Parallel Sequential Minimal Optimization for the Training of Support Vector Machines

L. J. Cao, S. S. Keerthi, Chong-Jin Ong, J. Q. Zhang, Uvaraj Periyathamby, Xiu Ju Fu, and H. P. Lee

***Abstract*—Sequential minimal optimization (SMO) is one popular algorithm for training support vector machine (SVM), but it still requires a large amount of computation time for solving large size problems. This paper proposes one parallel implementation of SMO for training SVM. The parallel SMO is developed using message passing interface (MPI). Specifically, 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 and the Mixing National Institute of Standard and Technology (MNIST) data set when many processors are used. There are also satisfactory results on the Web data set.**

***Index Terms*—Message passing interface (MPI), parallel algorithm, sequential minimal optimization (SMO), support vector machine (SVM).**

I. 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]–[5]. 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; and 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:

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

$$\text{subject to}: \quad \sum_{i=1}^{l} \alpha_i y_i = 0, \qquad 0 \le \alpha_i \le c, \qquad i = 1, \dots, l \tag{2}$$

where $k(X_i, X_j)$ is the kernel function. The most 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 predetermined by users. After solving the QP problem (1), the following decision function is used to determine the class label for a new data pattern:

$$\text{function}\ (X) = \sum_{i=1}^{l} \alpha_i y_i k(X_i, X) + b \tag{3}$$

where b is obtained from the solution of (1).

The main problem in SVM is reduced to solving the QP problem (1), where the number of variables α_i to be 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. However, for large size problems, standard QP techniques are not useful as they require a large amount of computer memory to store the kernel matrix $\boldsymbol{K}$ as the number of elements of $\boldsymbol{K}$ 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 [6], Osuna's decomposition [7], and Joachims's SVMs$^{\text{light}}$ [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 [9] 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 scale cubically or more in the number of training data patterns. Later, Keerthi *et al.* [10], [11] 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 particularly good and used in popular SVM packages such as a library for support vector machines (LIBSVM) [12]. We will refer to this modification as the modified SMO algorithm.

Recently, there were few works on developing parallel implementation of training SVMs [13]–[16]. In [13], 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 multilayer perceptron. The experiment shows that the proposed parallel algorithm can provide more efficiency than using a single SVM. In the algorithm proposed by Dong *et al.*

Manuscript received March 29, 2005; revised January 4, 2006. This paper was supported by National Natural Science Research Fund 70501008 and Shanghai Pujiang Program.

L. J. Cao and J. Q. Zhang are with the Financial Studies, Fudan University, ShangHai 200433, P.R. China (e-mail: ljcao@fudan.edu.cn).

S. S. Keerthi and C.-J. Ong are with the Department of Mechanical Engineering, National University of Singapore, Singapore 119260, Singapore.

U. Periyathamby, X. J. Fu, and H. P. Lee are with the Institute of High Performance Computing, Singapore 117528, Singapore.

Digital Object Identifier 10.1109/TNN.2006.875989

[14], 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 more efficiency of the algorithm. Zanghirati and Zanni [15] also proposed a parallel implementation of SVM$^{\text{light}}$ 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. Huang *et al.* [16] proposed a modular network implementation for SVM. The result showed that the modular network could significantly reduce the learning time of SVM algorithms without sacrificing much generalization performance.

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) [17]. 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 ten 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 II gives an overview of the modified SMO. Section III describes the parallel SMO developed using MPI. Section IV presents the experiment indicating the efficiency of the parallel SMO. A short conclusion then follows.

II. Brief Overview of the 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 $I_4 = \{i : y_i = -1, \alpha_i = 0\}$. $I = \bigcup I_i, i = 0, \ldots, 4$ denotes the index of training data patterns. $f_i = \sum_{j=1}^{l} \alpha_j y_j k(X_j, X_i) - y_i$. $b_{\text{up}} = \min\{f_i : i \in I_0 \cup I_1 \cup I_2\}$, $I_{\text{up}} = \arg\min_i f_i$. $b_{\text{low}} = \max\{f_i : i \in I_0 \cup I_3 \cup I_4\}$, $I_{\text{low}} = \arg\max_i f_i$. $\tau = 10^{-6}$.

