Parallel support vector machines on multi-core and multiprocessor systems

H. X. Zhao, F. Magoules

▶ To cite this version:

H. X. Zhao, F. Magoules. Parallel support vector machines on multi-core and multiprocessor systems. 11th International Conference on Artificial Intelligence and Applications (AIA 2011), Feb 2011, Innsbruck, Austria. 10.2316/P.2011.717-056 . hal-00617933

HAL Id: hal-00617933
https://hal.science/hal-00617933

Submitted on 31 Aug 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Parallel Support Vector Machines on Multi-core and Multiprocessor Systems

Hai-xiang Zhao, Frédéric Magoulès
Applied Mathematics and Systems Laboratory, Ecole Centrale Paris
Grande Voie des Vignes, 92295 Châtenay-Malabry Cedex, France
Email: haixiang.zhao@ecp.fr, frederic.magoules@hotmail.com

***Abstract*—This paper proposes a new and efficient parallel implementation of support vector machines based on decomposition method for handling large scale datasets. The parallelizing is performed on the most time-and-memory consuming work of training, i.e., to update the vector f. The inner problems are dealt by sequential minimal optimization solver. Since the underlying parallelism is realized by the shared memory version of Map-Reduce paradigm, our system is easy to build and particularly suitable to apply to multi-core and multiprocessor systems. Experimental results show that on most of the tested datasets, our system offers higher than four-fold increase in speed compared to Libsvm, and it is also far more efficient than the MPI implementation Pisvm.**

***Keywords*-support vector machine; parallel; multi-core; Map-Reduce**

I. INTRODUCTION

Support vector machine (SVM) is a popular supervised learning method of solving classification and regression problems [1]. It shows robust generalization ability in numerous applications such as image processing, text mining, neural analysis and energy efficiency modeling [2], [3]. The training of SVM is essentially a quadratic optimization problem which is both time and memory costly while running on computers, making it a challenge to apply SVM on large scale problems. Several optimizing or heuristic methods have been proposed to accelerate the training and reduce the memory occupation, such as shrinking, chunking [4], kernel caching [5], approximation of kernel matrix [6]. In addition, certain scalable solvers can be used such as Sequential Minimal Optimization (SMO) [7], mixture SVMs [8], primal estimated sub-gradient solver [9]. Despite these efforts, however, a more sophisticated and satisfactory solution is still expected for this challenging research problem.

Thanks to the modern chip manufacturing, we are entering the multi-core era. Computers with multi-cores or multiprocessors are becoming more available and affordable. This paper aims to investigate and demonstrate how SVM — a popular machine learning algorithm can benefit from this modern platform. A new parallel SVM that is particularly suitable to shared memory system is proposed. Decomposition method, caching and SMO inner quadratic problem solver are composed in the implementation as the key techniques. For the purpose of achieving easy implementation without sacrificing performance, the state-of-the-art parallel programming framework Map-Reduce is chosen to perform the underlying parallelism. The system is therefore called MRPsvm, which stands for ”Map-Reduce parallel SVM”. Comparative system analysis and experimental results on benchmark datasets show significant memory saving and overwhelming speed increase in our system.

The following sections are organized as follows. Section II introduces the basic problem of SVM training and how the decomposition method can solve the problem. Section III explains the some key points of system implementation. Section IV describes the related work. Section V presents the numerical experiments on benchmark datasets and corresponding results. Conclusions are drawn in section VI.

II. SUPPORT VECTOR MACHINE

SVM for classification purpose aims at finding a hyperplane to separate two classes with maximum margin. Let x_i denotes the ith training sample, y_i denotes the corresponding label with the value either -1 or 1, $i = 1, 2, ..., l$. l is the total number of training samples. The dual form of the SVM can be written as the following convex quadratic function.

