Storms of High-Energy Particles

João Soares *(jfc.soares@campus.fct.unl.pt)*, Martim Figueiredo *(mc.figueiredo@campus.fct.unl.pt)*, and Rúben Barreiro *(r.barreiro@campus.fct.unl.pt)*

Abstract—This was an assignment used in a Concurrency and Parallelism course, targeting the basic concepts of shared-memory programming with OpenMP. This assignment is based on a highly simplified simulation of the impact of high-energy particle storms in an exposed surface to test the robustness of space-vessels material. This project was adapted from the *EduHPC Workshop at SC18*.

Index Terms—Computer Science, Parallel Computing, High-Performance Computing, OpenMP, Shared-Memory, Hardware.

1 Introduction

The purpose of this report is to highlight the process of task and data analysis that must be considered, when parallelizing any sequential program, with multi-threading.

This project is mainly based on the simulation of the real-world problem of Storms of High-Energy Particles [1].

Our task was to parallelize a given sequential version of a program that simulates the effects of the bombardment of High-Energy Particles in a Storm, on an exposed surface.

2 Methodology

For this project, the methodology followed was based on some profiling analysis and parallelization techniques.

2.1 Initial Profiling Analysis

The first step was to analyze the problem, and inspect the respective source code given for the sequential version.

For a profiling analysis, the `Valgrind` [2] tool was used to acquire a better understanding of which of the parts of the sequential version consumed a bigger share of the execution time, providing clues as to which areas could be improved.

Then, the main focus shifted to the identification of hotspots, as possible candidates for parallelizations, through the inspection of the source code of the sequential version.

2.1.1 Valgrind Profiling

When using the `Valgrind` [2] tool for several of the given test files initially provided, it was verified that most of the sequential version's execution time was mainly being spent in three different tasks of its source code, such as:

- Memory allocations (the `mallocs` for the `layer` and `layer_copy` arrays) - [$10\% - 20\%$ of exec. time];
- `sqrtf` math function - [$10\% - 20\%$ of exec. time];
- Calls for `update` method - [$40\% - 85\%$ of exec. time];

2.1.2 Expectations for possible Improvements

From the earlier analysis and with some additional research, it was expected that the `sqrtf` function could be replaced by a faster one, by sacrificing a little of its precision. [3] However, since the main goal of this project was not code optimisation, but just its efficient and correct parallelization, we decided to not pursue this optimisation issue.

Additionally, from the research made about possible replacements for the `sqrtf` function, we also investigated some compilation flags for faster and optimised standard implementations of the mathematical functions, such as `-O3` and `ffast-math`, resulting, indeed, in much better performances for both sequential and parallel versions. [4]

In instance, it was assumed that the execution time of this mathematical function, as well as the `mallocs` calls, would always be constant, over the progress of this project, since they are always necessary, without any chance of improvement, in terms of the overall execution time.

Therefore, from then on, the focus shifted specifically to improvements that could be made to the `update` method and its subsequent computations of the maximums for each storm, identifying blocks of the sequential version's code as candidates for parallelization and efficiency optimisation.

2.2 Techniques and Approaches for Parallelization

Since the `OpenMP` [5] is a programming model designed for shared-memory, most of its parallelizations are achieved by using the `parallel for` directives, reason why it is necessary to find the right hotspots for the parallel loops.

2.2.1 Code Improvements and Hotspots for Parallelization

By inspecting the source code of the sequential version, it was immediately identified that several blocks of code were suitable for parallelization, such as the the following loops:

- Initialization of the `layer` and `layer_copy` arrays;
- Copies of the `layer` array to the `layer_copy` array;

Another observation made by inspecting the code, was the fact that the resources for the `layer` and `layer_copy` were never freed, similarly to what already happened with the `storms` array, at the end of the simulation's source code.
Initially, when thinking about the parallelization of the code, the code was split into three different scenarios:

- A single storm to be processed, with all parallel work shifted to the loop that processes the particles;
- Several storms to be processed, where the number of storms is lower than the number of threads given, with the parallel work divided first for the loop of processing the storms, and then, for the loop responsible for processing the particles;

- Several storms to be processed, where the number of storms is greater or equal to the number of threads given, with all the parallel workload based on the loop that processes the storms;

2.2.2 Parallelization of the Outer Loop for Storms

After successive tests were performed, it was also verified that, if there is more than one storm to be processed, then, most of the results of the simulation, in parallel, do not match the ones from the execution of the sequential version.

