Parallel SVM for large data-set mining

L. Qian & T. Hung
Institute of High Performance Computing, Singapore

Abstract

Support Vector Machine is gaining popularity as a data mining technique. As it is being extended and applied to more industrial applications, performance limitation will be a real concern when dealing with real commercial activities where the data set can be very huge and quick analysis turnaround is required for quick business decision making. This issue needs to be addressed in order for SVM to become a viable and practical commercial data mining tool. In this paper we will present, evaluate and compare the performance of one particular parallel SVM training algorithm based on MPI programming. This parallel version of SMO algorithm is proposed to speed up SVM training. One performance model based on generalized Amdahl’s formulation is applied to analyze the scalability in terms of problem size and complexity. Further, it is used to provide some guideline to determine the best algorithm/architecture combination.

1 Introduction

In the past few years, there has been a lot of excitement and interest in Support Vector Machines [1–3] as an important machine learning tools because they have yielded excellent generalization performance on a wide range of problems. Training SVM requires solving a QP problem, where the number of variables to be optimized is equal to the number of training data. For large size problems, which are quite normal in data mining applications, standard QP techniques are not useful, as they require a large amount of computer memory to store the kernel matrix completely and explicitly.

Fast iterative algorithms that are also easy to implement have been suggested [4, 5; 13, 14, 16]. Platt's Sequential Minimization Algorithm (SMO) [15; 4] is an important example. A remarkable feature of SMO is that it doesn’t store the Matrix completely. As pointed out by Platt, SMO scales quadratically in the
number of training data patterns, while the scalability of other algorithms is cubic. When data sets become huge, it is difficult to use the method for training SVM. Another issue in SVM training is that it is often necessary to adjust the regularization constant in order to achieve better generalization, however, when the parameter change to big values, the time performance of SVM training is significantly reduced.

In this paper we propose one parallel version of SMO training algorithm to tackle both problems mentioned above and evaluate the performance of our parallel program. Determining the best parallel program that solves a problem is considerably complicated, it requires comprehensive measurement of its scalability: the system’s ability to increase speedup as the number of process increased. We adopt Isoefficiency matrix method [7] and generalized Amadahl’s law [8] to measure and evaluate the scalability and performance of the parallel SMO. Computational comparison on bench-mark datasets shows that the modifications perform significantly faster than the original SMO in most situations. The ideas mentioned in this paper can also be applied to the SMO regression algorithm [9].

The paper is organized as follows. In section 2, we briefly discuss the SVM problem formulation, a short summary of Platt’s SMO algorithm and modified SMO method [16] which our parallel SMO is actually based. In section 3, we introduce the performance and scalability performance measurement model to evaluate parallel programme. In section 4, we discussed in detail our implementation of the parallel SMO training algorithm. In section 5, we show our computation and evaluation result on IBM SP2 parallel multiprocessor system located in IHPC, Singapore.

2 SMO method overview

We will base our discussion of SVM training on classification problems which can be defined as following. Given a set of data points \((X_i, y_i)\), \(X_i \in \mathbb{R}^d\) is the input vector of \(i\)th 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.

\[
\text{maximize: } R(\alpha_i) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j K(X_i, X_j) \\
\text{subject to: } \sum_{i=1}^{l} \alpha_i y_i = 0 \\
0 \leq \alpha_i \leq C, i = 1, \ldots, l
\]

where \(K(X_i, X_j)\) is the kernel function satisfying Mercer’s condition. The mostly widely used kernel function is the Gaussian function \(e^{-|X_i - X_j|^2/\sigma^2}\), where \(\sigma^2\) is the width of the Gaussian kernel. \(\alpha_i\) is the Lagrange multiplier to be optimized. For each of training data patterns, one \(\alpha_i\) is associated. \(C\) is the regularization constant. After solving the QP problem (1), the following decision
function is used to determine the class label for a new data pattern where \( b \) is obtained from the solution of (1).

\[
f(X) = \sum_{i=1}^{t} \alpha_i y_i K(X_i, X) - b \tag{3}
\]

Based on the idea of the working set methods, Platt 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. Later, Keerthi et al. ascertain inefficiency associated with Platt’s SMO and suggested two modified versions of SMO. We will refer to this modification as the modified SMO algorithm.

