
if (H < = L) return 0;
K11 = kernel (X1, X1);
K22 = kernel (X2, X2);
K12 = kernel (X1, X2);
eta = 2*K12-K11-K22;
if ( eta < EPS*(K11 + K22) )
{
    a2 = alph2-(y2*(F1–F2)/eta);
    if (a2 < L)
        a2 = L;
    else if (a2 > H)
        a2 = H;
else
{
    slope = y2*(F1-F2);
    change = slope *(H-L);
    if( fabs(change) > 0 )
    {
    if (slope > 0)
        a2 = H;
    else
        a2 = L;
    else a2 = alph2;
    }
    if (y2 = = 1)
    {       if (a2 > C-EPS*C)
        a2 = C;
    else if (a2 < EPS*C)
        a2 = 0;
    else;
    else
    {if (a2 > C-EPS*C)
        a2 = C;
    else if (a2 < EPS*C)
        a2 = 0;
    else;
    }
        if( fabs(a2-alph2) < eps* (a2 + alph2 + eps)return 0;
        if (s = = 1)
            a1 = gamma-a2
        else
            a1 = gamma + a2;
        if (y1 = = 1)
        {       if (a1 > C-EPS*C)
            a1 = C;
        else if (a1 < EPS*C)
            a1 = 0;
        else;
    else
        {if (a1 > C-EPS*C)
            a1 = C;
        else if (a1 < EPS*C)
            a1 = 0;
    else;
    }
        update the value of Dual
        return 1
    Endprocedure
    Procedure ComputeDualityGap( )
    DualityGap = 0;
    loop i over local_nsample training samples
    if (y[i] = = 1)
        DualityGap + = C*MAX(0, (b-fcache[i]) );
    else
        DualityGap + = C*MAX(0, (−b + fcache[i]) );
        loop i over training samples in I_0 and I_2 and I_3
        DualityGap + = alpha[i]*y[i]*fcache[i];
    return DualityGap;
Endprocedure

Procedure main()
    processor 0:        read the first block of local_nsample
                        training data patterns from the data
                        file and save them into the matrix X
                        for i = 1 to p
                                read the i^th block of
                                local_nsample training
                                data patterns from the
                                data file and send
                                them to processor i
                        end I
    processors 1 to     receive local_nsample training data
    p:                  patterns from processor 0 and save
                        them into the matrix X
    (all the
    processors)
    initialize alpha array to all zero (for local_nsample
    training data patterns )
    initialize fcache array to the negative of y array (for
    local_nsample training data patterns )
    store the indices of positive class in I_1 and negative
    class in I_4 (for local_nsample training data patterns )
```

```
set b to zero
initialize the value of Dual to zero
DualityGap = ComputeDualityGap( ) (for
local_nsample training data patterns )
sum up DualityGap of each processor and broadcast it
to every processor
compute ( b_low, i_low ) and ( b_up, i_up) using i in I
and fcache array (for local_nsample training data
patterns )
compute global b_low and global b_up using local
b_low and local b_up of each processor
find out processor Z1 containing global b_up
find out processor Z2 containing global b_low
processor Z1:    alph1 = alpha[ i_up ];
                 y1 = y[ i_up ];
                 F1 = fcache[ i_up ];
                 X1 = X[ i_up ];
                 broadcast alph1, y1, F1, and X1 to
                 every processor
