Bootstrapping: A parallel implementation with OpenMP

María José Núñez-Ruiz

Universidad Austral de Chile

Abstract. An adaptation of the Bootstrapping algorithm was carried out using the OpenMP programming model. Some performance measures such as execution time, acceleration and efficiency were evaluated. In addition, a comparison was made with its sequential version. All this with the aim of knowing how the performance measures are altered when executing the algorithm with different sample sizes and threads until the optimal ones are found.

Keywords: Bootstrapping · Parallelism · Performance.

1 Introduction

The data samples with which statistical analyzes are carried out are every day larger. In some cases, the processors in charge of doing work on this data crash, either due to memory capacity or run time. This is where parallel algorithms become important. These serve to accelerate the resolution of high-cost problems and decrease the use of computer resources.

Bootstrapping, like other resampling methods, is a tool used to understand sampling distributions in modern statistics. You specify some parameter to estimate, or some model to fit, bootstrapping is the act of repeatedly drawing samples from a data sample and readjusting or re-measuring an estimator on each new sample [4]. There are even algorithms that have been based on this method and have achieved great efficiency results, such as the Bags Little Bootstrap (BLB) algorithm [1].

The main idea of Bootstrapping is to treat a sample from the population of interest as an empirical distribution of the data. Then, the sampling distribution of that statistic can be estimated by bootstrapping from the sample. If n is the size of the original sample, in bootstrapping, one draws r resamples with replacement of size n from the original sample. Then, the statistic of interest is computed from each resample, and all r statistics are aggregated to form a bootstrap distribution which approximates the theoretical sampling distribution of the statistic [1]. A relatively new procedure that combines the characteristics of bootstrapping and subsampling in a resampling method that can accompany massive data sets, while maintaining the same bootstrapping power [2].

Bootstrapping is only performed on each subset, the computational cost is favorably rescaled to the size of each subset. This structure is ideal for parallel computing architectures that are often used to process large data sets by dividing a process into parts that will run simultaneously on different processors.

In the field of Artificial Intelligence Bootstrapping is used to combine classification models and thus obtain a more precise and exact result. In this study, a simplified implementation of the Bootstrapping algorithm (without replacement and without repetition) was developed. This algorithm was adapted to be executed in parallel, in order to know how the performance measures vary from one version to another and using different sizes of samples and execution threads. To achieve the objective of the study, the following two research questions were established:

RQ1: How do the performance measures vary using different numbers of samples and threads?

RQ2: How do the performance measures vary according to the number of threads when the sample is large enough?

2 Methodology

To carry out this work, the algorithm was implemented in its sequential and parallel versions in the C programming language. The algorithm was paralyzed using the OpenMP programming model. The main idea was to evaluate different performance measures such as execution time, acceleration and efficiency with different sample measures and processors.

The structure of the Bootstrapping algorithm that was implemented in this study can be seen in Fig. 1. In the first place, a data segmentation process is carried out, from where the subsamples are obtained, for which the number of subsamples and the data set to be analyzed must be defined. Then, for each subsample the estimated statistical calculation is performed. Finally, these results are averaged in a reduce process. This structure facilitates parallelism mechanisms. On the other hand, we know more than when using these strategies, map and reduce, our algorithm is reduced in time going from O(n) to O(n/p) and O(log (n)) respectively.

On the basis of the structure seen above, a sequential and parallel version of the algorithm was developed in the C programming language. This language allows us to integrate parallel regions to the code. First, the Map method was used to segment and calculate in parallel the values of the variances of each subset and then with the Reduce method the sum of these results was calculated. This sum was averaged across segments. This implementation was restricted in such a way that each thread took only one segment, so the sample must always be greater than the number of threads and must also be a power of 2. The analyzed sample was stored in an array of random data.

The performance tests were done with different sample sizes ranging from 2 ** 12 to 2 ** 23 bytes and different amounts of threads, which also correspond to the subsamples, in this case 2, 4, 8 and 16 threads of execution. The execution time was measured. In the case of a parallel program, the execution time is the time that elapses from the beginning of the program execution in the parallel

