Advances in Science and Technology
Research Journal

Volume 12, Issue 4, December 2018, pages 81–87
https://doi.org/10.12913/22998624/100341

Received: 2018.06.30
Revised: 2018.10.15
Accepted: 2018.11.05
Available online: 2018.12.15

Optimization of Machine Learning Process Using Parallel Computing

Michał K. Grzeszczyk[1]

[1] Faculty of Electronics and Information Technology, Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warsaw, Poland, e-mail: michal.k.grzeszczyk@gmail.com

ABSTRACT
The aim of this paper is to discuss the use of parallel computing in the supervised machine learning processes in order to reduce the computation time. This way of computing has gained popularity because sequential computing is often insufficient for large scale problems like complex simulations or real time tasks. After presenting the foundations of machine learning and neural network algorithms as well as three types of parallel models, the author briefly characterized the development of the experiments carried out and the results obtained. The experiments on image recognition, ran on five sets of empirical data, prove a significant reduction in calculation time compared to classical algorithms. At the end, possible directions of further research concerning parallel optimization of calculation time in the supervised perceptron learning processes were shortly outlined.

Keywords: parallel computing, machine learning, perceptron, neural networks, OpenMP

INTRODUCTION

This article touches two popular aspects of modern computer sciences. Nowadays, artificial intelligence (AI) is one of the fastest improving areas of information technologies. Many global companies are trying to outdo each other in implementing.

AI solutions are employed for everyday life tasks, e.g. Tesla is implementing algorithms for autonomous cars [19], while Amazon is working on a virtual assistant called Alexa (cloud-based voice service that allows you to use your voice to control smart devices, ask about real-time information and even talk with it) [15]. Google, apart from having its own Alexa (Google Assistant), works on various different projects; however, this is just a tip of an iceberg. There are many other important players in the game, such as Apple or Microsoft.

Machine learning (ML) gives us multiple possibilities of implementing the solutions that we were unheard of 50 years ago. Unfortunately, this approach requires high computational complexity, which usually results in long time of computing. This is the moment when Parallel Computing (PC) gives us a lot of opportunities. What is more, we can combine ML and PC and receive extremely powerful as well as fast tools, e.g. for data mining, image and pattern recognition [1].

In the range of intelligent systems and ML development, we have a typical task concerning running some perceptron algorithm with various parameters. After performing calculations, it often turns out that there are difficulties with obtaining proper results in a short time due to the high computational complexity. The genesis of the research undertaken and presented today is the attempt to simplify and shorten ML processes. In fact, a relatively simple improvement of the learning process is possible by applying PC in ML algorithm, which was the idea for this paper. Optimization will be shown on the example of perceptron algorithm and Open Multi-Processing (OpenMP) standard that allows building concurrency platforms for shared-memory parallel and multi-threaded processing. The software constructed using this standard is characterized by the following advantage: the process of converting a serial sequential source code into a parallel one is easy and only involves input of suitable compiler instructions for quick conversion to a parallel version of a program [9].

The aim of this paper is to discuss the use of PC in the supervised ML processes to reduce the computation time. PC has gained popularity because sequential computing is often insufficient for large scale problems like complex simulations or real time tasks. After presenting the foundations of ML and neural network algorithms as well as three types of parallel models (shared memory, distributed memory and hybrid model), the development of the experiments carried out (on 5 empirical data sets) and the obtained results were briefly characterized. At the end, possible directions of further research concerning parallel optimization of calculation time in the supervised perceptron learning processes were shortly outlined.

PERCEPTRON AND MACHINE LEARNING

Automatic learning is taxonomically divided into two groups: supervised and unsupervised learning. Supervised learning assumes that during the process of learning an algorithm receives a pair: sample data (specifically a vector) and expected output value (class to which the sample data should be classified) [18], an example of which is perceptron [16]. Then, based on the data samples and their desired value, the algorithm tries to create classifiers that would correctly classify unknown data sample. Unsupervised learning (often called clustering) is contrary to this model. It helps to find new, previously unknown common patterns in the data samples without classifying them beforehand.