We begin the description of the modified SMO by giving the notation used. Let

\[
I_0 = \{ i : y_i = 1, \ 0 < \alpha_i < C \} \cup \{ i : y_i = -1, \ 0 < \alpha_i < C \}, \quad I_1 = \{ i : y_i = 1, \ \alpha_i = 0 \},
\]

\[
I_2 = \{ i : y_i = -1, \ \alpha_i = C \}, \quad I_3 = \{ i : y_i = 1, \ \alpha_i = C \}, \quad \text{and} \quad I_4 = \{ i : y_i = -1, \ \alpha_i = 0 \}.
\]

\[
I = \bigcup I_i, \quad i = 0, \ldots, 4 \text{ denotes the index of training data patterns.}
\]

\[
F_i = \sum_{j=1}^{t} \alpha_j y_j K(X_j, X_i) - y_i.
\]

\[
b_{up} = \min \{ F_i : i \in I_0 \cup I_1 \cup I_2 \}, \quad F_{up} = \arg \min_i F_i.
\]

\[
b_{low} = \max \{ F_i : i \in I_0 \cup I_3 \cup I_4 \}, \quad F_{low} = \arg \max_i F_i. \quad \tau = 10^{-6}.
\]

The idea of the modified SMO is to optimize the two \( \alpha_i \) that are associated \( b_{up}, b_{low} \) and has the index of \( I_{up} \) and \( I_{low} \) at each step, according to (4) and (5).

\[
\alpha_i^{\text{new}} = \alpha_i^{\text{old}} - \frac{y_i (F_i^{\text{old}} - F_i^{\text{2,old}})}{\eta}
\]

\[
\alpha_i^{\text{new}} = \alpha_i^{\text{old}} + s(\alpha_2^{\text{old}} - \alpha_2^{\text{new}})
\]

where the variables associated with calculate \( I_{up}, I_{low} \) are represented using the subscripts “1” and “2”.

\[
s = y_1 y_2, \quad \eta = 2K(X_1, X_2) - K(X_1, X_1) - K(X_2, X_2).
\]

After optimizing \( \alpha_i \) of \( I_{up}, I_{low} \), denoting the error on the \( i \)th training data pattern, is updated according to the following:

\[
F_i^{\text{new}} = F_i^{\text{old}} + (\alpha_1^{\text{new}} - \alpha_1^{\text{old}}) y_i K(X_1, X_i) + (\alpha_2^{\text{new}} - \alpha_2^{\text{old}}) y_2 K(X_2, X_i)
\]

Based on the updated values of \( F_i, b_{up}, I_{up}, b_{low}, I_{low} \) are recalculated for the optimization of \( \alpha_i \) at the next step.
In addition, the value of Eq. (1), represented by Dual, is also updated at each step using

\[ Dual_{new} = Dual_{old}^2 - \frac{\alpha_i^{new} - \alpha_i^{old}}{y_i}(F_i - F_2) + \frac{1}{2} \eta (\alpha_i^{new} - \alpha_i^{old})^2 \]  

(7)

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

\[ DualityGap = \sum_{i=0}^{l} \alpha_i y_i F_i + \sum_{i=0}^{l} \varepsilon_i \]  

(8)

where \[ \varepsilon_i = C_{\text{max}}(0, b-F_i) \quad \text{if} \quad y_i = 1 \]
\[ \varepsilon_i = C_{\text{max}}(0, -b+F_i) \quad \text{if} \quad y_i = -1 \]

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 \( \alpha_i = 0 \), \( F_i = -y_i \), \( Dual = 0 \), \( i = 1, \cdots, l \)

Calculate \( b_{\text{up}}, I_{\text{up}}, b_{\text{low}}, I_{\text{low}}, \) and DualityGap

Until DualityGap \( \leq |Dual| \)