$$\min_{\alpha} \quad \frac{1}{2}\alpha^T Q\alpha - \sum_{i=1}^{l} \alpha_i \tag{1}$$

subject to

$$y^T\alpha = 0 \tag{2}$$

$$0 \leq \alpha_i \leq C \quad \forall i = 1, 2, ..., l \tag{3}$$

Where Q is a l by l positive semi-definite matrix. Each element of Q has the form $Q_{ij} = y_i y_j K(x_i, x_j)$ where $K(x_i, x_j)$ is called kernel function which can be substituted by any Mercer kernel. $C > 0$ is the upper bound used to trade off between classifier performance on training data and its generalization ability. α is a vector of l variables, where each element α_i is the weight of the corresponding training sample (x_i, y_i). The objective of the problem is to find the solution of α which makes (1) minimized and the constraints (2) (3) fulfilled. After we find out the optimal α, we can use the following decision function to predict the labels with the new input of x:

$$sgn\left(\sum_{i=1}^{l} y_i\alpha_i K(x_i,x)-b\right)$$

where b is a constant value which can be easily calculated in the training step, and

$$sgn(u)=\begin{cases}-1, & \text{for } u<0\\ 1, & \text{for } u>0\end{cases}$$

A. Decomposition method

Since the size of the kernel matrix Q is the square of the number of samples, i.e., l^2, so that it is difficult to store the whole matrix in memory when l is very big. Osuna et al. [4] proposed a method to decompose the problem into smaller tasks. In each task, certain parts of α are chosen to be optimized, while the rest of α remains in constant value. The selected part is called working set. Then the program repeats the select-optimize process until global optimality conditions are satisfied. Let B denote the working set with n variables and N denote the non-working set with $(l-n)$ variables. Then, α, y and Q can be correspondingly written as:

$$\alpha=\begin{vmatrix}\alpha_B\\ \alpha_N\end{vmatrix},\quad y=\begin{vmatrix}y_B\\ y_N\end{vmatrix},\quad Q=\begin{vmatrix}Q_{BB} & Q_{BN}\\ Q_{NB} & Q_{NN}\end{vmatrix}$$

Accordingly, the small task can be written as:

$$\begin{aligned}\min\quad & \frac{1}{2}\alpha_B^T Q_{BB}\alpha_B-\alpha_B^T(1-Q_{BN}\alpha_N)\\ & +\frac{1}{2}\alpha_N^T Q_{NN}\alpha_N-\alpha_N^T\end{aligned}\tag{4}$$

subject to

$$\alpha_B^T y_B+\alpha_N^T y_N=0 \tag{5}$$

$$0\le\alpha_B\le C \tag{6}$$

Since $(\frac{1}{2}\alpha_N^T Q_{NN}\alpha_N-\alpha_N^T)$ in (4) remains constant in an iteration, this term can be omitted while calculating. So that function (4) has essentially the same form as function (1). The advantage of this problem decomposition method is that the newly generated task is small enough to be solved by most off-the-shelf methods. In our work, we choose SMO as the inner small task solver due to its relative simplicity, yet high performance characteristics. Interestingly, SMO is itself an extreme case of the decomposition method where the working set contains only two variables. This kind of binary sub-problem can be easily solved analytically [7], [10]. As stated in [5], the solution of the sub-problem (4) is strictly feasible towards the optimum solution of global problem (1). This feature guarantees the global convergence of this decomposition method.

B. Optimality condition

When to stop the optimizing process is evaluated by the Karush-Kuhn-Tucker condition iteratively. The evaluating procedure can be summarized as follows. Firstly, we classify the training samples into two categories:

$$I_{up}(\alpha)=\{t|\alpha_t<C,y_t=1 \ or \ \alpha_t>0,y_t=-1\}$$
$$I_{low}(\alpha)=\{t|\alpha_t<C,y_t=-1 \ or \ \alpha_t>0,y_t=1\}$$

Secondly we search two extreme values $m(\alpha)$ and $M(\alpha)$ by

$$m(\alpha)=\max_{i\in I_{up}(\alpha)}-y_i f_i$$

$$M(\alpha)=\min_{i\in I_{low}(\alpha)}-y_i f_i$$

where $f_i=\sum_{j=1}^{l}\alpha_j Q_{ij}-1$ is the gradient of (1). Since f_i should be calculated for all l variables and Q_{ij} may not be stored in the memory, the calculation of f_i becomes the main task of the whole optimizing work.

After we find out $m(\alpha)$ and $M(\alpha)$, we define the stop condition as:

$$m(\alpha)-M(\alpha)\le\epsilon \tag{7}$$

where ϵ is a pre-defined value.