As far as terminology is concerned, all feature vectors of the data samples are considered in representation space (E). Each data sample is identified by the class it belongs to ($C = \{0,1, ..., C\}$). When a new object that needs to be correctly classified appears, a classifier is needed ($G: E \rightarrow C$). The classifier is built on data samples and its goal is to classify data samples correctly – as many times as possible. Every classifier consists of C discriminant functions (DFs):

$$g_c : E \rightarrow R,\ 0 \leq c < C$$

Classification rule of each data sample (x) is as follows [6]:

$$\hat{c} = argmax\ g_c(x),\ 0 \leq c < C$$

The classifier is called linear when all its DFs are linear. Let x be a feature vector of a data sample ($x = (1, x_1, ..., x_D)t$) and a_c be the vector of coefficients of linear discriminant functions (LDF) of class c ($a_c = (a_{c0}, a_{c1}, ..., a_{cD})t$). Then:

$$g_c(x) = a_c^t x$$

Perceptron is the example of algorithm for building LDFs [7].

The main loop of the perceptron (line 3 in algorithm 1) iterates until convergence (all data samples are classified correctly) or until maximum number of iterations has been reached. The inner loop (line 5) iterates over all data samples. The class of each data sample is known, because perceptron is a supervised learning algorithm [8] (in that case the class of the data sample (x_n) is marked as c'). Then the value (*discriminantValue*) of the $g_{c'}$ applied to the x_n is counted. The innermost loop iterates over all classes. If the class (c) that is currently analysed is not the same as c' (line 10) and value of the g_c applied to x_n plus b (margin parameter) is greater than the *discriminantValue*, it means that a classification error occurred. In that case, we need to update g_c by subtracting from it the feature vector of s_x multiplied by α (learning rate parameter). If at least one classification error appeared, we need to update the weight vector of class c' as well (this time by adding αx_n). This is an example of a very simple neural network.

Perceptron has three parameters that are passed to it during execution: α (learning rate), b (margin) and K (maximum number of iterations). Analysing the influence of each of them of the result of the perceptron one may notice that the parameter α does not have an impact on the quality of the result, it only determines how fast weight vectors are changing (in general the greater α, the smaller number of iterations). This parameter should always be greater than 0. Data samples are linearly separable when there is a possibility of creating such weight vectors that each data sample is classified correctly.

The most important parameter is margin. For linearly separable data, the set $b = 0$ sometimes can be too small to create good quality of the results. However, if b is greater than 0 and large enough, decision regions for each class will be clearly separated. For non-linearly separable data set $b = 0$ does not guarantee quality result. For large K and large margin, good quality results can be obtained (only a few samples would be misclassified) [14].

The neural models based on perceptron have many significant features, including: non-parametrical and non-linear, resistant to distortion, fuzzy and noise of analysed images, easy in

```
1: $classificationErrors \leftarrow true$
2:                                                  ▷ $K \leftarrow$ maximum number of iterations
3: for ($k \leftarrow 0$; $k < K$ && $classificationErrors$; $k$++) do
4:     $classificationErrors \leftarrow false$
5:     for ($n \leftarrow 0$; $n < N$; $n$++) do       ▷ $N \leftarrow$ number of training samples
6:         $error \leftarrow false$
7:         $c' \leftarrow \mathrm{c}_n$
8:         $discriminantValue \leftarrow \mathrm{g}_{\mathrm{c}'}(\mathrm{x}_n)$
9:         for ($c \leftarrow 0$; $c < C$; $c$++) do              ▷ $C \leftarrow$ number of classes
10:            if ($c$ != $c'$) then
11:                if ($\mathrm{g}_\mathrm{c}(\mathrm{x}_n)$ + $b$ > $discriminantValue$) then
12:                    $error \leftarrow true$
13:                    $a_\mathrm{c} = a_\mathrm{c} - \alpha x_\mathrm{n}$
14:                end if
15:            end if
16:        end for
17:        if ($error$) then
18:            $a_\mathrm{c} = a_\mathrm{c} + \alpha x_\mathrm{n}$
19:            $classificationErrors \leftarrow true$
20:        end if
21:    end for
22: end for
```

Algorithm 1. Perceptron

computer (software and hardware) implementation and various applications, and useful in generalizing knowledge acquired from empirical data [11].