This is due to the fact that, for more than one storm, the simulation acts as an "accumulator of maximums", taking into consideration the maximums of the previous storms. Thus, parallelizing that section of the code would give rise to data races and race conditions, since it could not ensure the correct ordering of the storms' processing.

As a solution for this situation, the alternative was to only parallelize the loop responsible for the processing of particles, instead of the one for the processing of storms. In fact, this was one of the solutions initially considered, but discarded, since in most of cases, it is better to parallelize the outer loops (i.e., the ones responsible for processing the storms), instead of the inner loops (i.e., the ones responsible for the processing of particles). However, it is important to be aware that this process may add some overhead, increasing the execution times, in tenth of microseconds. [6]

At a later stage, it was verified that, if the outer loop has a much smaller workload than the inner loop, then the overhead added in the inner loop would compensate.

Thus, after we theorized about the potential effectiveness of the parallelization of the storm processing loop, it was concluded that it would not be worth the effort, since it is never expected to be provided a huge number of storms for the simulation, and the number of particles will always be significantly greater, representing most of the workload.

Therefore, it was decided that the best solution would be to parallelize only the processing of the particles, for each storm individually, executing in a sequential fashion. This way, instead of having a team of threads responsible for the computation of the storms' maximums, we would have a team of threads executing the `update` method and its respective particles' impact on the `layer` array.

2.2.3 Parallelization of the Inner Loop for Particles

From the observations made, using the first skeleton for the parallel version, the three previous scenarios mentioned in 2.2.2, were reduced to a single scenario, by considering only the parallelization of the processing of the particles.

In order to parallelize the inner loop for the processing of the particles, it was necessary to compute, in advance, the work that each thread would have, in order to ensure that the global workload would be well distributed amongst all of the threads, aiming to ensure a good work-balance.

For this purpose, we divided the number of particles of the current storm by the number of threads given. As it may be possible for some particles to be left after the division of work (i.e., if the number of particles is not a multiple of the number of threads), it was decided that the last worker thread would also be responsible for the processing of the remaining number of particles.

The number of these remaining particles will always be proportional to the number of threads used, and in the worst case, will be equal to $(\texttt{num\_threads} - 1)$, resulting in a minimal additional workload for the last thread, and in a less than significant impact on the overall performance of the parallel execution (assuming that the number of threads employed will be confined to the usual standards).

It was also necessary to ensure that each thread accessed the correct interval of indexes in the array of particles, with no overlaps with other threads executing at the same time. This was achieved by computing the initial offsets for each thread, as $(\texttt{thread\_id} \times \texttt{num\_particles\_per\_thread})$.

However, by performing the parallelization at particle level, and since the impact of each individual particle can have implications throughout the whole `layer`, several data races arose, often providing incorrect outputs.
Two solutions were considered to solve this problem:

- The first and naive solution was to place the write operation of the `update` method inside a critical region (with the `omp critical` directive), which is equivalent to putting it inside a lock. As expected, it produced the correct results, but at the cost of a significantly worse performance, due to the overhead added by the threads constantly trying and waiting to acquire sovereignty over that region of code;
- The second was slightly more complex, but was still logically sound. In summary, it consisted of each thread having its own copy of the `layer` array, to which they perform all the necessary intermediate operations, and only at the end of the execution of the parallel region (i.e., after all the threads had finished their work), the computation of the final result would be done, by merging the intermediate results. This way, any concurrent writes to the same locations were avoided, and we took optimal advantage of the usage of multiple threads. This approach yielded the correct results and a significantly better performance;

Obviously, the second option was preferable, and that is what was implemented. In theory, this one has a behaviour similar to a reduction pattern, as in, some intermediate results are computed in a first stage, and those results, in a final stage, are merged into the final result.