C. Working set selection

The selection of working set B directly influences the speed of convergence. For inner SMO solver, the maximal violating pair is selected to be the binary working set according to the second order information [11]. We do not state here how inner SMO solver works since it has been discussed in detail in [7] [10]. For the selection of working set B, we simply consider the first order information and select, in some sense, the maximal violating pairs as proposed by Zanni et al. [12]. Suppose the required size of B is n, we choose q ($q<n$) variables from α by sequentially selecting pair of variables which satisfy (7). The remaining $(n-q)$ variables are chosen as those who entered B in the last iteration but not yet selected in current B. The selection of these $(n-q)$ variables follows the sequence: free variables which satisfy $0<\alpha_i<C$, lower bound variables which are equal to 0, upper bound variables which have the value C. The reason for putting restraint on the number of new variables entering the working set is to avoid frequent entering-leaving of certain variables. Otherwise, the speed of convergence would considerably slow down [12].

After the working set is optimized, f is updated by:

$$f_i^*=f_i+\sum_{j\in B}\Delta\alpha_j Q_{ij}\qquad i=1,2,...,l \tag{8}$$

where $\Delta\alpha_j$ is the newly optimized α_j minus the old α_j. This procedure is crucial as it prepares f for the next

iteration to do optimality condition evaluation and working set selection. In fact, this is the most time-consuming step in SVM training due to the computation of Q_{ij} . Therefore, the main parallelizing work of MRPsvm is based on the updating of f. The whole parallel decomposition method is briefly outlined in algorithm 1.

Algorithm 1 Parallel decomposition solver
INPUT: data set $(x_i, y_i), \forall i \in \{1, ..., l\}$, n, ϵ
INITIALIZE: $\alpha_i = 0$, $f_i = -1$, $\forall i \in \{1, ..., l\}$
CALCULATE: I_{up}, I_{low}, $m(\alpha)$, $M(\alpha)$
REPEAT
select working set B, $\mid B \mid= n$
optimize α_i, $\forall i \in B$, by SMO solver
update f_i, $\forall i \in \{1, ..., l\}$ in parallel
calculate I_{up}, I_{low}, $m(\alpha)$, $M(\alpha)$
UNTIL $m(\alpha) - M(\alpha) \leq \epsilon$

III. System implementation

In this section, we introduce some key points of our system implementation, i.e., why and how Map-Reduce is used to do the parallelism, caching technique and data representation.

A. Map-Reduce on multi-core system

Map-Reduce is a parallel programming framework originally proposed by Google [13]. It can help us extract parallelism of computations on large data sets by taking advantage of distributed systems. It allows users to write code in a functional style: map computations on separated data, generate intermediate key/value pairs and then reduce the summation of intermediate values assigned to the same key. The runtime system automatically handles low-level mapping, scheduling, parallel processing and fault tolerance. It is a simple but very useful framework.

Except the plenty researches of Map-Reduce in distributed environments, such as [14]–[17], Chu et al. [18] has attempted to use this technology to develop a general programming framework on multi-core systems for machine learning applications. For SVM, they have parallelized linear SVM with primal problem. To conduct further development, our work attempts to develop a parallel SVM for general classification problem, and the solved SVM problem is in dual form.

According to the mechanism of Map-Reduce, this framework is naturally suitable to deal with our problem (8) in parallel. Phoenix designed by Ranger et al. [19] implements Map-Reduce on shared memory systems. It supplies a common API for users to easily parallelize their applications without conducting concurrency management. The map and reduce tasks are performed in threads. An efficient integrated runtime system is supposed to handle the parallelism, resource management and fault recovery by itself. Their experimental results show that this system works just as efficient as pure Pthreads implementation. Therefore, we choose this system as the underlying Map-Reduce handler to implement our MRPsvm.

The parallel computation works as follows. Variables in working set B is divided into several parts, the calculation of f^* is also divided into several parts in the same manner as for B. Each part is then assigned to a map process. The major work of each map process is thus to calculate Q_{ij} if they are not saved in the memory, and then calculate $\sum_j \Delta\alpha_j Q_{ij}$ for all $i = 1, 2, ..., l$. After the distributed calculations of these maps, final f^* is summed up by the reduction. Figure 1 explicitly depicts the detailed problem and the partitioning manner. It is obvious that the problem is divided on kernel columns. $(j_k, k = 1, ..., n)$ are the variable index of working set in kernel matrix, which gives the kth variable in B with its index in Q as j_k. In practice, it is not necessary to update f on all the n variables since some of them are so marginal that can be omitted.