OPENMP IN PARALLEL COMPUTING

Contrary to traditional sequential computations, PC solves problems concurrently. Many classic algorithms can often be executed in parallel but that sometimes requires a change of the data structures or redesigning the algorithm. PC has gained popularity because sequential computing is often insufficient for large scale problems like complex simulations or real time tasks.

There are three types of parallel models: shared memory, distributed memory model and hybrid model. In the first one, each processor has access to the whole memory (single computers implement this concept through threads) [12], while in the second one, each processor has its own local memory space. In that case, processors have to communicate with each other through messages to cooperate, just like processes in the operating systems. Therefore, they are used for applying distributed memory model in a single computer. Sometimes, a hybrid model is implemented – different units working in shared memory model are integrated as if they were part of distributed memory model [13].

OpenMP is a standardised application programming interface (API) for shared memory parallel processing [4]. It is developed by most of the largest software and hardware companies such as IBM, Intel or Oracle. It supports most operating systems and languages such as: C, C++ or Fortran. The greatest advantages of OpenMP are: simplicity, portability and flexibility. This technology works in a fork-join model for multithreading [2]. During the execution of a parallel program, the master thread executes sequential parts and if a parallel region appears, the master thread populates slave threads and solves tasks with them in parallel. Then, all the threads are synchronized, and the master thread continues working on the sequential part, and the situation repeats. Because of the fact of using shared memory model mutual exclusion functionality is also provided in the standard.

The usage of OpenMP on a simple example is quite straightforward (alg. 2). In order to execute a loop in parallel, a special directive needs to be placed above the loop initialization, thanks to which different iterations will be distributed among the available threads. Using the pragma directive, one can easily customize the parameters of parallelization.

While designing the algorithm suitable for PC, it is worth bearing in mind one aspect of Bernstein's Conditions – it is possible to run two tasks in parallel if they are independent (there are no data dependencies between them). The conditions stating independency for two tasks (T_n and T_m executed one after another) are [3]:

1. Input T_m ∩ Output $T_n = \varnothing$ (no flow dependency)
2. Input T_n ∩ Output $T_m = \varnothing$ (no anti-dependency)
3. Output T_m ∩ Output $T_n = \varnothing$ (no output dependency)

If Bernstein's conditions are fulfilled, it is possible to run two tasks in parallel without the risk of race conditions.

EXPERIMENTS

While running perceptron, a decision about input parameters has to be made. However, to make the decision about parameters, the data samples have to be analysed. It is simpler and more efficient to run perceptron couple of times with parameters from different orders of magnitude and then to compare results (alg. 3). In this article, the parameter *K* will be considered as constant and equal 200. This value is reasonable with many examples. For other two parameters perceptron will be run many times with different values (a,b ∈{0.1, 1, 10, 100, 1000, 10000}).

The time of execution of the perceptron algorithm depends on the data set, for the examplary data sets, it does not exceed 10 seconds. However, because of running the program with different variations of parameters, the execution time increases dramatically. Computations need to be parallelized to decrease it. In order to decide whether a loop is able to be executed in parallel, it has to be analysed by deciding whether it fulfils Bernstein's Conditions or not. When the loops are nested, only one loop can be parallelized with OpenMP. The strategy is to choose the most outer loop because by that the best efficiency is acquired.

In the perceptron algorithm (alg. 1) only the innermost loop can be parallelized as there are no data dependencies inside it. Two other loops do not fulfil the first Bernstein's Condition since there are many flow dependencies, especially with all g_c variables. In the sequential algorithm, previous loop iterations recount (write) them, while the following read from them and re-count them. Parallelizing the most inner loop would not be efficient (specifically with few classes) due to the synchronization time of threads [17]. However, if perceptron is executed many times, different executions can be parallelized (Fig. 1). The necessary variables would be made private for each thread – both of the loops from alg. 3 can be parallelized. As stated earlier, the outer loop will be chosen, and results of each perceptron will be stored, in text files. For each execution final LDFs will be stored as well as number of misclassified samples from the training and testing data sets accordingly (training set is 70% of whole randomly shuffled data set, the rest is testing set). The comparison of results is not the aim of this paper.

