ENERGY-EFFICIENT TRAINING OF SVM ON MULTICORE PLATFORMS

Heegon Kim, Sungju Lee, Yongwha Chung, Daihee Park, Korea University, Korea, {khg86, peacfeel, ychungy, dhpark}@korea.ac.kr
Hansung Lee
ETRI, Korea
mohan@etri.re.kr

ABSTRACT

Recent developments in multicore processors have enabled high performance implementations of machine learning algorithms such as Support Vector Machine (SVM). In this paper, we parallelize the training phase of a SVM in order to reduce the energy consumption. After analyzing the computational characteristics of the SVM and the machine characteristics, we distribute the most time consuming steps to multiple cores while satisfying the data dependencies. Our experimental results show that the 4-core based parallel implementation can reduce the energy consumption by a factor of 1.5 with the exactly same quality as the sequential SVM training.

Keyword: Parallel processing, Multicore processor, Machine learning, SVM

1. INTRODUCTION

Among many classification methods, SVM [1] has demonstrated superior performance in machine learning applications [2]. However, training SVM is a very computationally intensive job. Although many researches have been conducted to accelerate training time [3,4], SVM training time is still significant for larger training sets.

In this paper, we consider a training of a binary-class SVM which implements the Sequential Minimal Optimization (SMO) [3]. Especially, in order to reduce the energy consumption, we focus on parallelizing the training phase of a SVM using Pthread [5] and the target architecture is a multicore-based platform. As we know, the multicore architecture has been a trend since 2005 and “many” core CPUs will be released continuously [6]. In fact, algorithms for machine learning applications need to consider such parallel architectures, and many researches were conducted in order to develop parallel SVMs. For example, [7] developed a distributed SVM approach for distributed memory machines, whereas [8] developed a parallel SVM approach for Single Instruction Multiple Data (SIMD) machines such as GPU. However, both coarse-grained (targeted for distributed memory machines) and fine-grained (targeted for SIMD machines) may not be suitable for multicore-based machines. Another difficulty in parallelizing SVM is that SVM has a complicated pattern of data dependencies. Thus, most of the previous parallel SVM approaches [7,8] ignored some data dependencies for easy parallelization and provided slightly different qualities from that of the sequential training.
To extract the suitable parallelism in the SVM training targeted for multicore-based platforms, we first analyze the computational characteristics of the SVM training. Then, we distribute the most time consuming steps to multiple cores while satisfying the data dependencies existed in the computation. Our experimental results show that the proposed parallel approach can provide both a speedup close to the upper limit computed by the Amdahl’s Law and the exactly same quality as the sequential SVM training. Also, by parallelizing SVM training, we can reduce the energy consumption of the sequential training by a factor of 1.5 and the execution time by a factor of 2.5, respectively.

The rest of the paper is structured as follows. Section 2 describes SVM and parallel SVM. Section 3 explains the proposed approach for parallelizing the SVM training using Pthread. Finally, Section 4 and 5 describe the experimental results and conclusion.

2. BACKGROUND

2.1 SVM
It has been shown that discriminative classifiers achieve better performance than generative classifiers in supervised learning. For example, SVMs directly maximize the margin of a linear separator between two sets of points in the vector space [1]. Since the model is linear and simple, the maximum margin criterion is more appropriate than maximum likelihood or other generative model criteria. Although the idea of SVM becomes one of the most widely used machine learning techniques, the size of real-world datasets makes the solution of a Quadratic Programming (QP) problem using general-purpose solvers impractical. For this reason, efficient ad-hoc algorithms that take advantage of the special structure of the QP problem have been developed. For example, the Sequential Minimal Optimization (SMO) algorithm decomposes the original QP problem into two-dimensional subproblems which can be solved analytically [3]. The idea of the SMO is to compute a solution iteratively by optimizing two coefficients at each iteration. In this paper, we choose SMO-based LIBSVM [9] because it is widely used in practice and many good open source implementations exist, and Figure 1 shows the computational structure of LIBSVM training.