B. Caching technique

As stated in the previous sections, for large scale problems, kernel matrix Q is too large to be stored in memory, and the calculation of kernel elements Q_{ij} is the dominant work that slows down the training. It is an effective technique to cache the kernel elements in memory as much as possible. MRPsvm maintains a fix-sized cache which stores recently accessed or generated kernel columns. The cache replacement policy is a simple least-recent-use strategy, as same as that of Libsvm. Only the column currently needed but not hit in the cache will be calculated. All parallel maps share an unique copy of cache in the shared memory. In consequence, the operation of inserting a new column into the cache performed by whichever map should be synchronized.

For inner SMO solver, the kernel matrix size is dependent on the size of working set B which is normally set as 1024 according to the knowledge of experience, it is practical to pre-compute and cache the full version of this small kernel matrix.

C. Sparse data representation

To reduce the storage requirements, the sample vectors x_i are stored by sparse representation. When calculating a kernel column $(Q_{ij}, i = 1, 2, ..., l)$, we need to unroll the jth sample vector to dense format and then calculate the dot products of this vector with the other l sample vectors.

IV. Related work

Since the essential convex quadratic problem can be solved by several methods, different kinds of parallel SVMs

map.1 … map.r

$$f_1^* = f_1 + \Delta\alpha_{j_1} Q_{1j_1} + \Delta\alpha_{j_2} Q_{1j_2} + \cdots\cdots + \Delta\alpha_{j_k} Q_{1j_k} + \cdots\cdots + \Delta\alpha_{j_m} Q_{1j_m} + \Delta\alpha_{j_n} Q_{1j_n}$$

$$f_2^* = f_2 + \Delta\alpha_{j_1} Q_{2j_1} + \Delta\alpha_{j_2} Q_{2j_2} + \cdots\cdots + \Delta\alpha_{j_k} Q_{2j_k} + \cdots\cdots + \Delta\alpha_{j_m} Q_{2j_m} + \Delta\alpha_{j_n} Q_{2j_n}$$

$$\vdots$$

$$f_i^* = f_i + \Delta\alpha_{j_1} Q_{ij_1} + \Delta\alpha_{j_2} Q_{ij_2} + \cdots\cdots + \Delta\alpha_{j_k} Q_{ij_k} + \cdots\cdots + \Delta\alpha_{j_m} Q_{ij_m} + \Delta\alpha_{j_n} Q_{ij_n}$$

$$\vdots$$

$$f_l^* = f_l + \Delta\alpha_{j_1} Q_{lj_1} + \Delta\alpha_{j_2} Q_{lj_2} + \cdots\cdots + \Delta\alpha_{j_k} Q_{lj_k} + \cdots\cdots + \Delta\alpha_{j_m} Q_{lj_m} + \Delta\alpha_{j_n} Q_{lj_n}$$

Figure 1. Problem partitioning and tasks assignment to mappers for parallel computing.

were proposed according to the particular quadratic problem solver. Based on stochastic gradient descent method, P-packSVM optimizes SVM training directly on the primal form for arbitrary kernels [20]. Very high efficiency and competitive accuracy have been achieved by the parallel implementation. Psvm proposed in [6] is based on interior point solver. It approximates the kernel matrix by incomplete Cholesky factorization. Memory requirement is reduced and scalable performance has been achieved. Bickson et al. [21] solve the problem by Gaussian belief propagation, a method from complex system domain. The parallel solver brings competitive speedup on large scale problems.

The decomposition method attracts more attentions than the above solvers. Graf et al. [22] train several SVMs on small data partitions, then they aggregate support vectors from two pair SVMs to form new training samples on which another training is performed. The aggregation is repeated until only one SVM remains. The similar idea is used by Dong et al. [23], in this work, sub-SVMs are performed on block diagonal matrices which are the approximation to the original kernel matrix. Consequently, nonsupport vectors are removed when dealing these sub-problems. Zanni et al. [12] parallelize SVM-light with improved working set selection and inner quadratic problem solver. Hazan et al. [24] propose a parallel decomposition solver using Fenchel Duality.

Similar to our implementation, Cao et al. [25] and Catanzaro et al. [26] parallelized SMO solver for training SVM. Both work mainly focuses on updating gradient for Karush-Kuhn-Tucker condition evaluation and the working set selection. The difference between them is on the implementation details and the programming models. Specifically speaking, the first work is conducted by using MPI on clusters while the second one by Map-Reduce threads on modern GPU platform. In our work, we also adopt SMO algorithm. But we use it as the inner quadratic problem solver without any parallel computation, instead, we perform the parallelization on external decomposition procedure. The main advantage of our coarse-grained parallelism is that it can significantly reduce the burden of overheads since the number of iterations in global decomposition procedure (where $n \gg 2$) is extremely smaller than that of pure SMO algorithm (where $n = 2$).