processor Z2:;   alph2 = alpha[ i_low ]
                 y2 = y[ i_low];
                 F2 = fcache[ i_low];
                 X2 = X[ i_low];
                 broadcast alph2, y2, F2, and X2 to
                 every processor

numChanged = 1;
while ( DualityGap > tol*abs(Dual) &&
numChanged! = 0 )
{
  processor 0: numChanged = takeStep( );
     broadcast numChanged to every processor
  if (numChanged = = 1)
  {
  processor 0: broadcast a1, a2, and Dual to every
  processor
  processor Z1: alph[i_up ] = a1;
     if (y1 = = 1)
     {
        if (a1 = = C)
           move i1 to I_3;
        else if (a1 = = 0 )
           move i1 to I_1;
        else
           move i1 to I_0;
        else
        {
           if (a1 = = C)
           move i1 to I_2;
        else if (a1 = = 0)
           move i1 to I_4;
        else
           move i1 to I_0;
        }
  processor Z2: alph[i_low] = a2;
     if (y2 = = 1)
     {
     if (a2 = = C)
        move i2 to I_3;
     else if ( a2 = = 0 )
        move i2 to I_1;
     else
        move i2 to I_0;
     else
     {
     if (a2 = = C)
        move i2 to I_2;
     else if (a2 = = 0)
        move i2 to I_4;
     else
        move i2 to I_0;
     }
  (all the processors)
  update fcache[i] for i in I using new Lagrange
  multipliers (for local_nsample training data patterns )
  compute (b_low, i_low) and (b_up, i_up) using i in I
  and fcache array (for local_nsample training data
  patterns )
  compute global b_low and global b_up using local
  b_low and local b_up of each processor
  find out processor Z1 containing global b_up
  find out processor Z2 containing global b_low
  processor Z1: alph1 = alpha[ i_up ];
  y1 = y[ i_up ];
  F1 = fcache[ i_up ];
  X1 = X[ i_up ];
  broadcast alph1, y1, F1, and X1 to every processor
  processor Z2: alph2 = alpha[ i_low ];
     y2 = y[ i_low];
     F2 = fcache[ i_low];
     X2 = X[ i_low];
     broadcast alph2, y2, F2, and X2 to every processor
  b = (b_low + b_up)/2
  DualityGap = ComputeDualityGap( )
  sum up DualityGap of each processor and broadcast
  it to every processor
(end of while loop)
  b = (b_low + b_up)/2
  DualityGap = ComputeDualityGap( )
  sum up DualityGap of each processor and broadcast
  it to every processor
  Primal = Dual + DualityGap
Endprocedure
```

Appendix B. Implementation of the parallel SMO for regression estimation

In regression estimation, the modified SMO is described by giving the following notation [13]. Let $I_{0a} = \{i : 0 < \alpha_i < C\}$, $I_{0b} = \{i : 0 < \alpha_i' < C\}$, $I_0 = I_{0a} \cup I_{0b}$, $I_1 = \{i : \alpha_i = 0, \alpha_i' = 0\}$, $I_2 = \{i : \alpha_i = 0, \alpha_i' = C\}$, $I_3 = \{i : \alpha_i = C, \alpha_i' = 0\}$. $I = \cup I_i, i = 0, \cdots, 3$ denotes the index of training data

patterns. $F_i = y_i - \sum_{j=1}^{l}(\alpha_j^k - \alpha'^k_j)K(X_j, X_i)$. $\tilde{F}_i = F_i - \varepsilon$, $i \in I_{0a} \cup I_1$, $\tilde{F}_i = F_i + \varepsilon$, $i \in I_{0b} \cup I_2$, $\bar{F}_i = F_i + \varepsilon$, $i \in I_{0b} \cup I_1$, $\bar{F}_i = F_i - \varepsilon$, $i \in I_{0a} \cup I_3$. $b_{\text{up}} = \min\{\tilde{F}_i : i \in I_0 \cup I_1 \cup I_3\}$, $I_{\text{up}} = \arg\min_i F_i b_{\text{low}} = \max\{\bar{F}_i : i \in I_0 \cup I_1 \cup I_3\}$, $I_{\text{low}} = \arg\max_i F_i$. $\tau = 10^{-6}$.