3 Tests Performed

In the following section, we will describe which hardware was used, what procedures were applied to perform the tests, and how they were then treated and studied.

3.1 Local Machines' Hardware

The local machines on which the tests were executed, had the following CPU models from Intel:

- Intel i7-8550U (1.8 GHz, 4 Cores and 8 threads);
- Intel i7-9750H (2.6 GHz, 6 Cores and 12 threads);

3.2 Department's Cluster's Hardware

The IT Department Cluster's machines used to perform the tests, had the following CPU models from AMD:

- 4 × AMD Opteron 6272 (2.1 GHz and 64 threads);
- 8 × AMD Opteron 8220 (2.8 GHz and 16 threads);

3.3 Automated Tests, Scripts and Macros

For testing purposes, two scripts were developed:

1) A Shell script, to automatically test our solution;
2) A Python script, whose goal was to generate storm files with a variable number of particles;

3.3.1 Shell Script

This script consisted of a battery of tests, grouped by the number of control points used, and each one was executed with a different number of threads, to fully understand the behaviour of our parallel version, separated in two groups:

- Storm files with a small number of particles (1 000, 10 000, 100 000, 1 000 000, 10 000 000 and 100 000 000 control points);
- Storm files with a bigger number of particles (1 000, 10 000, 100 000, 500 000 and 1 000 000 control points);

3.3.2 Python Script

This script generates random datasets, with the number of particles proportional to the number of threads, to analyze our parallel version, in terms of the Gustafson-Barsis Law.

It receives an initial number of particles and a list of threads, generating the respective number of particles (`num_threads` × `initial_num_particles`).

Afterwards, two random integer values are chosen to calculate the means for the positions and energies, in the ranges of [0, 30 000[, and [150, 250[, respectively, acting as the seeds of two Truncated Normal Distributions in `NumPy`.

3.4 Final Tests and Results

To obtain the final results to be presented and discussed in this report, three different test cases were selected:

1) A single storm file (`test_01_a35_p8_w1`), with a small workload (number of particles);
2) A single storm file (`test_07_a1M_p5k_w2`), with a sizeable workload (number of particles);
3) Two storm files, executed in sequence (`test_07_a1M_p5k_w3 test_07_a1M_p5k_w4`), both with a huge workload (number of particles);

4 Results

For this specific problem, we verified that most of the work to be processed was represented by the number of particles, and the layer's sizes, given as arguments. For instance, a higher number of particles would result in bigger workloads for the threads launched, and bigger layer sizes would translate to more cells needing processing, for the impact of each particle on the layer, as well.

Since most of the parallelism was applied at the particle's processing level, it should be expected that, when applying our parallel program to files of storms that contain a small number of particles, it would not offer better performances, when compared to the given sequential version.

This behaviour was confirmed by testing that hypothesis with the provided test files 1, 3, 4, 5, 6, 8 and 9 (max. of 8 particles), due to the small workloads for each thread.

When testing the reverse hypothesis, i.e, running our parallel version against storm files with a huge number of particles, such as the provided test files 2 and 7 (with 20 000 and 5 000 particles, respectively), the real advantages of our parallel version started to take the desired effect, providing better performances, with some considerable speedups.