Pisvm [27] uses the same decomposition method as ours to train SVM in parallel. But its implementation is different from our MRPsvm in many aspects. First, it is based on MPI implementation and aims at extracting parallelism from distributed memory systems, while our parallel algorithm is conducted by Map-Reduce threads on shared memory system. They are based on totally different models, and the number of threads, the granularity are different as well. Second, in its implementation, each process stores a copy of data samples. In the contrary, MRPsvm stores only one copy in the shared memory. Third, Pisvm adopts a distributed cache strategy in order to share the saved kernel elements across all of the processes. Each process stores locally a piece of the cache. Consequently, the work for updating gradients is divided and assigned globally to proper processors according to the cache locality. In contrast, our MRPsvm has only one copy of the cache, and each processor accesses the cache equally, so that the overhead of global assignment is avoided. However, we have to note that synchronization on cache write is required. In next section, we will compare the performance of Pisvm with that of MRPsvm on some benchmark datasets, providing direct evidence that our system is more efficient and suitable than the MPI implementation on multi-core systems.

V. Experiments and Results

We test MRPsvm by comparing it with the parallel implementation Pisvm and the serial implementation Libsvm on five widely used benchmark datasets. Although this comparison may not based on systems especially designed for multi-core architecture, we still have good reasons for doing so. Firstly, as the best knowledge as we know, there

Table I
THE PHYSICAL FEATURES OF THE MULTI-CORE SYSTEMS.

Features	Computer I	Computer II
# of CPUs	1	2
# of cores	4	8
Frequency	1600MHz*4	2327MHz*8
Memory	2G	4G
L2 cache	4M	6M*2

is no existing parallel implementation of general SVM that is specially developed for multi-core systems. Therefore there is a strong need to verify if our system could outperform the state-of-the-art parallel implementation. Secondly, most of the systems surveyed in section IV are not available to the public, while Pisvm , as a typical parallel implementation of SVM, is easy to obtain. Thirdly, the quadratic problem solver of Pisvm is the same as MRPsvm, hence, if we compare our system with Pisvm, the advantage of Map-Reduce framework is more convincing.

Two computers with different hardware architectures are adopted to check hardware effects. As shown in table I, the first computer has 4 cores with a shared L2 cache and memory. The second one is a dual-processor system with 4 cores in each processor. The cores in the same processor share one cache, and the main memory is shared among all of the cores. Both of the two computers are running Linux 2.6.27-7.

The five datasets are shown in table II. They vary in sample size and dimension. We train all SVMs with Gaussian kernel. The tolerance of termination criterion is set to 0.01. Since we focus on comparing of three systems, the outputs of these classifiers may not be optimal. In other words, we do not guarantee the parameters of SVM, i.e., C and γ, to reach optimal values. They are just chosen from the literature as shown in the last two columns of table II.

Since the caching technique is crucial for performance, for a reliable comparison, we set the cache size to be the same for all three systems. Furthermore, we restrict the cache size to be far smaller than the memory size in order to minimize page faults in runtime. Here we have to emphasize that the following reported performance might not be optimal for all three systems, only serving for comparison purpose.

Table III shows the results of the three implementations performed on 4 processors. The time columns represent the whole training time, i.e., from reading the problem to writing the outputs. Here we use "Times" to denote how many times faster of parallel implementation over sequential implementation:

$$Times = \frac{Time\ of\ Libsvm}{Time\ of\ parallel\ implementation}$$

By analyzing the results, we can see that MRPsvm has successfully parallelized SVM training. For all five datasets, much more time is saved when running MRPsvm than Libsvm. Especially in the first four cases, the speed of MRPsvm is more than 4 times higher than that of Libsvm. In all of the cases, MRPsvm achieves outstanding higher Times than Pisvm, indicating that MRPsvm is more suitable than Pisvm on multi-core systems.