The experiment was conducted on 5 data sets, courtesy of School of Informatics at Universitat Politecnica de Valencia:

- OCR14x14 – 1000 samples for digits recognition (196 features per sample),
- expressions – 225 facial expressions (4096 features per sample, 5 classes – surprise, happiness, sadness, anguish and displeasure),
- gauss2D – 4000 samples representing a bidimensional Gaussian distribution of 2 classes that are equally-probable,

```
#pragma_omp_parallel_for          ▷ directive for running a loop in parallel
for ($i \leftarrow 0$; $i < I$; i++ do
    <compute something>
end for
```

Algorithm 2. OpenMP example

```
for ($a \leftarrow 0.1$; $a \leq 10000$; $a = a * 10$) do
    for ($b \leftarrow 0.1$; $b \leq 10000$; $b = b * 10$) do
        perceptron(a,b,K,dataSamples)
    end for
end for
```

Algorithm 3. Perceptron multi-execution

- gender – 2836 facial expressions (1280 features per class) classified by gender,
- videos – 7985 basket and non-basket videos (2000 features per class) computed from local-feature histograms.

Perceptron was implemented in C language for compatibility with OpenMP. The experiments were performed on a computer characterized by: Ubuntu 16.04, Intel Core i7–6700HQ 2.6 GHz, 4 cores, 8 logical cores and 16 GB RAM.

RESULTS

The results achieved in the experiments are presented in Tab. 1. The parallelized loop was executed maximally with 6 threads, because it has only 6 iterations throughout the experiment. Running the algorithm with more than 6 threads would result in the situation that some of them would always stay idle and only take part in the synchronization of threads and increasing the execution time. The table presents times of execution of each example data set for 1 to 6 threads (p ∈ {1, ..., 6}). Decreasing time of execution for an increasing number of threads can be observed. In order to present more clearly the influence of parallelizing on the time of execution two additional parameters are presented (Speed-up and Efficiency).

Speed-up shows how the parallel algorithm gains the speed with respect to its sequential version. Here, it is counted as a ratio between time of execution with one thread and time of execution with p threads [20].

$$S(p) = \frac{t(1)}{t(p)}$$

Efficiency is a parameter denoting the usage of parallel units by an algorithm [5].

$$E(p) = \frac{S(p)}{p}$$

The ideal efficiency of 100% was unreachable, because of the unpredictable load distribution between threads, depending on randomly shuffled data samples and different times of convergence according to the parameters. It is important to notice the data access time – the more threads try to access data, the larger overhead with comparison to the sequential algorithm. Speed-up and efficiency for shorter times of executions of sequential algorithms (OCR14x14, gauss2D) were worse than the others because of the synchroni-