Sequential SMO Algorithm

Initialize $\alpha_i = 0$, $\alpha'_i = 0$, $F_i = y_i$, $i = 1, \cdots, l$
Calculate b_{up}, I_{up}, b_{low}, I_{low}
Until *satisfying KKT conditions*

(1) Optimize $\alpha_{I_{\text{up}}}(\alpha'_{I_{\text{up}}})$, $\alpha_{I_{\text{low}}}(\alpha'_{I_{\text{low}}})$
(2) Update F_i, $i = 1, \cdots, l$
(3) Calculate b_{up}, I_{up}, b_{low}, I_{low}
Repeat

Based on the same methodology for developing the parallel SMO in classification, the parallel SMO for regression estimation can be developed as follows:

Parallel SMO Algorithm:

Notation: p is the total number of processors used. $\{l^k\}_{k=1}^{p}$, $\bigcup_{k=1\to p} l^k = l$ is a subset of all the training data patterns and assigned to processor k. F_i^k, b_{up}^k, I_{up}^k, b_{low}^k, I_{low}^k, $\alpha_i^k(\alpha'^k_i)$, $i \in l^k$ denote the variables associated with processor k. $F_i^k = y_i - \sum_{j=1}^{l}(\alpha_j^k - \alpha'^k_j)K(X_j, X_i)$. $\tilde{F}_i^k = F_i^k - \varepsilon, i \in I_{0a} \cup I_1 \cup l^k$. $\tilde{F}_i^k = F_i^k + \varepsilon, i \in I_{0b} \cup I_2 \cup l^k$. $\bar{F}_\infty^k = F_i^k + \varepsilon$, $i \in I_{0b} \cup I_1 \cup l^k$. $\bar{F}_i^k = F_i^k - \varepsilon i \in I_{0a} \cup I_3 \cup l^k$. $b_{\text{up}}^k = \min\{\tilde{F}_i^k : i \in I_0 \cup I_1 \cup I_3 \cup l^k\}$, $b_{\text{low}}^k = \max\{\tilde{F}'_i : i \in I_0 \cup I_1 \cup I_3 \cup l^k\}$, $I_{\text{up}}^k = \arg\min_i F_i^k$, $I_{\text{low}}^k = \arg\max_i F_i^k$. $\alpha_{I_{\text{up}}}(\alpha'_{I_{\text{up}}})$, $\alpha_{I_{\text{low}}}(\alpha'_{I_{\text{low}}})$, b_{up}, I_{up}, b_{low}, I_{low} still denote the variables on the entire training data patterns. $b_{\text{up}} = \max\{b_{\text{up}}^k\}$, $I_{\text{up}} = \arg_{I_{\text{up}}^k} b_{\text{up}} = b_{\text{up}}^k, b_{\text{low}} = \max\{b_{\text{low}}^k\}, I_{\text{low}} = \arg_{I_{\text{low}}^k} b_{\text{low}} = b_{\text{low}}^k$. $\tau = 10^{-6}$.

Initialize $\alpha_i^k = 0$, $\alpha'^k_i = 0$, $F_i^k = y_i$, $i \in l^k$, $k = 1, \cdots, p$Calculate b_{up}^k, I_{up}^k, b_{low}^k, I_{low}^k
Obtain b_{up}, I_{up}, b_{low}, I_{low}
Until satisfying KKT conditions

(1) Optimize $\alpha_{I_{up}}(\alpha'_{I_{up}})$, $\alpha_{I_{low}}(\alpha'_{I_{low}})$
(2) Update F_i^k, $i \in l^k$
(3) Calculate b_{up}^k, I_{up}^k, b_{low}^k, I_{low}^k
(4) Obtain b_{up}, I_{up}, b_{low}, I_{low}
Repeat

References

[1] C.J.C. Burges, A tutorial on support vector machines for pattern recognition, Knowledge Discovery Data Mining 2 (2) (1998) 955–974.
[2] L.J. Cao, F.E.H. Tay, Support vector machines with adaptive parameters in financial time series forecasting, IEEE Trans. Neural Networks 14 (6) (2003) 1506–1518.
[3] C.C. Chang, C.J. Lin, LIBSVM: a library for support vector machines, available at 〈http://www.csie.ntu.edu.tw/~cjlin/libsvm/〉.
[4] R. Collobert, S. Bengio, Y. Bengio, A parallel mixture of SVMs for very large scale problems, Neural Comput. 14 (5) (2002) 1105–1114,.
[5] J.X. Dong, A. Krzyzak, C.Y. Suen, A fast SVM training algorithm, Pattern Recog. Artif. Intell., 2002, in press.
[6] J.X. Dong, A. Krzyzak, C.Y. Suen, A fast parallel optimization for training support vector machine, In: P. Perner, A. Rosenfeld, (Eds.) *Proceedings of third International Conference on Machine Learning and Data Mining*, Springer Lecture Notes in Artificial Intelligence (LNAI 2734), Leipzig, Germany, July 5–7, 2003, pp. 96–105.