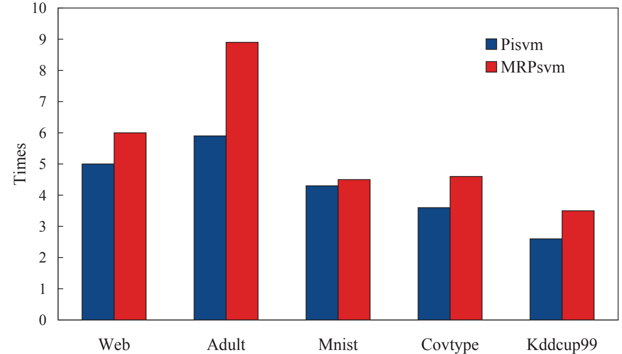

Figure 2. Times up of Pisvm and MRPsvm over Libsvm when running on computer II.

We note that in these experiments, the accuracy of the three classifiers is almost the same. In fact, the numbers of support vectors generated by these classifiers are also quite close. Actually, these three implementations have essentially the same mechanism in quadratic problem solving, i.e., to iteratively optimize one pair of variables until achieving global optimization. The difference in runtime mainly caused by the selected working set. Selecting different variables to perform optimization may induce totally different results in an iteration, but generally speaking, as long as global convergence is reached, the influence is marginal.

We show in figure 2 the times up of the two parallel solvers over the sequential solver when running on the second computer. MRPsvm again outperforms Pisvm on all of the datasets. Among them the best Times is achieved on Adult, while the worst Times is found on Kddcup99. This indicates that MRPsvm performs better on smaller datasets. The main reasons for worse performance on larger problem are due to locality and overheads of reduction. In each map, the updating of f requires accessing the whole data samples and several temporal vectors with the size close to l. Therefore, for large datasets, it is difficult to guarantee the locality for using L2 cache, especially when the cache is shared by several threads. Since we partition the global problem by columns, each map generates l intermediate f_i, so that the reduction is costly when l is very large.

In fact, the parallel performance on 8 cores only slightly outperforms that on 4 cores. As explained at the beginning of this section, this is because we did not make full use of the memory on computer II. Far more time can be saved if we increase the cache size with caution to the maximum. In this optimal case, the cache size of MRPsvm

Table II
DESCRIPTION OF THE FIVE DATASETS AND THE TWO PARAMETERS OF SVM ON EACH DATASET.

Dataset	# training samples	# testing samples	# Classes	# Dimensions	C	γ
Web	24,692	25,075	2	300	64	7.8152
Adult	32,561	16,281	2	123	100	0.5
Mnist	60,000	10,000	2	576	10	1.667
Covtype	435,759	145,253	8	54	10	2e-5
Kddcup99	898,430	311,029	2	122	2	0.6

Table III
THE TRAINING TIME AND ACCURACY OF THE THREE SYSTEMS ON FIVE DATASETS PERFORMED ON COMPUTER I. THE UNIT OF TIME IS SECOND.

Dataset	Libsvm		Pisvm			MRPsvm		
	Time	Accuracy	Time	Accuracy	Times	Time	Accuracy	Times
Web	306.4	97.6%	117.5	97.6%	2.6	65.8	97.6%	4.7
Adult	311.6	82.7%	91.4	82.7%	3.4	59.2	82.7%	5.3
Mnist	517.8	99.8%	148.7	99.8%	3.5	123.2	99.8%	4.2
Covtype	20260.7	51.0%	5612.6	51.0%	3.6	3895.1	51.0%	5.2
Kddcup99	726.8	92.0%	415.5	92.0%	1.7	351.9	92.0%	2.1

and Libsvm is larger than that of Pisvm, since the former two systems generally require less memory. Therefore, it is implied that more improvements can be achieved for MRPsvm and Libsvm than Pisvm.

VI. CONCLUSION

This paper proposes a parallel implementation of SVM for multi-core and multiprocessor systems. It implements decomposition method and utilize SMO as inner solver. The parallelism is conducted to update the vector f in the decomposition step and is programmed in the simple, yet pragmatic programming framework Map-Reduce. A shared cache is designed to save the kernel matrix columns when the data size is very large. Extensive experimental results show that MRPsvm is very efficient in solving large scale problems. For instance, the speed on 4 processors can increase more than 4 times than Libsvm for most of the applications. It overwhelms the state-of-the-art Pisvm in all benchmark tests in the sense of both speed and memory requirement.

The multi-core system is already highly available in modern market, and is dominating the trend of processor development. This makes MRPsvm to be practical since it requires only easily available, affordable and single computer. These performance improvements brought by MRPsvm can extend the potential feasibility of SVM in solving increasingly challenging problems. Furthermore, the success of MRPsvm indicates that Map-Reduce is a possible option to parallelize machine learning algorithms.