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 indexes are I_{up} and I_{low}

$$\alpha_2^{\text{new}} = \alpha_2^{\text{old}} - \frac{y_2\left(f_1^{\text{old}} - f_2^{\text{old}}\right)}{\eta} \tag{4}$$

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

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, $0 \le \alpha_1^{\text{new}} \le c$ and $0 \le \alpha_2^{\text{new}} \le c$.

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

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

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 (1), represented by *Dual*, is updated at each step

$$Dual^{\text{new}} = Dual^{\text{old}} - \frac{\alpha_1^{\text{new}} - \alpha_1^{\text{old}}}{y_1}\left(f_1^{\text{old}} - f_2^{\text{old}}\right) + \frac{1}{2}\eta\left(\frac{\alpha_1^{\text{new}} - \alpha_1^{\text{old}}}{y_1}\right)^2. \tag{7}$$

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

$$DualityGap = \sum_{i=0}^{l} \alpha_i y_i f_i + \sum_{i=o}^{l} \varepsilon_i$$
$$\text{where} \begin{cases} \varepsilon_i = C\max(0,\ b - f_i), & \text{if} \quad y_i = 1 \\ \varepsilon_i = C\max(0, -b + f_i), & \text{if} \quad y_i = -1. \end{cases} \tag{8}$$

A more detailed description of *Dual* and *DualityGap* can be referred to [8]. *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

Sequential SMO Algorithm

```
Initialize $\alpha_i = 0$, $f_i = -y_i$, Dual $= 0$, $i = 1, \ldots, l$.
Calculate $b_{\text{up}}$, $I_{\text{up}}$, $b_{\text{low}}$, $I_{\text{low}}$, DualityGap.
Until DualityGap $\le \tau|$ Dual $|$
1) optimize $\alpha_{I_{\text{up}}}$, $\alpha_{I_{\text{low}}}$;
2) update $f_i$, $i = 1, \ldots, l$;
3) calculate $b_{\text{up}}$, $I_{\text{up}}$, $b_{\text{low}}$, $I_{\text{low}}$, DualityGap and
update Dual.
Repeat
```

III. Parallel SMO

MPI is not a new programming language, but a library of functions that can be used in C, C++, and FORTRAN [17]. 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 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 first 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 and t_f be 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 II. 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. This program can also be executed in parallel using the SPMD mode. By running the program of (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 updates F_i array and calculates 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.

```
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 $k$. $f_i^k$, $b_{\text{up}}^k$, $I_{\text{up}}^k$, $b_{\text{low}}^k$, $I_{\text{low}}^k$, $DualityGap^k$, $\alpha_i^k$, and $i \in l^k$ denote the variables associated with processor $k$. $f_i^k = \sum_{j=1}^{l} \alpha_j y_j k(X_j, X_i) - y_i$.
$b_{\text{up}}^k = \min\{f_i^k : i \in I_0 \cup I_1 \cup I_2 \cup l^k\}$, $I_{\text{up}}^k = \arg\min_i f_i^k$.
$b_{\text{low}}^k = \max\{f_i^k : i \in I_0 \cup I_3 \cup I_4 \cup l^k\}$, $I_{\text{low}}^k = \arg\max_i f_i^k$.
$b_{\text{up}}$, $I_{\text{up}}$, $b_{\text{low}}$, $I_{\text{low}}$, and $DualityGap$ still denote the variables on the entire training data patterns. $b_{\text{up}} = \max\{b_{\text{up}}^k\}$, $I_{\text{up}} = \arg_{I_{\text{up}}^k} b_{\text{up}} = b_{\text{up}}^k$,
$b_{\text{low}} = \max\{b_{\text{low}}^k\}$, $I_{\text{low}} = \arg_{I_{\text{low}}^k} b_{\text{low}} = b_{\text{low}}^k$, and
$DualityGap = \sum_{k=1}^{p} DualityGap^k$.
Initialize $\alpha_i^k = 0$, $f_i^k = -y_i$, $Dual = 0$, $i \in l^k$, $k = 1, \ldots, p$.
Calculate $b_{\text{up}}^k$, $I_{\text{up}}^k$, $b_{\text{low}}^k$, $I_{\text{low}}^k$, and $DualityGap^k$.
Obtain $b_{\text{up}}$, $I_{\text{up}}$, $b_{\text{low}}$, $I_{\text{low}}$, and $DualityGap$.
Until $DualityGap \le \tau|Dual|$
1) optimize $\alpha_{I_{\text{up}}}$, $\alpha_{I_{\text{low}}}$;
2) update $f_i^k$, $i \in l^k$;
3) calculate $b_{\text{up}}^k$, $I_{\text{up}}^k$, $b_{\text{low}}^k$, $I_{\text{low}}^k$, and $DualityGap^k$;
4) obtain $b_{\text{up}}$, $I_{\text{up}}$, $b_{\text{low}}$, $I_{\text{low}}$, and $DualityGap$, and update $Dual$
Repeat
```