[7] C.W. Hsu, C.C. Chang, C.J. Lin, A Practical Guide to Support Vector Classification. at 〈http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf〉.
[8] T. Joachims, Making large-scale support vector machine learning practical, in: B. Scholkopf, C. Burges, A. Smola (Eds.), Advances in Kernel Methods: Support Vector Machines, MIT Press, Cambridge, MA, 1998.
[9] S.S. Keerthi, S.K. Shevade, C. Bhattaacharyya, K.R.K. Murthy, Improvements to Platt's SMO algorithm for SVM classifier design, Neural Comput. 13 (2001) 637–649.
[10] E. Osuna, R. Freund, F. Girosi, An improved algorithm for support vector machines, NNSP'97: Proceedings of the IEEE Signal Processing Society Workshop, Amelia Island, USA, 1997, pp. 276–285.
[11] P.S. Pacheco, Parallel Programming with MPI, Morgan Kaufmann Publishers, San Francisco, California, 1997.
[12] J.C. Platt, Fast training of support vector machines using sequential minimal optimisation, in: B. Scholkopf, C.J.C. Burges, A.J. Smola (Eds.), Advances in Kernel Methods—Support Vector Learning, MIT Press, Cambridge, MA, 1999, pp. 185–208.
[13] S.K. Shevade, S.S. Keerthi, C. Bhattacharyya, K.R.K. Murthy, Improvements to the SMO algorithm for SVM regression, IEEE Trans. Neural Networks 11 (5) (2000) 1188–1193.
[14] V.N. Vapnik, Estimation of Dependence Based on Empirical Data, Springer, New York, 1982.
[15] V.N. Vapnik, The Nature of Statistical Learning Theory, Springer, New York, 1995.
[16] G. Zanghirati, L. Zanni, A parallel solver for large quadratic programs in training support vector machines, Parallel Comput. 29 (4) (2003) 535–551.

Dr L. J. Cao received her Ph.D. in Mechanical Engineering from the National University of Singapore in 2002. She is currently working as an associate professor in the institute of financial studies in Fudan University in Shanghai, P.R.China. Her research area focuses on support vector machines on financial applications and financial derivative pricing.

Dr. Keerthi is a Scientist at Yahoo! Research. His research has focused on the development of practical algorithms for a variety of areas, such as machine learning, robotics, computer graphics and optimal control. His current research focuses on kernel methods, in particular on the development of fast kernel methods for large scale data mining problems.

Dr. Chong-Jin Ong received Ph.D. degrees in mechanical and applied mechanics from University of Michigan, Ann Arbor 1993. He joined the National University of Singapore in 1993 and is now an Associate Professor with the Department of Mechanical Engineering. His current research interests are in the areas of machine learning and robust control theory and applications.

Uvaraj Periyathamby obtained his M.S. degree at the Department. of Mechanical Engineering in National University of Singapore in 1994. He is currently a Senior Research Engineer cum Industry Development Manager at the Institute of High Performance Computing under the Agency for Science Technology and Research in Singapore. His areas of interests include parallel computing, aerospace engineering, computational fluid dynamics and grid computing.

Dr. Xiuju Fu obtained her Ph.D. degree at the department of Mechanical Engineering from Nanyang Technological University in Singapore in 2003. Currently, she is working as a research engineer in Institute of High Performance Computing. Her research areas include: neural networks, support vector machine, genetic algorithms, data mining, classification, data dimensionality reduction, and linguistic rule extraction.

Dr H. P. Lee is currently the Deputy Executive Director (Research) of the Institute of High Performance Computing (IHPC) as well as an Associate Professor at the Department of Mechanical Engineering, National University of Singapore. He has held position as the Subdean for External Relations in the Faculty of Engineering, National University of Singapore prior to taking up the appoint at IHPC.

Dr. Lee obtained his Ph.D. and M.S. from the Department of Mechanical Engineering, Stanford University, B.A. from the University of Cambridge. His research interests are in the areas of vibration and acoustics, simulation techniques, metal forming, MEMS and data mining.