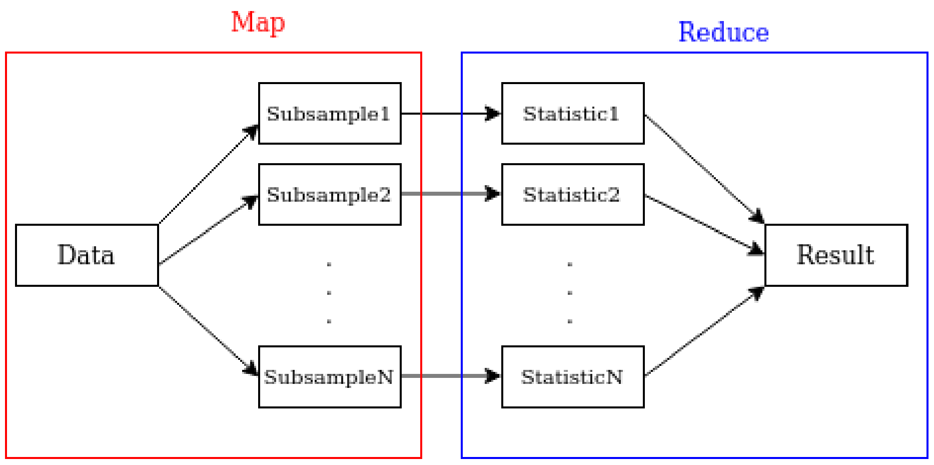

Fig. 1. Structure of the logic that follows the Bootstrapping algorithm.

system until the last processor completes its execution. In addition, speed gain and acceleration (Speedup) were measured. The Speedup is the quotient between the execution time of a sequential program and the execution time of the parallel version of said program. On the other hand and to finalize the analysis, the efficiency of the algorithm was also measured. The results are shown in the next section.

3 Results

3.1 Performance

To calculate the execution time both sequentially and in parallel, it is necessary to calculate the time at the beginning of the process and then at the end, the subtraction of these being the final result.
The execution time for either the sequential or parallel algorithm is calculated with the following mathematical equation

$$T = Tf - Ti \tag{1}$$

The result for each test can be seen in Fig. 2. In it we can see that for relatively small sample sizes, the behaviors between the sequential and parallel algorithm, independent of the execution threads, remain without many differences over time, this in turn is a value that does not exceed the 0.1 seconds.

The change occurs and begins to be noticeable in 2**18. With this sample size we see how the execution time of the sequential algorithm is notably impaired as the sample size grows. It even looks like it increases exponentially, reaching a running time of more than 0.5 seconds.

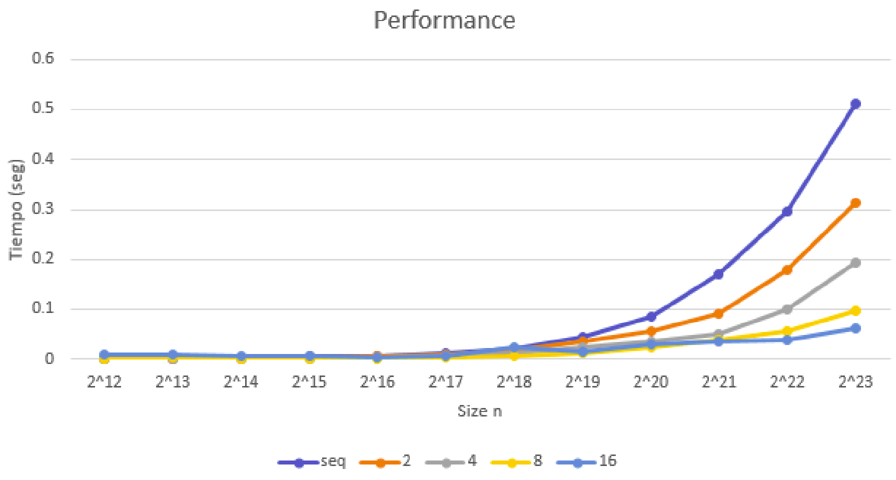

Fig. 2. Results of the tests corresponding to the execution time.

Regarding its parallel version, it can be seen that by increasing the sample size and occupying more execution threads, the algorithm is much faster than in sequential mode. The algorithm performs very well when the sample is large and 16 threads are used. Along with its execution with 8 threads, this version has a good performance, its times do not exceed 0.1 seconds.