1. Optimize \( \alpha(I_{\text{up}}), \alpha(I_{\text{low}}) \)
2. Update \( F_i, i = 1, \cdots, l \)
3. Calculate \( b_{\text{up}}, I_{\text{up}}, b_{\text{low}}, I_{\text{low}}, \) and DualityGap and update Dual

repeat

3 Isoefficiency performance model

The definition of scalability comes from Amdahl’s law which is tied to efficiency and speedup. There are two important scalability metrics: the isoefficiency function based on parallel computing efficiency and iso-speed metric based on parallel computing speed [10]. The isoefficiency function of a parallel system is determined by abstracting the size of a computing problem as a function of the number of processors, subject to maintaining a desired parallel efficiency (between 0 and 1). Specially, the efficiency is defined as \( E = \frac{1}{1+T_0(n,W)/Wtc} \), Where \( T_0 \) is the total overhead caused by all processors to do the computation in parallel. \( tc \) is the average executing time per operation in the architecture. \( W \) is the problem size and \( n \) is the number of processors. Here \( tW \) is the sequential runtime of an algorithm. In order to keep efficient \( E \) at certain none negative value between 0 and 1, the scalability is determined by the overhead function \( T_0(n,W) \). the larger the function \( T_0(n,W) \), the lower the scalability of the algorithm on the architecture will be. If an analytical form \( T_0 \) of a given algorithm on a given architecture is described as a function of \( n \) and \( W \), the isoefficiency \( (E(n)) \) curve using up to \( n \) processors can be generated for evaluating the scalability.
The isoefficiency function captures, in a single expression, the effects of characteristics of the parallel algorithm as well as the parallel architecture on which it is implemented. In addition, the isoefficiency function shows that it is necessary to vary the size of a problem on a size changeable parallel architecture so that the processing efficiency of each process can remain constant.

Isoefficiency analysis has been found to be very useful in characterising the scalability of a variety of parallel systems. An important feature of Isoefficiency analysis is that in a single expression, it succinctly captures the effects of characteristics of a parallel algorithm as well as the parallel architecture on which it is implemented. By performing Isoefficiency analysis, one can test the performance of a parallel program on a few processors and then predict its performance on a large number of processors. However Isoefficiency analysis is more of a theoretical tool which provide certain guideline to evaluate, in practice it is quite to define Isoefficiency function explicitly. We use experiment data to estimate the actual function.

4 Parallel version of SVM training algorithm

In the modified SMO, updating $F_i$ array at the iteration (2) which includes the kernel evaluations dominates most of the computation time. As shown in our experiment, over 90% of the total computation time is used for this calculation. So it is better for us to perform the updating of $F_i$ array using multiple CPU processors. As the updating of $F_i$ array is independently evaluated one training data pattern at a time according to (6), the SPMD mode can be used to execute this program in parallel. Firstly, the entire training data set is equally partitioned into smaller subsets. And each of the partitioned subsets is distributed into one CPU processor. The updating of $F_i$ array on the entire training data set is then performed by simultaneously running multiple CPU processors, while 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 should be saved.

Based on the SPMD mode, each processor could obtain one $b_{up}$ and one $b_{low}$ as well as associated $I_{up}$ and $I_{low}$ for its assigned training data patterns. These $b_{up}$, $I_{up}$, $b_{low}$ and $I_{low}$ are local in the sense they are obtained only based on a subset of all the training data patterns. According to the notation described in Section 2, $b_{up}$ and $b_{low}$ on the entire training data set are respectively the minimum value of local $b_{up}$ and the maximum value of local $b_{up}$ of each processor, thus determining $I_{up}$ and $I_{low}$ on the entire training data set. The $b_{up}$, $I_{up}$, $b_{low}$ and $I_{low}$ on the entire training data set are referred to $b_{up}$, $I_{up}$, $b_{low}$ and $I_{low}$ in the later sections.

As shown in (8), calculating DualityGap is independently evaluated one training data pattern at a time. So this program can also be executed using SPMD. According to (8), each processor could calculate a different part of DualityGap based on its assigned training data patterns. The DualityGap on the entire training data patterns is the sum of the DualityGap of each processor.

In summary, the parallel SMO firstly equally partitions the entire training data set into smaller subsets and distributes each of the partitioned subsets into one different CPU processor. Based on its assigned training data patterns, each
processor finds out one local $b_{up}$ and one local blow as well as associated $I_{up}$ and $I_{low}$. From the $b_{up}, I_{up}$, $b_{low}$ and $I_{low}$, the global $b_{up}, I_{up}$, $b_{low}$ and $I_{low}$ are found out. The two $\alpha_i$ associated with global $b_{up}$ and global blow are then optimized using any one CPU processor. Later, the updated $\alpha_i$ values are sent back to all the processors for updating $Fi$ array, with each processor updating a different subset of $Fi$ array. In addition, each processor also updates a different subset of DualityGap based on its assigned training data patterns. The final value of DualityGap is the sum of the DualityGap of each processor. The procedure is repeated until the optimal condition is satisfied. A brief description of the parallel SMO can be summarized as follows.