Unfortunately, training times are still significant for many real-world datasets [10]. The reason is that matrix Q can be very large and cannot be kept entirely in memory. Thus, the SMO needs to recompute the values $Q_{ij}$ many times, which requires many evaluations of the kernel function. Thus, we need to consider parallelizing the SMO training in order to make it tractable.
2.2 Parallel SVM

Several approaches have been proposed to parallelize SVM. They can be classified into several categories according to the type of QP solver. Based on stochastic gradient descent method, P-pack SVM optimized SVM training directly on the primal form of SVM [11]. Psvm proposed in [12] was based on interior-point QP solver. It approximated the kernel matrix by incomplete Cholesky Factorization. [13] solved the problem by Gaussian belief propagation. The decomposition method attracts more attention than the above solvers. [7] trained several SVMs on small data partitions, then it aggregated support vectors from two-pair SVMs to form new training samples. The similar idea was adopted by [14] and sub-SVMs were performed on block diagonal matrices which were regarded as the approximation to the original kernel matrix. [14] parallelized SVM-light with improved working set selection and inner QP solver. [15] proposed a parallel decomposition solver using Fenchel Duality. [15] parallelized a randomized sampling algorithm for SVM. [14] and [15] parallelized SMO solver for training SVM. Both works mainly focus on updating the gradient for KKT condition evaluation and the working set selection.

Although many parallel approaches have been proposed, they provide different qualities compared to the sequential approach. Note that the output of training is the set of support vectors which can classify forthcoming inputs. However, the number of support vectors from the parallel approach is different from that of the sequential approach [8]. This may be due to that some of the data dependencies in the training computation have been ignored and/or avoided (through approximation) for easy of parallelization. In this paper, we parallelize the SVM training such that the output of the parallel SVM training is the same as that of the sequential training. Also, we exploit the parallelism suitable for multicore processors.
3. PARALLELIZATION FOR SVM TRAINING

In this paper, we analyze the computational characteristics of SVM training and its parallelism by using Amdahl’s law. Then, we parallelize the SVM training with Pthread on a shared memory machine (i.e., multi-core PC).

3.1 Computational Characteristics of SVM Training

To understand the computational characteristics of the LIBSVM training, we first analyze the execution times and data dependencies of each major step. The measurement was conducted on an Intel core i7 720QM 1.6GHz 4-core PC with Adult and Web dataset [10] and summarized in Table 1. In Table 1, “Sequential” means the computational step has some data dependencies and should be executed sequentially, whereas “Parallel” means the step has no data dependency and can be executed in parallel. Since the Making Q Matrix step occupies most of the time, we need to parallelize the step. Note that, based on Table 1, we can compute the upper limit of a speedup by using the Amdahl’s Law [16]. Since 87% of the computation should be executed sequentially, the maximum speedup on a 4-core processor is 2.89.

The Making Q Matrix step consists of a LOOP phase, and uses \textit{Get}_Q() operation in order to generate the Q Matrix with a kernel function. Although the execution time of a single \textit{Get}_Q() operation is relatively short, the Making Q Matrix step calls a large number of \textit{Get}_Q() operations in order to train the dataset. Accordingly, in order to improve the speed of a whole training process, we should parallelize the \textit{Get}_Q() operations. Fortunately, the single \textit{Get}_Q() operation does not have any data dependency, and thus can be easily parallelized. However, some data dependencies exist between each \textit{Get}_Q() operations. Therefore, we need to parallelize the \textit{Get}_Q() operations efficiently by using barrier synchronization.

### Table 1. Computational Characteristics of LIBSVM Training

<table>
<thead>
<tr>
<th>Step</th>
<th>Execution Time</th>
<th>Sequential or Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>0.003 second (0.02%)</td>
<td>Sequential</td>
</tr>
<tr>
<td>Making Q Matrix</td>
<td>13.114 second (87.22%)</td>
<td>Parallel</td>
</tr>
<tr>
<td>Other Steps</td>
<td>1.918 second (12.76%)</td>
<td>Sequential</td>
</tr>
</tbody>
</table>

3.2 Parallelizing SVM Training using Pthread

As mentioned above, the execution time of a single \textit{Get}_Q() operation is relatively short, but the Making Q Matrix step consumes the most of the execution time due to a large number of \textit{Get}_Q() calls. Also, some data dependencies exist between each \textit{Get}_Q() operations. To parallelize the \textit{Get}_Q() operations, a straightforward approach may assign each of \textit{Get}_Q() to a separate thread. Since many threads need to be created within the LOOP (i.e., the Making Q Matrix step), however, the parallel overhead of this straightforward approach is increased significantly. In order to reduce this overhead, we create threads outside of the LOOP (i.e., Initialize step), and use barrier synchronizations. In this case, the \textit{Get}_Q() operations can be executed in parallel by creating multiple threads, while the other operations are executed sequentially. Figure 2 shows the proposed approach to parallelize the Making Q Matrix step.
Figure 2. Parallel Processing of LIBSVM Training