3.2 Speedup

Speedup is the improvement in the execution speed of a task executed in two similar architectures with different resources. It can be used more generally to show the effect on performance after any resource enhancement. In this case study, it was tested with different execution threads and sample sizes, ranging from 2**12 to 2**23 bytes, and a last size 2**30 was added to see how considerable it was the change for that type of size and thus to be able to make the corresponding comparisons to its growth or decay.
The acceleration for a parallel algorithm is calculated with the following mathematical equation. Where Ti is the time at the beginning of the execution and Tf is the final time.

$$S = Ts/Tp \tag{2}$$

When observing the results shown in Fig. 3 on the acceleration or speed gain of the execution of the algorithms with their different values of execution threads and sample sizes. As a result, we can conclude that with 2 threads the algorithm does not acquire many changes, it is kept at a constant speed over time, for 4 threads its maximum speed is obtained by calculating the variance for a sample size of 2**16 bytes. As for the execution with 8 threads, its maximum speed is boosted with 2**15 and 2**17, falling by 2**16. With 16 threads, the

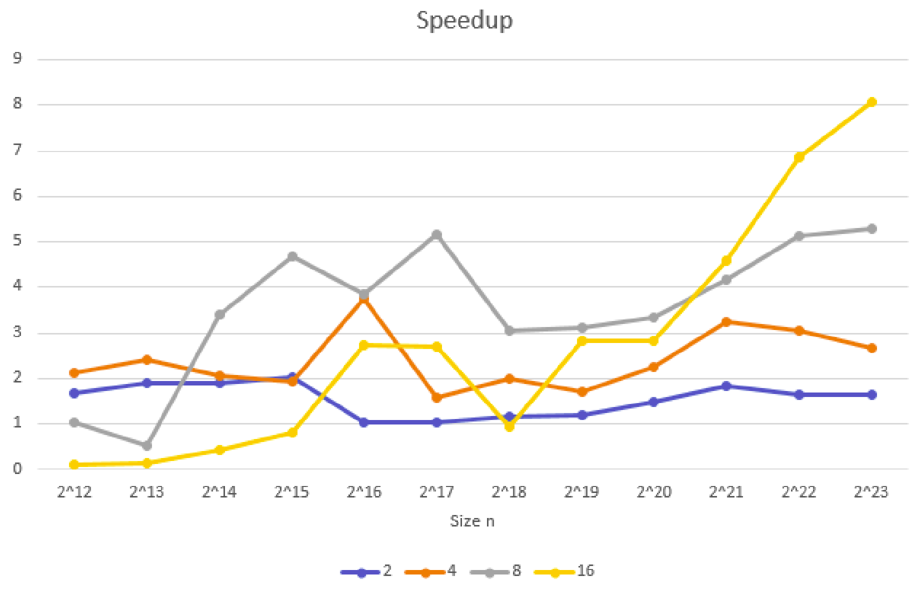

Fig. 3. Results of test for speedup gain over time.

maximum speed acquired is observed when analyzing a sample of 2**23. Tests were also carried out for a sample of 2**30 bytes, where it can be seen that with 8 threads it reaches a considerably high speed, however, with 16 threads, the speed is higher than all, being 14 times better than in its traditional execution.

3.3 Efficiency

Efficiency is defined as the ratio of speed to the number of processors. Efficiency measures the fraction of time for which a processor is usefully used. It corresponds to a normalized value of the speedup (between 0 and 1), with respect to the number of processors used. Efficiency values close to one will identify almost ideal performance improvement situations.
The efficiency of a parallel algorithm is calculated with the following mathematical equation. Where S is the speedup previously calculated and p is the number of thread to which said calculation corresponds.

$$E = S/p \tag{3}$$

Having said the above and as we can see in Fig. 4 with 2 threads, there is a substantial improvement when executing the algorithm for 2**15 and then 2**2. These being the highest levels of efficiency. As for threads 4 and 8, their best performance regarding efficiency is seen in 2**16 and 2**17 respectively. For 16 threads, your best mark is 2**23.

