Implementation and Performance Analysis of a Parallelized Solution for *Storms of High-energy Particles*

Gonçalo Moura, *Student, 52361,* Gonçalo Querido, *Student, 45616,* and Renato Oliveira, *Student, 52393*

Abstract—In this report we will approach topics regarding the parallelization of a problem concerning a programming code that simulates high-energy particles bombardment on a surface. Through a given code, this report is focused on analysing the effect of the speed up and accuracy of a parallelized code compared to its sequential version, using the application programming interface *OpenMP*, that allows for a multithreading implementation, and using the *Instruments* Profiler to measure memory/resource usage during the execution to evaluate where better to act. To overcome the parallelization problems found, the solution will be based on the slides given to us on the theoretical and practical classes, having always in regard the bibliography suggested for study. [2] [3]

1 Introduction

This report will be focused on the implementation of a possible parallelized solution regarding the problem of *Storms of High-energy Particles*. This solution will be used to compare and analyze performance and speed-up results against its sequential version.

During the report, every bit of code that was thought to be beneficial to be parallelized will be discussed, giving a theoretical context on why it was decided, supported by tested results. We will discuss the implementation strategy as well as all the problems had in its execution. To measure the amount of resources used and to analyze the code *hotspots*, we resorted to a *profiler*, and in this report we chose *Instruments* for *MAC OS* for this purpose. Due to running errors later encountered, our results from comparing the sequential version to the pararellized one were measured from a machine with *Linux OS*.

The *OpenMP* API (aplication programming interface) was used for this implementation as specified in the project requirements, allowing for a simple and flexible interface for programs that require parallelism within a multi-core node, allowing for an implementation of multithreading.

To proceed to a proper analysis, assuring there are as fewer changes as possible, all tests were run in the same machine. Most of them were run forty times using each solution (forty times the sequential code and forty times the parallelized code) while others, more specifically tests seven, eight and nine, were only run twenty-five times due to their bigger execution runtime. The results discussion will be based around test seven due to it being big enough to compare possible solutions.

2 OpenMP API

OpenMP is a parallel programming model for multiprocessor architectures, providing a portable and scalable model for developers of shared memory parallel applications. It allows programmers to set the maximum number of threads a program can create to run in parallel within the bit of code that is marked accordingly, running that section independently. [1] At the end, the threads then join back into the primary one, continuing to the end of the program. It gives access to an extensive library specialized in dealing with threads for parallelization purposes.

3 Profiler

In order to discover CPU activity variation and memory allocation for each step of the code, we used a profiler to display the events occurring in the application. The one chosen to discover what was needed was *Instruments* (formerly known as *Xray*), for *MAC OS*. [4]

By using this profiler, we discovered that, by running the *.omp* file, it uses 20.3 percent of the computer processor. On the same information, we can see that the update function present in the given code takes 18.4 percent, as seen in *Fig. 1*. By doing the math, we can conclude that the update function ends up weighting as much as 90.64 percent on the processor.

With this information, we can conclude that most of the processing power goes to the *update* function. With this information, we can claim that this function is a discovered

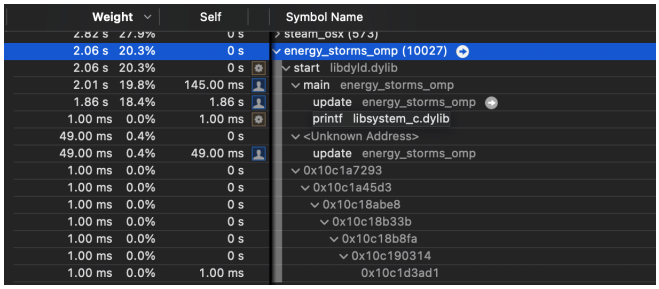

Fig. 1. *Output of the Profiler Instruments after the first excecution*

• *All three students were enrolled at NOVA School of Science and Technology, Department of Computer Science.*
E-mails: gb.moura@campus.fct.unl.pt; gq.sousa@campus.fct.unl.pt; rjs.oliveira@campus.fct.unl.pt

hotspot, so it his in our interest to parallelize the parts of the code that invoke this function in order the improve the execution performance.

4 Parallelized Code

In this section we will talk about the parts of the code we thought would improve performance by applying parallelization, as present in the *.omp* file, referencing the sections by the name of the method and the number of the block in that method (if there is one, if not it will be specified by the line number).
As discussed before, we were able to find the *hotspot* through the analysis of profiler's result, and therefore we want to try to parallelize all the situations where the function *update* is invoked and check for results and possible improvements on performance.

4.1 Parallelizing the *update* Function

A simple parallelization was aplied to the function *for* cycle, using, as mentioned before, the *OpenMP* library. After this implementation we proceeded in analysing the next blocks of code to see if it was worth to parallelize. For example, we analyzed block 4.2, that updates the arrays *layer* and *layer_copy* using basic operations, not being necessary to apply a parallelization.