4.1 Test Case #1 - Single Storm File - Small Workload

For the 1st test case, the speedup was barely noticeable, a behaviour which corresponded to our expectations:

- As stated earlier in the report, this case study was tested locally, on two different machines, with slightly different hardware (see sub-section 3.1);
- The machine with an 8th generation Intel processor, with fewer cores, showed the bigger speedup;
- However, in both of cases, there was a noticeable decline of the speedup, with more than two threads;
- The efficiency also followed the same trend, with a declining in executions with more than two threads;

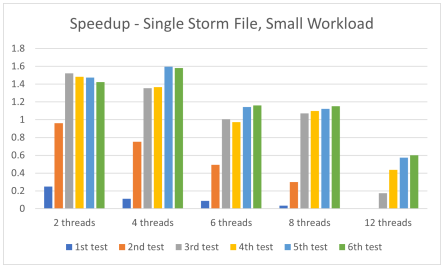

Fig. 1: Speedups for the 1st test case

4.2 Test Case #2 - Single Storm File - Huge Workload

For the 2nd test case, the results were still in the realm of our expectations, with the more powerful of the two machines showing a better performance, with very positive speedup:

- A speedup up to ≈ 4 was registered, maintaining a steady increase as the number of threads grew;
- Oppositely, the efficiency values showed a decrease;

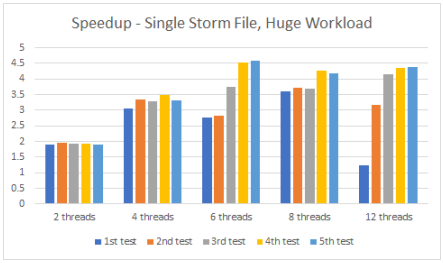

Fig. 2: Speedups for the 2nd test case

4.3 Test Case #3 - Two Storm Files - Huge Workload

The 3rd test case revealed similar results to the 2nd test case:

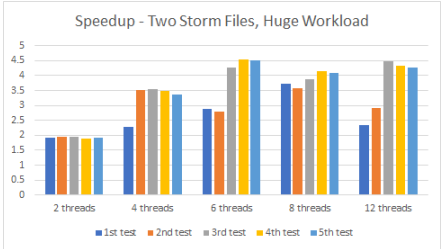

Fig. 3: Speedups for the 3rd test case

4.4 Profiling after Parallelizations

After the parallelizations made, we used the `Valgrind` [2] tool again, from which it were noticed some improvements, most specifically, verifying that the `read_storm_file` method started to dominate the most of the computation:

- Calls for `__vfscanf_` function - [80% of exec. time];
- Calls for `__strtol_` function - [20% of exec. time];

4.5 Other Observations

Some tests were also performed in the IT Department's Cluster, by running the script with the automated tests, verifying worse results than the ones obtained in the local machines, for the same number of threads.

In general, the local machines performed better than the Cluster, for 2, 4, 6, 8 and 12 Threads, however some additional tests were run in the Cluster for 16, 32 and 64 Threads, where, as expected, the Cluster achieved slightly better execution times than the ones achieved in the local machines, using a maximum of 12 Threads. This behavior could be explained due to the fact that, the AMD processors used in the Department's Cluster are models much older than the Intel processors used in the local machines.

This is a practical example to show that the hardware used for Parallel and High-Performance Computing can also have a profound impact on performance and should always be taken as a considerable aspect to be considered.

5 Performance Analysis

For a performance analysis, we analysed the results from the perspective of Amdahl's Law (fixed size speedups) and Gustafson-Barsis' Law (scaled size speedups).

5.1 Amdahl's Law (Fixed Size Speedups)

The Amdahl's Law (for fixed size speedups) argues that the execution time T_1 of a program falls into two categories:

- The time doing non-parallelizable serial work (W_{ser});
- The time doing parallelizable work (W_{par});

Thus, the total execution time for the serial version is:

- $T_1 = W_{ser} + W_{par}$

If the parallel work is divided by P processors, the total execution time for the parallel version is lower bounded by:

- $T_P \geq W_{ser} + \frac{W_{par}}{P}$

Additionally, the Amdahl's Law also defines that the Speedup is upper bounded by the following expression:

- $S_P \leq \frac{1}{1+\frac{(1-f)}{P}}$

So, first, we estimated the fraction of non-parallelizable work, f, and the fraction of parallelizable work, $(1-f)$, verifying that, as expected, the percentage of serial work tends to dominate the most part of the computation:

Test Case Num.	Num. Threads				
	2	4	6	8	12
1	0.238	0.384	0.446	0.591	1.000
2	0.041	0.072	0.141	0.153	0.308
3	0.038	0.089	0.131	0.152	0.227

For the test case #2 (see sub-section 4.2), the serial fraction, f, increases, as more threads are used in the parallel version:

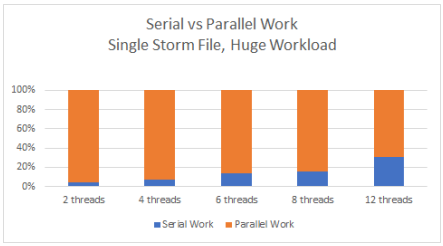

Fig. 4: Fraction of Serial vs Parallel Work (Test Case #2)

For the analysis of the Amdahl's Law, we computed the following maximum speedups, for the 3 test cases selected:

Maximum Speedups	Num. Threads				
	2	4	6	8	12
Test Case #1	5.287	2.945	4.842	1.912	N/A
Test Case #2	22.025	14.457	8.484	6.091	5.440
Test Case #3	25.040	20.398	14.424	6.745	5.335

After computing the maximum speedups for all the 3 test cases selected, we verified that all the speedups achieved did not violate the Amdahl's Law, for fixed size speedups.

5.2 Gustafson-Barsis' Law (Scaled Size Speedups)

The Gustafson-Barsis Law argues that the problem's sizes grow as the parallel computing capabilities also grow and states that the execution times for the serial and parallel versions are the following, where a and b are the fractions of the serial and parallel work, respectively:

- $T_1 = a + b$; • $T_P = a + (P \times b)$;

Furthermore, from the Gustafson-Barsis' Law, it is also possible to set an upper for the scaled speedups:

- $S_P \leq \frac{T_P}{T_1}$

In this way, for the performance analysis, in terms of the Gustafson-Barsis' Law, we also made some tests, using scaled datasets (see sub-section 3.3.2), on where was verified the expected trend for the scaled size speedups:

Num. Threads	2	4	6	8	12
a (Serial fraction)	0.893	0.225	0.082	0.000	0.000
b (Parallel fraction)	0.107	0.775	0.918	1.000	1.000
S_P	1.057	1.193	1.418	2.043	3.064
$\leq$	✓	✓	✓	✓	✓
Scaled Speedup	1.107	3.324	5.590	8.000	12.000

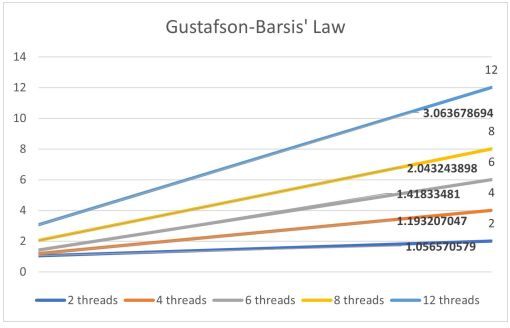

Fig. 5: Scaled Size Speedups (Gustafson-Barsis' Law)

6 Conclusions

In this section, are approached some final conclusions and insights about the project and the write of the report.

6.1 Final Insights

In the end, the results matched all our expectations, since:

- All the relevant theoretical concepts learnt during the lectures were utilized, and verified;
- The profiling analysis of the last parallel version demonstrated how successful the parallelization was, by verifying the reducing of the workload on the hotspots verified initially;
- The overall performance of the program improved substantially, reducing the execution times, with the expected behavior, after a deep analysis;

All the methodologies and overall workflow employed will be a valuable tool for some future projects and researches, involving tasks of parallelization and data analysis.

To that end, this project and the correspoding report exposed us to a new way of rationalizing data, and how to transform it into a more refined end-product, and will surely be an asset for future endeavours.

6.2 Difficulties Found

Unfortunately, it was not possible to successfully parallelize all the sections asked in the given source code.