However, several aspects are still worth considering for the purpose of further improvements. For instance, how to define the best granularity of map work for a particular dataset, how to improve data locality, shrinking, further parallelizing other slow operations such as the computation of inner kernel matrix.

REFERENCES

[1] V. N. Vapnik, *The nature of statistical learning theory*. Springer-Verlag New York, Inc., 1995.

[2] F. Lai, F. Magoulès, and F. Lherminier, "Vapnik's learning theory applied to energy consumption forecasts in residential buildings," *International Journal of Computer Mathematics*, vol. 85, no. 10, pp. 1563–1588, 2008.

[3] H. X. Zhao and F. Magoulès, "Parallel support vector machines applied to the prediction of multiple buildings energy consumption," *Journal of Algorithms & Computational Technology*, vol. 4, no. 2, pp. 231–249, 2010.

[4] E. Osuna, R. Freund, and F. Girosi, "Training support vector machines: an application to face detection," in *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition)*, 1997, pp. 130–136.

[5] T. Joachims, "Making large-scale support vector machine learning practical," in *Advances in kernel methods: support vector learning*. Cambridge, MA, USA: MIT Press, 1999, pp. 169–184.

[6] E. Y. Chang, K. Zhu, H. Wang, H. Bai, J. Li, Z. Qiu, and H. Cui, "Psvm: Parallelizing support vector machines on distributed computers," in *NIPS*, vol. 20, 2007.

[7] J. C. Platt, "Fast training of support vector machines using sequential minimal optimization," in *Advances in kernel methods: support vector learning*. Cambridge, MA, USA: MIT Press, 1999, pp. 185–208.

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

[9] S. Shalev-Shwartz, Y. Singer, and N. Srebro, "Pegasos: Primal estimated sub-gradient solver for svm," in *Proceedings of the 24th international conference on Machine learning*, 2007, pp. 807–814.

[10] C. C. Chang and C. J. Lin, *LIBSVM: a library for support vector machines*, 2001, available online at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[11] R. E. Fan, P. H. Chen, and C. J. Lin, "Working set selection using second order information for training support vector machines," *Journal of Machine Learning Research*, vol. 6, pp. 1889–1918, 2005.

[12] L. Zanni, T. Serafini, and G. Zanghirati, "Parallel software for training large scale support vector machines on multiprocessor systems," *Journal of Machine Learning Research*, vol. 7, pp. 1467–1492, 2006.

[13] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[14] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 29–42.

[15] J. Pan, Y. L. Biannic, and F. Magoulès, "Parallelizing multiple group-by query in share-nothing environment: a mapreduce study case," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 856–863.

[16] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 260–269.

[17] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007, pp. 1029–1040.

[18] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," in *NIPS*, 2006, pp. 281–288.

[19] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 13–24.

[20] Z. A. Zhu, W. Z. Chen, G. Wang, C. G. Zhu, and Z. Chen, "P-packsvm: Parallel primal gradient descent kernel svm," *Proceedings of the 9th IEEE International Conference on Data Mining*, vol. 0, pp. 677–686, 2009.

[21] D. Bickson, E. Yom-tov, and D. Dolev, "A gaussian belief propagation solver for large scale support vector machines," in *Proceedings of the 5th European Conference on Complex Systems*, 2008.

[22] H. P. Graf, E. Cosatto, L. Bottou, I. Durdanovic, and V. Vapnik, "Parallel support vector machines: The cascade svm," in *NIPS*, 2004.

[23] J. X. Dong, A. Krzyzak, and C. Y. Suen, "Fast svm training algorithm with decomposition on very large data sets," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 4, pp. 603–618, 2005.

[24] T. Hazan, A. Man, and A. Shashua, "A parallel decomposition solver for svm: Distributed dual ascend using fenchel duality," *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, vol. 0, pp. 1–8, 2008.

[25] L. J. Cao, S. S. Keerthi, C. J. Ong, J. Q. Zhang, U. Periyathamby, J. F. Xiu, and H. P. Lee, "Parallel sequential minimal optimization for the training of support vector machines," *IEEE Transactions on Nueral Networks*, vol. 17, no. 4, pp. 1039–1049, 2006.

[26] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast support vector machine training and classification on graphics processors," in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 104–111.

[27] D. Brugger, "Parallel support vector machines," in *Proceedings of the IFIP International Conference on Very Large Scale Integration of System on Chip*, 2007.