A more detailed description of the parallel SMO can be referred to the pseudocode in Appendix A.

IV. 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 SuperComputer 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.

A. Adult Data Set

The first data set used to test the parallel SMO's speed is the UCI adult data set [10]. 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 [12] is also investigated using a single processor on the experiment. The aim is to see

TABLE I
ELAPSED TIME (SECONDS) USED IN THE SEQUENTIAL SMO AND THE PARALLEL SMO AND LIBSVM ON THE ADULT DATA SET

	LIBSVM	Sequential SMO	Parallel SMO					
			1P	2P	4P	8P	16P	32P
Time(s)	1132.06	2010.1	2048.06	1026.81	521.92	275.80	145.05	93.79
SVs	8563	10591	10763	10683	10825	10853	10948	11022
BSVs	7649	8631	9023	8972	8953	9013	9038	9215

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 I, 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

$$\text{speedup} = \frac{\text{the elapsed time of the sequential SMO}}{\text{the elapsed time of the parallel SMO}} \quad (9)$$

$$\text{efficieny} = \frac{\text{speedup}}{\text{number of processors}}. \quad (10)$$

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 1/21 of that of the sequential SMO, which is very good.

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

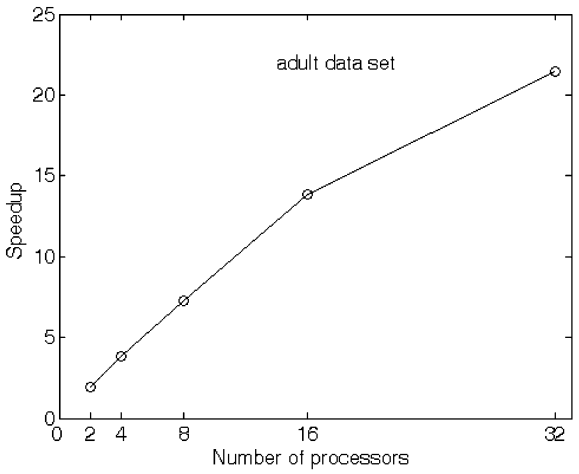

Fig. 1. Speedup of the parallel SMO on the adult data set.

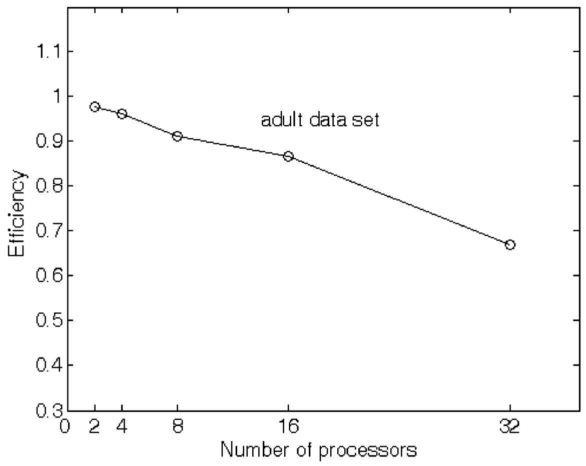

Fig. 2. Efficiency of the parallel SMO on the adult data set.