4.2 Block 4.3

By analysing this block, we came across a considerable problem: by applying parallelization on this section and updating the values, they were being printed with incorrect values. This happened due to the fact that several threads were trying to access the same position of the vector, and changing the same value each time a thread accessed one position. To overcome this issue, three different approaches were tried:

- using the sequential function;
- using the *atomic* function from the *OpenMP* library;
- using the *critical* function also from the *OpenMP* library.

test_01		
#runs	Sequential	Parallel
1	0,000037	0,001293
2	0,000037	0,001543
3	0,000037	0,001916
4	0,000038	0,002048
5	0,000038	0,002172
6	0,000038	0,002211
7	0,000038	0,002229
8	0,000038	0,002266
9	0,000038	0,002274
10	0,000038	0,002316
Average Time	0,0000377	0,0020268
Average Speedup	0,01860075	

Fig. 2. *Time and speed up results upon running the Test 1 sequentially and parallelized with sixteen threads running.*

#runs	Sequential	Parallel	Atomic	Critical
1	104,620956	15,904412	16,413855	16,393815
2	104,648702	15,929889	16,585601	16,599968
3	104,838325	15,974921	16,589466	16,630719
4	105,218778	16,030422	16,640356	16,631057
5	105,278465	16,133449	16,649018	16,639079
6	105,310193	16,198853	16,653677	16,647073
7	104,734859	16,222402	16,655781	16,652567
8	104,968545	16,286194	16,673428	16,660705
9	105,288856	16,322218	16,67767	16,680819
10	105,34579	16,579771	16,679659	16,682485
11	105,387294	16,61959	16,684872	16,703575
12	105,388964	16,754182	16,688614	16,726603
13	105,527028	16,87299	16,694646	17,740797
14	106,804677	16,873629	16,710067	16,743559
15	111,795377	16,884132	16,712973	16,747918
Average Time	105,6771206	16,37247027	16,6473122	16,7253826
Average Speedup		6,454561766	6,347998964	6,318367904

Fig. 3. *Results obtained by running test_07 fifteen times using all four possible solutions.*

4.2.1 Sequential Function

Although it is not an implemented solution, it is important to analyse the performance and speed up values the sequential version can produce.
As seen on *Fig. 2*, when running the code using the test file *test_01*, the sequential version has a considerable lower average time of execution when compared to its parallel counterpart, having no speedup at all.
Although one might assume that if this happened with this test case, then a sequential version of the code might be better than any form of parallelization for this problem. However, we need to have in consideration the size of the test case. *test_01* is very small compared to a real situation, or even the other test cases, so it cannot serve as a reference. It is only safe to say that, for an array with so few positions filled, it is not worth to parallelize on this problem.

4.2.2 Atomic Function

The usage of *OpenMP's* atomic operation allows a multiple threads to update a shared numeric variable. [5]
At first this implementation was slightly faster than the other two enumerated, and was decided to keep it parallelized with this function. Although, after running several test using a different number of threads, we arrived at the conclusion that it was not a viable solution, because, as happened in the initial try, some values were being returned incorrectly. This could be because of the fact that the *atomic* function only just be able to perform atomic operations to update variables, conflicting with the verification to determine the maximum of the array of values.

4.2.3 Critical Function

Using *OpenMP's* critical function allows to set a section of the code as critical, meaning that only one thread at a time can enter that bit. Only when a thread is finished running the delimited section, can the next thread run it. [5]
Compared to the atomic implementation, by being able to synchronize the access to blocks by different threads, the accuracy errors discovered above mentioned were solved, at the cost of decreased performance.

test_07				
#runs	Sequential	Parallel	Parallel atomic	Parallel Critical
Average Time	106,5200749	16,99159875	16,82405853	16,85246043
Average Speedup		6,268984838	6,331413718	6,320743218

Fig. 4. *Final results after running test seven forty times*

4.3 Result comparison

After implementing the best found solution, resorting to *OpenMP's* critical operation, maintaining the needed accuracy, we proceeded to run the tests with different numbers of threads. To analyse the values in the best way possible, we run the the code with two, four, eight, sixteen and eighteen threads.

In *Fig. 3* we present the results obtained by running *test_07* with sixteen threads using the sequential, parallel (with simple *.omp* techniques), *Atomic* and *Critical* functions. To facilitate the result's presentation, of the forty tests run only the first fifteen are shown. We can interpret that either the *atomic* solution or the *critical* promising, whereas the parallel perform worse. Even though the parallel version has a promising start, its execution time increases considerably with each iteration, worsening test after test.

Looking at *Fig. 4*, we can observe the final results of running the *test_07* using all four solutions. Here we can see that the solution with highest speedup and the best average time after the forty tests is the one using the *atomic* approach, although, as discussed in a previous topic, it fails in accuracy. On the other hand, the *critical* approach has nearly one hundred per cent accuracy, and the cost of performance versus accuracy is something we can bear, as the performance and average speedup is very similar.

Regarding the number of threads and how it affects performance, we can conclude by looking at *Fig. 5* that, as we increase the number of threads, the execution time of our program decreased. We can observe this phenomenon happen on up to sixteen threads. After this number, the execution time starts rising, being the performance cap at sixteen threads.