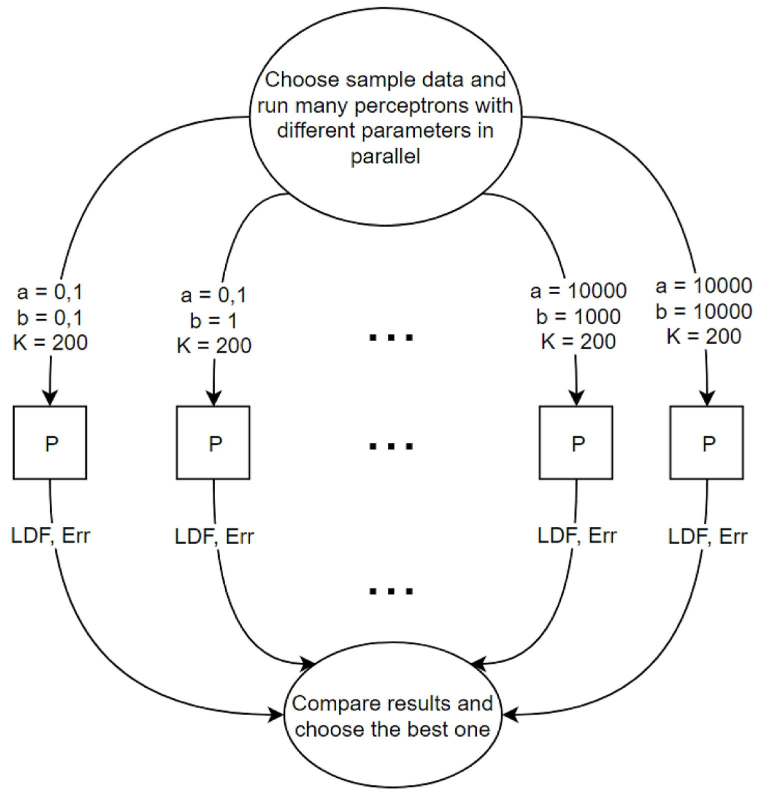

Fig. 1. Parallel computations of perceptrons

Table 1. Results from experiments

	OCR14x14	Expressions	Gauss2D	Gender	Videos
Samples (n)	1000	225	4000	2836	7985
Features	197	4096	2	1280	2000
Classes	10	5	2	2	2
t(1) (s)	6.135	25.819	0.070	43.206	657.495
t(2) (s)	5.434	13.923	0.068	22.176	394.441
t(3) (s)	4.534	9.738	0.061	15.335	272.166
t(4) (s)	4.739	9.736	0.060	15.333	277.438
t(5) (s)	4.532	9.676	0.060	14.919	269.351
t(6) (s)	3.007	6.928	0.040	10.740	188.172
S(6)	2.04	3.73	1.75	4.02	3.49
E(6) (%)	34.00	62.11	29.17	67.05	58.24

sation time of the threads which exposes itself more for shorter executions. For other example data sets, the results are more satisfying. Taking into account all of the above, the results achieved in the experiments can be interpreted as positive.

In Fig. 2 the normalized execution times of perceptron for different data sets are presented. It can be noticed that improvement in the time of execution appears when the number of iterations is divisible by the number of threads. In such situations, all of the threads were active. In the case of 4 and 5 threads, there is no possible distribution of 6 iterations among them without some of them remaining idle for some time. This results in the lack of improvement or even deterioration of the execution time due to the longer synchronization of threads. Perceptron obtained good quality results for each of the data sets. If the training samples were linearly separable, perceptron was converging for some of the executions with different parameters, which would have been chosen as the best ones. Otherwise, it was not converging; however, some of the results for each of the data sets were good for comparison and deciding which one to choose.

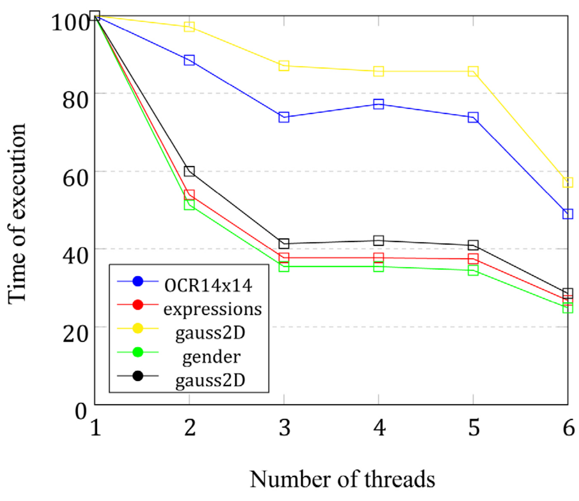

Fig. 2. Normalized execution times for different data sets

CONCLUSIONS

In this paper it was proven that PC can be efficiently used for the improvement of reaching good quality results of supervised learning. The experiments on image recognition, run on five sets of empirical data, clearly state a significant reduction in the calculation time, compared to classical algorithms. Future investigations can cover redesigning perceptron for application of PC with the aid of MPI protocol [10] for distributed memory systems. Many neural network algorithms are possible to parallelize, and extensive studies are being carried out in this field. By applying parallel optimization, better results can be achieved and new applications of neural networks for everyday life problems can be reached.