For a better understanding of the cost of various subparts in the parallel SMO, the computation time in different components (I/O; initialization; optimizing $\alpha_{I_{\text{up}}}$ and $\alpha_{I_{\text{low}}}$; updating f_i^k and calculating b_{up}^k, I_{up}^k, b_{low}^k, I_{low}^k, and *DualityGap*^{k}; and obtaining b_{up}, I_{up}, b_{low}, I_{low}, and *DualityGap*) is reported in Table II. The time for updating f_i^k and calculating b_{up}^k, I_{up}^k, b_{low}^k, I_{low}^k, and *DualityGap*^{k} is called as the parallel time as the involved calculations are done in parallel. The time for obtaining b_{up}, I_{up}, b_{low}, I_{low}, and *DualityGap* is called as the communication time as there are many processors included in the calculation. The table shows that the time for I/O, initialization, and optimizing $\alpha_{I_{\text{up}}}$ and $\alpha_{I_{\text{low}}}$ is little and irrelevant to the number of

TABLE II
COMPUATION 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^k_{up}, I^k_{up}, b^k_{low}, I^k_{low}, DualityGap^k	2041	1017	507	261	129	66

TABLE III
ELAPSED TIME USED IN THE SEQUENTIAL SMO AND THE PARALLEL SMO AND LIBSVM ON THE WEB DATA SET

	LIBSV M	Sequential SMO	Parallel SMO					
			1P	2P	4P	8P	16P	30P
Time(s)	104.27	172.75	191.33	95.70	52.42	31.59	23.11	16.0
SVs	528	672	703	712	726	752	805	817
BSVs	493	658	685	687	694	703	718	742

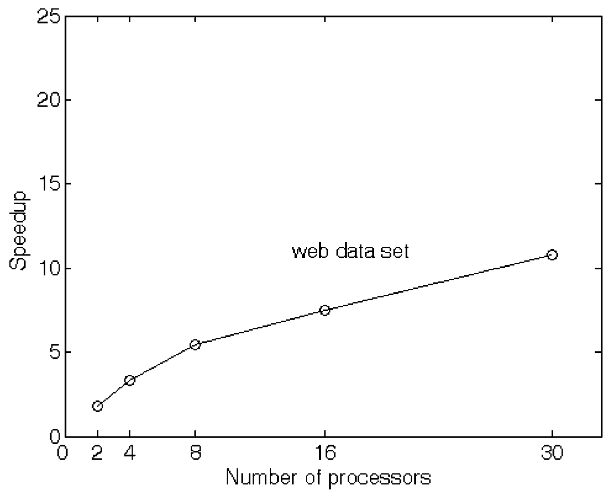

Fig. 3. Speedup of the parallel SMO on the web data set.

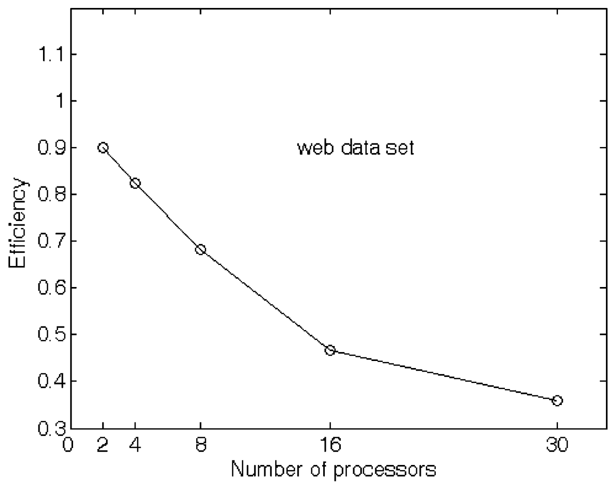

Fig. 4. Efficiency of the parallel SMO on the web data set.