We associate this behaviour to the fact that the machine used to run the tests was running on a *WSL* environment, enabling to run native *Linux* command-line tools directly on Windows. This *WSL* environment was set to sixteen logical cores, limiting the recognized cores to run this implementation, being consequently limited in terms of efficiency to sixteen threads.

5 Cluster

To provide us with further data to analyze our solution, we ran *test_07* with our parallelized solution on the *cluster* with one wave file, two wave files and three wave files. The performance values returned are present on *Fig. 6*.

By analyzing the data, we can see that the execution times in the *cluster* compared to the local machine used are considerably different, having a large gap between the average time each of them takes to run each wave.

In order to better understand why these values are so apart from each other, we decided to check the *cluster specifications* and compare them to our local machine. Even though,

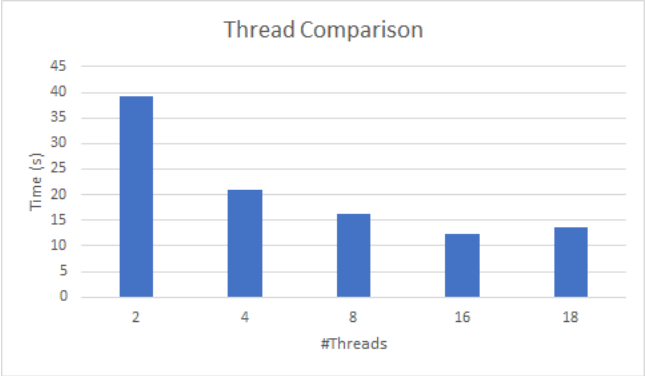

Fig. 5. *Thread comparison while running test_07 with a parallelized solution.*

Cluster_test_07						
	Cluster			Local Machine		
#runs	1 wave file	2 wave file	3 wave file	1 wave file	2 wave file	3 wave file
1	11,860348	24,085868	35,697923	4,878854	10,036967	16,393815
2	11,870675	24,840164	35,708337	4,920388	10,046702	16,599968
3	11,872644	24,018571	35,769626	4,941397	10,063478	16,630719
4	11,987873	24,160421	35,780639	5,010322	10,084389	16,631057
5	11,991682	24,840164	35,816328	5,042348	10,171656	16,639079
6	12,019996	23,884023	35,952182	5,075184	10,183222	16,647073
7	12,031482	23,98682	35,996258	5,11055	10,203141	16,652567
8	12,034881	24,031691	36,059167	5,116288	10,206423	16,660705
9	12,091997	24,119154	36,260199	5,12381	10,20837	16,680819
10	14,410291	24,016759	37,349294	5,126636	10,214676	16,682485
Average Time	12,2171869	24,1983635	36,0389953	5,0345777	10,1419024	16,6218287

Fig. 6. *Comparison between cluster and local machine while running test_07 with a parallelized solution.*

after checking, we could not conclude anything that would interfere so much with the performance, as the number of *cores* in the cluster are the same as the local machine.

What we suspect might have some impact in these values is the CPU of each machine.

6 Conclusion

To parallelize a program it is very important to find the *hotspots* and dependencies present in the code, as it gives us an insight on where to start analyzing. By discovering where most of the processing power goes to, we were certain that, if it could be parallelized, the performance could be improved.

According to the results obtained, the parallelization was, in most cases, successful, leading to a decrease in execution time compared to the given sequential version.

We believe better results could be obtained by discovering other relevant bits of code where parallelization could be applied. We do not take this as a perfect solution for the whole problem, but as a considerable improvement in situations where large amount of data are present.

Acknowledgments

We would like to thank Pedro Dimas, student number *52919* for providing us with *script* that allowed us to run a test multiple times. We would also like to thank *Group 24* for being part of the brainstorming on how to approach this project's parallelization problem.

Individual Contributions

The work was divided in the following form: Gonçalo Moura was the main responsible for the code parallelization (33%), helped by Renato Oliveira. Renato was responsible by the testing fase (33%), helped by Gonçalo Moura, both in his machine and in the cluster. Gonçalo Querido was the main responsible for the report and analysing the data (33%), where in this last part he was supported by his fellow group members who also gave him the graphs. Since we were able to be physically together it the last days, we do believe we all worked the same amount, helping in everything we could, and so splitting the relative contribution evenly.

References

[1] R. Gonçalves, *Paralelização de Aplicações com OpenMP*, Revista PROGRAMAR, 2014.
[2] M. Raynal, *Concurrent Programming: Algorithms, Principles, and Foundations*, Springer, 2013.
[3] M. McCool, A. Robison and J. Reinders, *Structured Parallel Programming*, Elsevier, 2012.
[4] https://docs.elementscompiler.com/Platforms/Cocoa/Instruments/
[5] http://portal.nacad.ufrj.br/online/intel/advisor2017/help/index.htm#GUID-BFE5F55E-A3BD-4440-B285-8FBF6A53A30E.htm