Figure 3 illustrates the Making Q Matrix step executed by 3 threads. In order to generate the Q Matrix, the \textit{Get.Q()} operations require some parameters. Each created thread can access the parameters by using the shared memory-based programming (\textit{i.e., Pthread}). Also, by using barrier synchronization, we can solve the data dependency problem existing between each \textit{Get.Q()} operations.

Figure 3. An Illustration of Executing the Making Q Matrix Step with 3 Threads
4. EXPERIMENTAL RESULTS

For evaluating the proposed approach, we used an Intel i7 720QM processor having 4-cores (1.6 GHz) and RAM 4.0 GB, and Adult and Web dataset [10]. Note that, the goal of this evaluation is to measure the energy saving as well as the time saving by parallelizing LIVSVM. As shown in Figure 4, we set the configuration of the power measurement environment. WT210 supplies the main power, and an Intel i7 720QM 4-core platform performs LIBSVM in order to train the dataset. Simultaneously, another Intel (CPU 1.7GHz, RAM 1.0GB) machine accumulates the measured data with WT FileReader.

![Figure 4. Configuration of the Power Measurement](image)

First, in order to evaluate the time saving, we measured the execution time of the sequential LIBSVM and the parallel LIBSVM, respectively. Note that, the sequential LIBSVM is open source code [9], our parallel LIBSVM was implemented by using Pthread. As shown in Table 2, we can confirm that the speedup of parallel LIBSVM is not enough to improve the performance sufficiently. However, the obtained speedup is very close to the upper limit of a speedup computed by the Amdahl’s Law. Also, the proposed approach can provide the exactly same quality as the sequential SVM training.

<table>
<thead>
<tr>
<th># of Threads</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adult-6</td>
<td>1.00</td>
<td>1.71</td>
<td>2.23</td>
<td>2.52</td>
</tr>
<tr>
<td>Web-8</td>
<td>1.00</td>
<td>1.66</td>
<td>2.15</td>
<td>2.51</td>
</tr>
</tbody>
</table>

In order to evaluate the energy saving, we measured the power consumption and the energy consumption of LIBSVM with Adult and Web dataset. As shown in Table 3, the parallel SVM on a multicore platform requires more power consumption than the sequential SVM. It is because each core performs some computation. Since the execution time was reduced by 2.5 times, however, the parallel approach could reduce the total energy consumption (= power consumption × execution time). As shown in Figure 5, we confirm that parallelizing training can reduce the energy consumption of the sequential training by a factor of 1.5. That is, we can verify that parallel processing with a multicore platform is an effective approach to reduce energy consumption, which may be required for embedded applications.
Table 3. Comparison of Sequential SVM and Parallel SVM

<table>
<thead>
<tr>
<th></th>
<th>Sequential SVM on 1-core</th>
<th>Parallel SVM on 4-core</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time(Sec)</td>
<td>Power(W)</td>
</tr>
<tr>
<td>Adult-6</td>
<td>15.10</td>
<td>34</td>
</tr>
<tr>
<td>Web-8</td>
<td>36.26</td>
<td>34</td>
</tr>
</tbody>
</table>

Figure 5. Comparison of Energy Consumption

5. CONCLUSION

In this paper, for speeding up the training time, we parallelized the SVM training using Pthread. After analyzing the computational characteristics of the SVM training, we exploited the medium-grained parallelism by distributing the workload to a multicore platform while satisfying the data dependencies. Our experimental results confirmed that the proposed parallel approach can provide a speedup of 2.5 (which is very close to the upper limit of a speedup computed by the Amdahl’s Law) using a 4-core processor. By parallelizing SVM training, we can also reduce the energy consumption of the sequential training by a factor of 1.5. Finally, the proposed parallel approach can provide the exactly same quality as the sequential SVM training.

6. ACKNOWLEDGEMENT

This research was financially supported by the Ministry of Education, Science Technology (MEST) and National Research Foundation of Korea (NRF) through the Human Resource Training Project for Regional Innovation.

7. REFERENCES