Acknowledgements

The author would like to thank Lecturers from School of Informatics at Universitat Politecnica de Valencia for sharing the knowledge and data sets necessary for writing this article.

REFERENCES

1. Abu-Aisheh Z., Raveaux R., Ramel J. -Y., Martineau P., A parallel graph edit distance algorithm Expert Systems with Applications, Volume 94, 15 March 2018, 41–57.

2. Akhter S., Roberts J., OpenMP: A Portable Solution for Threading. OpenMP provides an easy method for threading applications without burdening the programmer, 2010, http://drdobbs.com/high-performance-computing/225702895 (access: September 2018).
3. Bernstein A. J., Program Analysis for Parallel Processing, IEEE Transactions on Electronic Computers, EC-15, 1996, 757–762.
4. Bhugul A. M., International Journal of Computer Science and Mobile Computing, Vol. 6, Issue2, February, 2017, 90–94.
5. Colombet L., Desbat L., Speedup and efficiency of large-size applications on heterogeneous networks, Theoretical Computer Science, Volume 196, Issues 1–2, 6 April 1998, 31–44.
6. Devroye, L., Gyorfi, L., Lugosi, G., A probabilistic theory of pattern recognition. Springer: New York, 1996.
7. Duda R. O., Stork D. G., Hart P. E., Pattern Classification, New York: John Wiley & Sons 2001.
8. Freund, Y., Schapire, R. E., Large margin classification using the perceptron algorithm. Machine Learning. 37 (3), 1999, 277–296.
9. Głowacz A., Pietroń M., Implementation of Digital Watermarking Algorithms in Parallel Hardware Accelerators, International Journal of Parallel Programming, October 2017, Volume 45, Issue 5, 2017, 1108–1127.
10. Gropp W., Lusk E., Doss N., Skjellum A., A high-performance, portable implementation of the MPI message passing interface standard, Parallel Computing. Volume 22, Issue 6, September, 1996, 789–828.
11. Grzeszczyk T. A., Neural Networks Usage in the Evaluation of European Union Cofinanced Projects, Foundations of Management, Volume 2, Issue 1, 2010, 7–20.
12. Inderpal S., Review on parallel and distributed computing. Scholars Journal of Engineering and Technology, 1(4), 2013, 218–25.
13. Kang S. J., Lee S. Y., Lee K. M., Performance comparison of OpenMP, MPI, and MapReduce in practical problems, Adv. Multimedia, 2015, 1–9.
14. Mohri M., Rostamizadeh A., Perceptron Mistake Bounds, 2013, arXiv preprint, https://arxiv.org/abs/1305.0208 (access: September 2018).
15. Moore R. K. Nicolao M., Toward a Needs-Based Architecture for ‘Intelligent’ Communicative Agents: Speaking with Intention, Frontiers in Robotics and AI, Volume: 4, Article Number: 66, Dec 2017.
16. Rosenblatt, F., The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, 65(6), 1958, 386–408.
17. Russo A., Sabelfeld, A., Securing Interaction between Threads and the Scheduler in the Presence of Synchronization, The Journal of Logic and Algebraic Programming, Volume 78, Issue 7, August–September 2009, 593–618.
18. Sathya, R. and Abraham, A., Comparison of Supervised and Unsupervised Learning Algorithms for Pattern Classification. International Journal of Advanced Research in Artificial Intelligence, 2, 2013, 34–38.
19. Stilgoe J., Machine learning, social learning and the governance of self-driving cars, Social Studies of Science, February 2018, Volume: 48, Issue: 1, 25–56.
20. Xian-He Sun, Lionel M. Ni, Another view on parallel speedup, Proceedings of the 1990 ACM/IEEE conference on Supercomputing, October 1990, New York, USA, 324–333.