Parallel SMO Algorithm:

Notation: $P$ is the total number of processors. $\cap_{i=1}^{P} \cap_{l=1}^{l} = l$ is a partition of all the training data patterns and the index of training data patterns on the processor $j$.

\[ F_{i}^{j} = \sum_{k=1}^{\cap_{i}^{j}} \alpha_i y_i K(X_k, X_i) - y_i. \]

\[ b_{up}^{j} = \min \{ F_{i}^{j} : i \in I_0 \cup I_1 \cup I_2 \}, I_{up}^{j} = \arg \min_i F_{i}^{j}. \]

\[ b_{low}^{j} = \max \{ F_{i}^{j} : i \in I_0 \cup I_3 \cup I_4 \}, I_{low}^{j} = \arg \max_i F_{i}^{j}. \]

$F_{i}^{j}$, $b_{up}^{j}$, $I_{up}^{j}$, $b_{low}^{j}$, $I_{low}^{j}$, and DualityGap$^{j}$, $i \in l$ denote the variables associated with processor $j$.

\[ b_{up} = \max_{j=1}^{P} b_{up}^{j}, I_{up} = \arg I_{up}^{j}, b_{low} = \max_{j=1}^{P} b_{low}^{j}, I_{low} = \arg I_{low}^{j}. \]

\[ DualityGap = \sum_{j=1}^{P} \text{DualityGap}^{j}. \] $b_{up}$, $I_{up}$, $b_{low}$, $I_{low}$, and DualityGap denote the global variables.

Initialize $\alpha_i^{j} = 0$, $F_{i}^{j} = -y_i$, Dual = 0, $j = 1, \ldots, P$.

Calculate $b_{up}^{j}$, $I_{up}^{j}$, $b_{low}^{j}$, $I_{low}^{j}$, DualityGap$^{j}$, $j = 1, \ldots, P$.

Obtain $b_{up}$, $I_{up}$, $b_{low}$, $I_{low}$, and DualityGap

Until DualityGap $\leq \tau \times $ Dual

1. Optimize $\alpha_{up}^{j}$, $\alpha_{low}^{j}$

2. Update $F_{i}^{j}$, $j = 1, \ldots, P$.

3. Calculate $b_{up}^{j}$, $I_{up}^{j}$, $b_{low}^{j}$, $I_{low}^{j}$, DualityGap$^{j}$, $j = 1, \ldots, P$.

4. Obtain $b_{up}$, $I_{up}$, $b_{low}$, $I_{low}$, and DualityGap and update Dual
repeat

Due to the use of multiple CPU processors, communication is required in the parallel SMO, which takes additional time. As aforementioned, for getting $b_{up}$, $I_{up}$, $b_{low}$ and $I_{low}$ from local $b_{up}$, $I_{up}$, $b_{low}$ and $I_{low}$ of each processor,
communication is required. For updating $F_i$ array, the updated $\alpha_i$ values, as well as the corresponding training data patterns, the associated class labels, the training errors should be broadcast to all the processors, which also takes additional time. As demonstrated in our experiment, in the parallel SMO the communication time is very little compared to the total computation time.

5 Scalability and performance evaluation

The parallel SMO is tested against the sequential 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 7 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, all the stop criteria used in both algorithms such as the value of $\tau$ are the same.

The MNIST hand written digit data 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/terraSvm/ and has also been used in Dong et al.'s work on speeding up the sequential SMO [11].

Table 1: CPU time for different problem size & processors configurations.