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 $b^k_{\rm up}$, $I^k_{\rm up}$, $b^k_{\rm low}$, $I^k_{\rm low}$, and *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.

B. Web Data Set

The web data set is examined in the second experiment [10]. 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 III, 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

TABLE IV
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_{I_up}, a_{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 V
ELAPSED TIME USED IN THE SEQUENTIAL SMO AND THE PARALLEL SMO AND LIBSVM ON THE MNIST DATA SET

Class	LIBSVM	Sequential SMO	Parallel SMO					
			1P	2P	4P	8P	16P	30P
0	2931.668	3597.97	3948.83	1862.49	1006.46	483.51	283.19	210.10
1	2753.418	3717.91	3326.05	1845.33	895.45	462.50	266.70	196.09
2	5160.932	5644.19	5595.01	2781.18	1302.27	656.56	372.72	248.32
3	5737.956	6021.50	5404.18	2749.00	1330.94	703.06	399.22	271.97
4	5145.859	6044.60	6143.85	2771.65	1544.05	719.86	400.72	274.08
5	4825.642	5568.70	5529.62	2551.38	1408.74	655.09	378.62	267.57
6	3448.498	4232.65	4226.76	2099.81	973.81	491.43	294.33	194.78
7	5421.564	5788.88	5796.86	3124.36	1467.97	731.57	412.99	292.19
8	6565.783	7183.05	7243.13	3321.72	1800.28	822.35	468.53	314.70
9	7642.706	8033.80	7960.56	3645.48	1844.40	932.33	554.03	353.78
Averaged	4963.403	5583.325	5517.485	2675.24	1357.437	665.826	383.105	262.358

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.

The computation time in different components of the parallel SMO is reported in Table IV. The same conclusions are reached as in the adult data set. The time for I/O, initialization, and optimizing $\alpha_{I_{\rm up}}$ and $\alpha_{I_{\rm low}}$ 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 II and IV. This also means that the advantage of using the parallel SMO is more obvious for large-size problems.

C. *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's *et al.* work on speeding up the sequential SMO [18].

The MNIST data set is actually a ten-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

TABLE VI
NUMBER OF CONVERGED SUPPORT VECTORS AND BOUNDED SUPPORT VECTORS IN THE SEQUENTIAL SMO AND THE PARALLEL SMO AND LIBSVM ON THE MNIST DATA SET

Class	LIBSVM #SVs #BSVs	Sequential SMO #SVs #BSVs	Parallel SMO					
			1P #SVs #BSVs	2P #SVs #BSVs	4P #SVs #BSVs	8P #SVs #BSVs	16P #SVs #BSVs	30P #SVs #BSVs
0	1871 1807	2021 1865	2130 2011	2048 1929	2060 1946	2073 1947	2074 1958	2095 1996
1	1982 1862	2104 1928	2167 2072	2140 1968	2170 1981	2179 1934	2131 1988	2190 2039
2	2108 2010	2811 2084	2338 2297	2547 2209	2571 2132	2479 2163	2597 2242	2314 2409
3	2976 1989	3092 2607	2403 2046	3316 2460	3073 2594	3119 2108	303 2331	3109 2322
4	2126 1934	2374 2463	2317 2221	2384 2272	2674 2129	2565 2104	2614 2021	2863 2100
5	2106 2213	2356 2093	3022 2143	2799 2191	2628 2615	3128 2055	2997 2544	3190 2651
6	2483 2199	2551 2236	2891 2179	2650 2165	2718 2018	2716 2215	3036 2117	3179 2110
7	2265 2008	2807 2313	2985 2187	2752 2178	2805 2204	2803 2214	3027 2126	3287 2121
8	2146 2008	2813 2102	3035 2163	2741 2159	2852 2205	2879 2220	3065 2203	3242 2259
9	2317 2011	2887 2112	3003 2218	2879 2173	2896 2213	2982 2225	3201 2216	3384 2255
Averaged	2248 2004	2572 2180	2630 2154	2627 2170	2686 2204	2744 2119	2719 2175	2788 2226