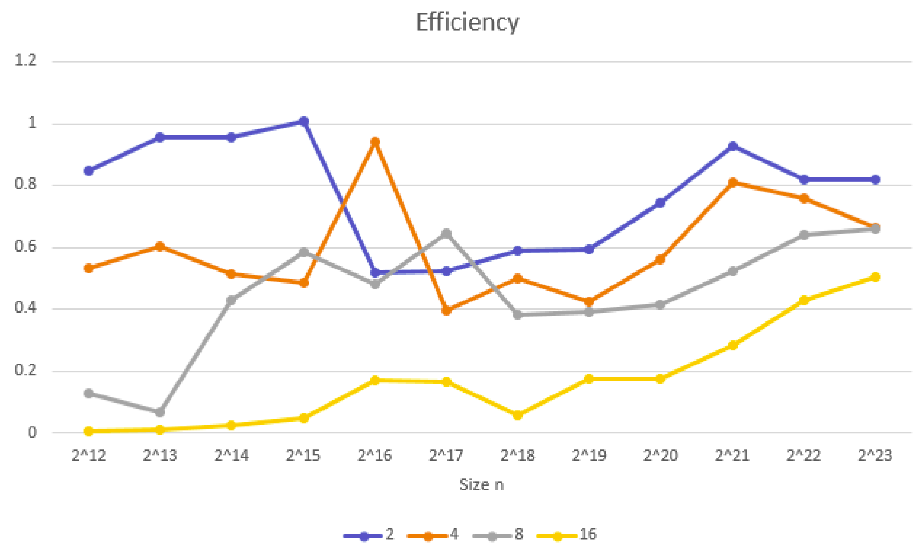

Fig. 4. Results of test for efficiency over time

With the above, we have answered the first research question **How do the performance measures vary using different numbers of samples and threads?**. We can then say that the parallel algorithm has better performance than the sequence algorithm. Segmenting the sample, calculating the statistic and then averaging those results within a parallel algorithm allows to have the result in a much shorter time as the sample grows, however this is also related to the number of processors that carry out this work.

Additionally and to answer the question **How do the performance measures vary according to the number of threads when the sample is large enough?**, acceleration and efficiency were measured for a sample large enough size equal to 2 ** 30 bytes. The results can be seen in the figures Fig. 5 and Fig. 6. It can be seen how for each thread its values regarding the acceleration increase as more threads are occupied. On the other hand, we can see that for thread 2 a higher efficiency is obtained.

4 Conclusions

An important contribution of this analysis is to propose a parallel implementation of the well-known Bootstrapping algorithm and the corresponding performance tests to demonstrate its efficiency. This version of the algorithm did not include replacement sampling or repetitions, features that have been included in other developments. It can be seen as a simple algorithm with which it was mainly sought to demonstrate the differences between its sequential and parallel

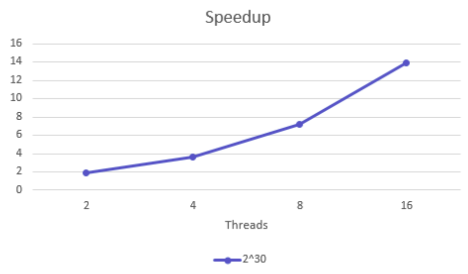

Fig. 5. Speed gain with a sufficiently large sample (2**30)

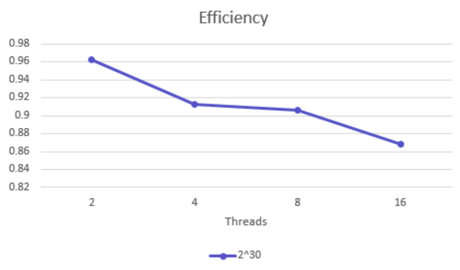

Fig. 6. Efficiency with a sufficiently large sample (2**30)

implementation and the variants of the latter regarding the size of the sample and the number of processors during its execution.

Based on the results shown above, one can appreciate the great potential that parallel programming has compared to the sequential approach. Communication and synchronization between different sub-tasks are some of the biggest obstacles to getting good performance from a parallel program. However, here the objective was achieved and the research questions could be answered satisfactorily since the Bootstrapping algorithm structure allows it to be paralyzed without major difficulties.

In a future version it should be implemented with replacement sampling to fulfill everything that is expected of a Bootstrapping algorithm, perhaps using a library. With this, it was possible to verify that there is indeed an improvement in the performance of the sequential version to the parallel one.

Regarding the high variations reflected in the Speedup graph, there is no accurate explanation of what could have happened. In a future analysis, a greater number of tests should be carried out and those results compared.

References

1. Zhang, Y.: Integrating Random Forests into the Bag of Little Bootstraps. (2017)
2. Kleiner, A., Talwalkar, A., Sarkar, P., and Jordan, M. I.: A scalable bootstrap for massive data. Journal of the Royal Statistical Society: Series B (Statistical Methodology), 76:795–816 (2014)
3. Prasad, A., Howard, D., Kamil, S., Fox, A. (2012). Parallel high performance bootstrapping in python. Proc. Eleventh Annual Scientific Computing with Python.
4. De Rose, C. A., Fernandes, P., Lima, A. M., Sales, A., Webber, T. (2011, May). Exploiting multi-core architectures in clusters for enhancing the performance of the parallel bootstrap simulation algorithm. In 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (pp. 1442-1451). IEEE.