6.2.1 Updates of `layer`, `maximum` and `position` arrays

In addition to the successful parallelization of the sections 4.1 and 4.2.1 of the code, other parallel approaches were also experimented with, such as the sections 4.2.2, and 4.3.

For the parallelization of the above mentioned sections, for the an approach similar to the parallel stencil pattern was used, using an individual copy for each neighbor of the current cell accessed, to avoid situations of data races, regarding the update of energy values of the cells on the `layer`, by considering the energies of its neighbors' copies, as also, the computation of the maximums of the storms.

In a similar strategy for the one applied to the section 4.1 and 4.2.1, where the `layer` array size was divided by the number of threads, and again, we specifically made the distinction between the points attributed to the last thread, and the points attributed to the remaining ones.

Furthermore, two auxiliary arrays were created to store the positions and energies of the local maximums found by each thread, regarding the `layer` array and its copies.

After the mentioned updates to the `layer` array and the calculation of each `local maximum`, the only remaining step is to compare all the local maximums and conclude which one is the best, for the global `maximum` for each storm.

However, after testing and analyzing performance, this approach was later abandoned, due to the fact that the obtained results differed from the ones of the benchmark from the sequential version, and that, from a performance perspective, the time gained (less 2 seconds of execution time, on average) did not justify its implementation.

The link for the branch on the GitHub, with the code of the parallelization of these two sections is the following:

- https://github.com/rubenandrebarreiro/cp2020-2021_g32_42648_52701_53042/blob/Ruben_Barreiro_and_Martim_Figueiredo_parallelization_update_with_neighbors_and_maximums/Src/energy_storms_omp.c

6.3 Division of Tasks amongst the Group's Elements

The tasks amongst the group's elements were divided as:

- João Soares [30.00% of the tasks]:
  - Execution of the script of the automated tests, in the Department's Cluster;
  - Macros in Visual Basic for the plots and charts of the final results (Speedups and Efficiency);
  - Co-wrote this report, and acted as reviser and linguistics editor;
- Martim Figueiredo [35.00% of the tasks]:
  - Parallelization of code's sections 4.2.2 and 4.3;
  - Macros in Visual Basic for pre-processing of the results extracted from the execution of the script for the automated tests (i.e., the standard deviation, outliers' removal and average);
  - Macros in Visual Basic for computation of the final results (i.e., the Speedups and Efficiency);
  - Provided a draft for the report, summarizing the information;
- Rúben Barreiro [35.00% of the tasks]:
  - Parallelization of code's sections 4.1 and 4.2.1;
  - Script in Shell, for the automated tests and its extraction of results to a `.csv` file;
  - Script in Python or the generation of datasets from a Normal Distribution for the analysis of the Gustafson-Barsis Law;
  - Co-wrote this report, and was responsible for its structural integrity;

References

[1] Arturo Gonzalez-Escribano, Storms of High-Energy Particles: An assignment for OpenMP, MPI, and CUDA/OpenCL, https://sc18.supercomputing.org/proceedings/workshops/workshop_pages/ws_eduhpca114.html.

[2] Nicholas Nethercote, Julian Seward, Valgrind: A Program Supervision Framework, Electronic Notes in Theoretical Computer Science, Volume 89, Issue 2, 2003, Pages 44-66, ISSN 1571-0661, https://doi.org/10.1016/S1571-0661(04)81042-9.

[3] Mahmoud Hesham El-Magdoub, Best Square Root Method - Algorithm - Function (Precision VS Speed), https://www.codeproject.com/Articles/69941/Best-Square-Root-Method-Algorithm-Function-Precisi

[4] SPEC Flag Description for the GNU C/C++/Fortran Compiler, http://www.spec.org/omp2012/flags/gcc-linux64.20190904.html

[5] Timothy Mattson, "An introduction to openMP", Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid, 2001, pp. 3-3, doi: 10.1109/CCGRID.2001.923161.

[6] Is it better to parallelize the outer loop? - OpenMP Forum, https://forum.openmp.org/viewtopic.php?t=1479