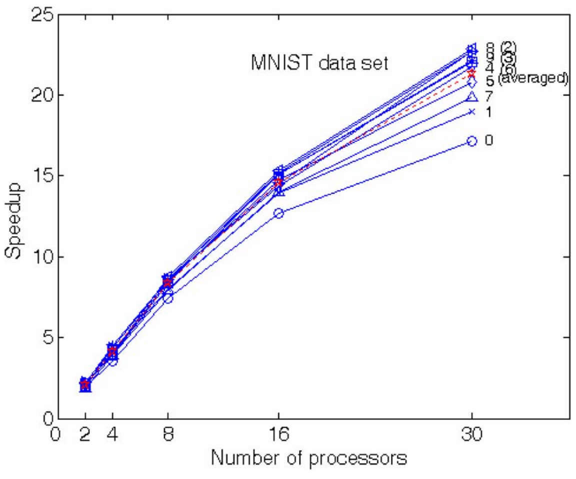

Fig. 5. Speedup of the parallel SMO on the MNIST data set. (Color version available online at http://ieeexplore.ieee.org.)

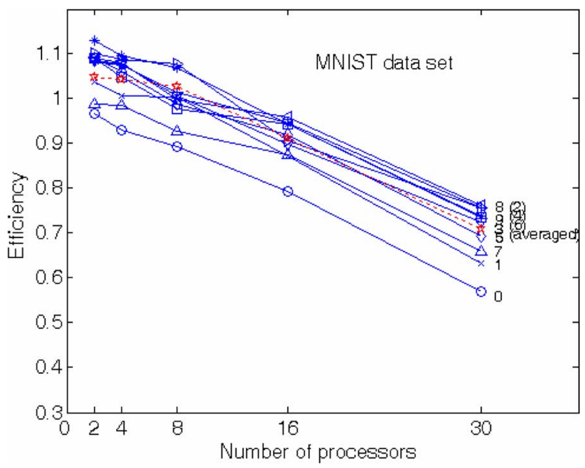

Fig. 6. Efficiency of the parallel SMO on the MNIST data set. (Color version available online at http://ieeexplore.ieee.org.)

classifiers. The values of σ^2 and c are, respectively, used as 0.6 and 10, same as those used in [14].

The elapsed time with different number of processors in the sequential SMO and the parallel SMO and LIBSVM for each of ten SVM classifiers is given in Table V. The number of converged support vectors and bounded support vectors is described in Table VI. The averaged value of the elapsed time in the ten 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 illustrate the speedup and the efficiency of the parallel SMO, respectively. 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 ten 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.

V. CONCLUSION

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 $b_{\rm up}$, $b_{\rm low}$, 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 uses the kernel cache, while the sequential and parallel SMO does 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 multiclass classification problem is performed by considering one class by one class. In the future work, it is worthy to perform the multiclass classification problem in parallel by considering all the classes simultaneously for further improving the efficiency of the parallel SMO. In such an approach, there is a need 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. Future work also needs to extend the parallel SMO from classification for regression estimation by implementing the same methodology for SVM regressor.

APPENDIX
PSEUDOCODE FOR THE PARALLEL SMO

(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 ($y2 == 1$)
{
$L = \mathrm{MAX}(0, \mathrm{gamma} - C)$;
$H = \mathrm{MIN}(C, \mathrm{gamma})$;
} else
{
$L = \mathrm{MAX}(0, \mathrm{gamma} - C)$;
$H = \mathrm{MIN}(C, \mathrm{gamma})$;
}
} else
{
$L = \mathrm{MAX}(0, -\mathrm{gamma})$;
if ($y2 == 1$)
$H = \mathrm{MIN}(C, C - \mathrm{gamma})$;
else
$H = \mathrm{MIN}(C, C - \mathrm{gamma})$;
}
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 ($a2 > C - \mathrm{EPS} * C$)
$a2 = C$;
else if ($a2 < \mathrm{EPS} * C$)
$a2 = 0$;
else;
} else
{if ($a2 > C - \mathrm{EPS} * C$)
$a2 = C$;
else if ($a2 < \mathrm{EPS} * C$)
```