<table>
<thead>
<tr>
<th>Data</th>
<th>20K</th>
<th>40K</th>
<th>60K</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>5</td>
<td>50</td>
<td>300</td>
</tr>
<tr>
<td>Processor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>276</td>
<td>380</td>
<td>2507</td>
</tr>
<tr>
<td>2</td>
<td>144</td>
<td>198</td>
<td>1294</td>
</tr>
<tr>
<td>8</td>
<td>120</td>
<td>139</td>
<td>734</td>
</tr>
<tr>
<td>16</td>
<td>87</td>
<td>108</td>
<td>567</td>
</tr>
<tr>
<td>20</td>
<td>138</td>
<td>148</td>
<td>920</td>
</tr>
<tr>
<td>25</td>
<td>66</td>
<td>79</td>
<td>517</td>
</tr>
<tr>
<td>32</td>
<td>76</td>
<td>162</td>
<td>1074</td>
</tr>
</tbody>
</table>

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. Since our purpose is to test and evaluation of the scalability performance, to simplify the scenario, I just choose the case 7 data. In order to better define problem size, I will two different parameters: number of data points and different value of regularization constant C. The different data points tested are 10K, 20K, 40K, and 60K and the different regularization constant tested is 5, 50, 300. Since our system is busy with some other programs, to capture the actually computation time, CPU time is used instead of program elapse time. According to our experiment, these two times may differ in scale of 1-3, depends on utilization of the system.

The CPU time with different configurations and different processors are given in Table 1. It can be observed from the table that our parallel SMO algorithm
does achieve better performance in training large scale problem in terms of cpu time used. Further, it can be seen that our algorithm works better and can gain significant improvement when problem size grows, no matter in terms of data point used or regularization constant C used. We can further notice that processors used increase to around 25, the performance stops increasing which should attribute to heavy communication involved.

Table 2: Speedup table for different problem size & processors configurations.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Data</th>
<th>20K</th>
<th>40K</th>
<th>60K</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>5</td>
<td>50</td>
<td>300</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>1.92</td>
<td>1.92</td>
<td>1.94</td>
<td>1.88</td>
</tr>
<tr>
<td>8</td>
<td>2.30</td>
<td>2.74</td>
<td>3.42</td>
<td>2.55</td>
</tr>
<tr>
<td>16</td>
<td>3.16</td>
<td>3.51</td>
<td>4.42</td>
<td>4.37</td>
</tr>
<tr>
<td>20</td>
<td>2.00</td>
<td>2.57</td>
<td>2.72</td>
<td>4.07</td>
</tr>
<tr>
<td>25</td>
<td>4.18</td>
<td>4.82</td>
<td>4.85</td>
<td>2.57</td>
</tr>
<tr>
<td>32</td>
<td>3.63</td>
<td>2.35</td>
<td>2.33</td>
<td>2.55</td>
</tr>
</tbody>
</table>

In order to measure the scalability of the problem, speed up and efficiency matrix are calculated and the result is given in Table 2&3. Here speedup and efficiency is defined as

\[
\text{speedup} = \frac{\text{the elapsed time of the sequential SMO}}{\text{the elapsed time of the parallel SMO}} \quad \text{and} \quad \frac{\text{speedup}}{\text{number of processors}}
\]

respectively. When use isoefficiency model to evaluate scalability of parallel program, it is more of our interest to analysis the efficiency matrix. The aim is to define some isoefficiency combination so that to have an idea about the isoefficiency function.

From table 3 we can find following combination points with isoefficiency such as (C5, p16), (C50, p20) and (C300, p20) for data size of 60K and (20K, p16), (40K, p20) and (60K, p20) for C50. Thus we can have an idea about our parallel algorithm's scalability: when processor utilized increase to 1.25 times, our problem size has to increase almost 10 times in terms of C and about 1-2 times in terms of data points. According to isoefficiency theory the ideal parallel program should have linear growth rate to be highly scalable. According to this standard our parallel program performs differently in respect to data size and C parameter. It scalability is good when we use data size as an indicator of problem size, it can scale quite efficiently. But the scalability performance is not good if we change regularization constant.
Summary and future works

This paper proposes and evaluates the parallel implementation of SMO using MPI. The parallel SMO uses multiple CPU processors to deal with most of the computation of SMO. The entire training data set is firstly equally partitioned into smaller subsets. Each of the partitioned subsets is distributed into one CPU processor. The computation experiment shows that the parallel SMO does reduce computation time dramatically and can be used for big data set problems. The scalability performance analysis shows that the parallel algorithm behaves differently under different indication of problem size. It can scale nicely in terms of data points size computed but scale badly in terms of regulation constant change. This work is very useful to speedup SVM training time for the research where multiple CPU processors machine is available, especially when dealing with large data sets problems. Future work needs to extend the parallel SMO from classification for regression estimation.

References

670 Data Mining IV

Kernel Methods: Support Vector Machines, MIT Press Cambridge, MA.