```
$a2 = 0$;
else;
}
if (fabs $(a2-\text{alph2}) < \text{eps}*(a2+\text{alph2}+\text{eps})$ return 0;
if $(s == 1)$
$a1 = \text{gamma} - a2$
else
$a1 = \text{gamma} + a2$;
if $(y1 == 1)$
{if $(a1 > \text{C} - \text{EPS} * C)$
$a1 = C$;
else if $(a1 < \text{EPS} * C)$
$a1 = 0$;
else;
} else
{ if $(a1 > C - \text{EPS} * C)$
$a1 = C$;
else if $(a1 < \text{EPS} * C)$
$a1 = 0$;
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* $+= C * \text{MAX}(0, (b - \text{fcache}[i]))$;
else
*DualityGap* $+= C * \text{MAX}(0, (-b + \text{fcache}[i]))$;
loop $i$ over training samples in I_0 and I_2 and I_3
*DualityGap* $+= \text{alpha}[i] * y[i] * \text{fcache}[i]$;
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 p**: receive local_nsample training data 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** $Z1$: $\text{alph1} = \text{alpha}[\text{i\_up}]$;
$y1 = y[\text{i\_up}]$;
$F1 = \text{fcache}[\text{i\_up}]$;
$X1 = X[\text{i\_up}]$;
broadcast alph1, $y1$, $F1$, and $X1$ to every processor
**processor** $Z2$: $\text{alph2} = \text{alpha}[\text{i\_low}]$;
$y2 = y[\text{i\_low}]$;
$F2 = \text{fcache}[\text{i\_low}]$;
$X2 = X[\text{i\_low}]$;
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** $Z1$: $\text{alph}[\text{i\_up}] = a1$;
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
$b = (\text{blow} + \text{bup})/2$
DualityGap = ComputeDualityGap()
sum up DualityGap of each processor and
broadcast it to every processor
} (end of while loop)
$b = (\text{blow} + \text{bup})/2$
DualityGap = ComputeDualityGap()
sum up DualityGap of each processor and
broadcast it to every processor
Primal = Dual + DualityGap
Endprocedure
```

References

[1] V. N. Vapnik, *The Nature of Statistical Learning Theory*. New York: Springer-Verlag, 1995.

[2] C. J. C. Burges, "A tutorial on support vector machines for pattern recognition," *Knowledge Discovery and Data Mining*, vol. 2, no. 2, pp. 955–974, 1998.

[3] L. J. Cao and F. E. H. Tay, "Support vector machines with adaptive parameters in financial time series forecasting," *IEEE Trans. Neural Netw.*, vol. 14, no. 6, pp. 1506–1518, Nov. 2003.

[4] S. Gutta, R. J. Jeffrey, P. Jonathon, and H. Wechsler, "Mixture of experts for classification of gender, ethnic origin, and pose of human faces," *IEEE Trans. Neural Netw.*, vol. 11, no. 4, pp. 948–960, Jul. 2000.

[5] K. Ikeda, "Effects of kernel function on nu support vector machines in extreme cases," *IEEE Trans. Neural Netw.*, vol. 17, no. 1, pp. 1–9, Jan. 2006.

[6] V. N. Vapnik, *Estimation of Dependence Based on Empirical Data*. New York: Springer-Verlag, 1982.

[7] E. Osuna, R. Freund, and F. Girosi, "An improved algorithm for support vector machines," in *Proc. IEEE Signal Processing Society Workshop (NNSP'97)*, Amelia Island, USA, 1997, pp. 276–285.

[8] T. Joachims, , B. Scholkopf, C. Burges, and A. Smola, Eds., "Making large-scale support vector machine learning practical," in *Advances in Kernel Methods: Support Vector Machines*. Cambridge, MA: MIT Press, 1998.

[9] J. C. Platt, , B. Scholkopf, C. Burges, and A. J. Smola, Eds., "Fast training of support vector machines using sequential minimal optimisation," in *Advances in Kernel Methods—Support Vector Learning*. Cambridge, MA: MIT Press, 1999, pp. 185–208.

[10] S. S. Keerthi, S. K. Shevade, C. Bhattaacharyya, and K. R. K. Murthy, "Improvements to Platt's SMO algorithm for SVM classifier design," *Neural Comput.*, vol. 13, pp. 637–649, 2001.

[11] S. K. Shevade, S. S. Keerthi, C. Bhattacharyya, and K. R. K. Murthy, "Improvements to the SMO algorithm for SVM regression," *IEEE Trans. Neural Netw.*, vol. 11, no. 5, pp. 1188–1193, Sep. 2000.

[12] C. C. Chang and C. J. Lin, LIBSVM: A Library for Support Vector Machines [Online]. Available: http://www.csie.ntu.edu.tw/~cjlin/libsvm/

[13] R. Collobert, S. Bengio, and Y. Bengio, "A parallel mixture of SVMS for very large scale problems," *Neural Comput.*, vol. 14, no. 5, pp. 1105–1114, 2002.

[14] J. X. Dong, A. Krzyzak, and C. Y. Suen, P. Perner and A. Rosenfeld, Eds., "A fast parallel optimization for training support vector machine," in *Proc. 3rd Int. Conf. Machine Learning Data Mining*, Leipzig, Germany, Jul. 5–7, 2003, vol. LNAI 2734, Springer Lecture Notes in Artificial Intelligence, pp. 96–105.

[15] G. Zanghirati and L. Zanni, "A parallel solver for large quadratic programs in training support vector machines," *Parallel Comput.*, vol. 29, no. 4, pp. 535–551, 2003.

[16] B. H. Guang, K. Z. Mao, C. K. Siew, and D. S. Huang, "Fast modular network implementation for support vector machines," *IEEE Trans. Neural Netw.*, vol. 16, no. 6, pp. 1651–1663, Nov. 2005.
[17] P. S. Pacheco, *Parallel Programming With MPI*. San Francisco, CA: Morgan Kaufmann, 1997.
[18] J. X. Dong, A. Krzyzak, and C. Y. Suen, "A fast SVM training algorithm," *Pattern Recognit. Artif. Intell.*, 2006, accepted for publication.

L. J. Cao received the Ph.D. degree in mechanical engineering from the National University of Singapore, Singapore, in 2002.

She is currently working as the Associate Professor in the Institute of Financial Studies, Fudan University, Shanghai, China. Her research area focuses on support vector machines, financial applications, and financial derivative pricing.

S. S. Keerthi, photograph and biography not available at the time of publication.

Chong-Jin Ong received the Ph.D. degree in mechanical and applied mechanics from University of Michigan, Ann Arbor 1993.

He joined the National University of Singapore, Singapore, in 1993 and he 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.

J. Q. Zhang received the Ph.D. degree from the Department of Mathematics, ShangDong University, P.R. China, in 2000.

He is currently with the Institute of Financial Studies, Fudan University, Shanghai, China. His research area focuses on mathematical finance.

Uvaraj Periyathamby received the M.S. degree from the Department of Mechanical Engineering, National University of Singapore, Singapore, in 1994.

He is a Senior Research Engineer and Industry Development Manager at the Institute of High Performance Computing, Agency for Science Technology and Research, Singapore. His areas of interests include parallel computing, aerospace engineering, computational fluid dynamics, and grid computing.

Xiu Ju Fu received the Ph.D. degree from the Department of Mechanical Engineering, Nanyang Technological University, Singapore, in 2003.

Currently, she is working as a Research Engineer at the Institute of High Performance Computing, Agency for Science Technology and Research, Singapore. Her research areas include neural networks, support vector machine, genetic algorithms, data mining, classification, data dimensionality reduction, and linguistic rule extraction.

H. P. Lee, photograph and biography not available at the time of publication.