Project report

Francisco Mateus (53270), Dionisie Tarlev (52612) and Diogo Silva (52369)

1 INTRODUCTION

To date, computers have not only increased in raw processing power, but, more importantly, in the number of processing cores in their central processing units (CPU). This substantially increases the number of tasks that can be done at the same time, something a single core would have difficulty doing given the physical constraints that prevent further frequency scaling. This type of computing is known as parallel computing and it is important for the programmer to learn some of the related concepts to make better use of the hardware. This project intends to apply some concepts taught in Concurrency and Parallelism classes, such as parallel patterns and data dependencies, in a program that simulates the effects of high energy particle bombardment on an exposed surface, for example on the surface of spacecraft in outer space. The development of the paralleled version of this program was aided with the use of the OpenMP (Open Multi-Processing) application programming interface (API), a very handy tool that simplifies multi threading program development since the programmer can start developing from a sequential version of a program without usually much changes in the source code.

2 MAIN CONTENTS

2.1 Energy bombardment phase

2.1.1 Description

During the energy bombardment phase, the layer is being bombarded by multiple particles. When a cell is hit by a particle, the impact cell and the ones near it receive some of the particle's energy and the further a neighboring cell is to the impact cell, the less energy it receives.

When a cell is far enough from the point of impact of the particle, to the point where the energy it receives is below a certain threshold (which depends on the material), the value of the energy received is counted as negligible and it is not added to the cell.

2.1.2 Parallel version implementation

Depending on the particle, there could be time saved by only updating the values of certain cells. Since the bigger the distance from the cell of impact, the lower the energy on subsequent cells, after a certain point it becomes unnecessary to calculate the energy that the subsequent cells receive, because if the current cell is below the threshold, the next cell will also be, because the energy received can only go lower.

From these equations, we can calculate the positions of the limits of the layer, inside of which the energy received by each cell is above the threshold.

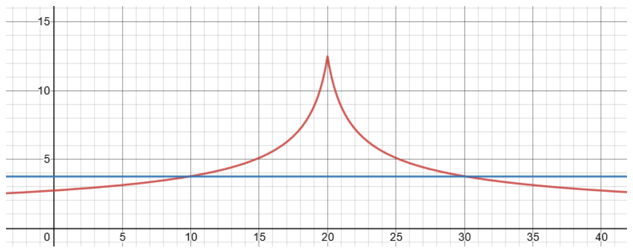

Fig. 1. With the layer being the x axis and energy being the y axis in this figure, only the x positions where the red function (energy received by the particle) is above the blue line (the threshold) are relevant for updating

distance = pos - k+1

attenuation = $\sqrt{distance}$

energy$_k = energy/layer\_size/attenuation$

energy$_k \geq threshold/layer\_size$

threshold/layer size $\leq energy/layer\_size/attenuation$

$\iff attenuation \leq energy/threshold$

$\iff distance \leq (energy/threshold)^2$

$\iff k = pos + (energy/threshold)^2 - 1 \vee k = pos - (energy/threshold)^2 - 1$

So instead of trying to update the whole layer, and only updating the relevant cells, it now only updates the relevant cells from k_{min} to k_{max} , which can save considerable time when the the amount of cells in the layer and/or the threshold are high.

The second step is to simply divide the resulting layer for each thread using OMP and update it. Since the update function has no dependencies among cells there will never be any concurrency issue among them. This parallel pattern is known as **Map**, where a simple operation is applied to all elements and it is used to solve embarrassingly parallel problems

Another approach could have been for instead of dividing the layer for each thread, dividing the storm particles for each thread. This would make it so the layer is shared among threads, and there will be an output dependency when the threads update the same cell. To bypass this issue, there's the possibility of updating the cells atomically,

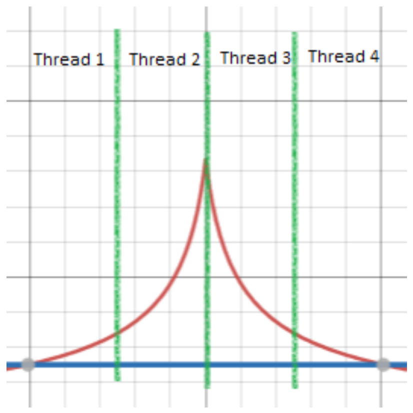

Fig. 2. The easiest way to quickly compute the energy on the relevant part of the layer is to divide the relevant part equally for each thread

wasting the efficiency of concurrency, or having each thread work on its own layer, and adding them all at the end, which would drastically increase space used and also increasing time by adding them all, compared to the chosen solution.

2.2 Energy relaxation phase

2.2.1 Description

In this phase the material reacts by slightly distributing its charge. Each cell is updated with the average value of three cells: the previous cell, the cell itself, and the next cell. The extremes of the layer are not updated during this phase.

2.2.2 Sequential version analysis

To avoid destroying the original values in the array while they are still needed to update other neighbors, the array values are first copied to a second ancillary array before the relaxation. The old values are read from the ancillary array, while the new values are written to the original array. There is both a true dependency and an anti-dependency in this process.

2.2.3 Parallel version implementation

It was evident that creating the ancillary layer to preserve the old values of the original one creates a lot of overhead each time a storm file is processed, so it was natural to think about removing this array, not only to improve memory consumption, but also speedup.

However, doing so implies resolving some complex loop-carried dependencies, besides preserving the old values in some other way, like using ancillary variables.

First, instead of a whole ancillary array, there would only be needed 2 values: the original value of the previous cell and the original value of the current cell. After updating the current cell, its previous cell's original value becomes the current cell's original value, and the cycle proceeds to updating the next cell.

Second, in order to make this process parallelizable, the layer would be split equally by the threads. However, since each cell needs its neighbors, the extremes of each sub-layer would need to be given ancillary cells with the original values of their missing neighbors. This process is done before the updating process. This is a implementation of a Map related pattern known as **stencil**, where each instance of the map function accesses neighbors of its input, offset from its usual input.

2.3 Maximum location phase

In this last stage that process each wave, the point with the maximum energy is located and its value and position are stored. After all the waves have been processed, the maximums and positions for all waves are printed.

2.3.1 Sequential version analysis

Upon analysing the sequential code, we can easily identify two output dependencies, as both the array maximum and the array positions are written to the same indexes inside a for loop.

We also encountered an error in the source code, as the original loop works by checking for local maximums, comparing the current layer position value, with the previous and next one; by this logic, the first and last positions of the array are never evaluated, so in a scenario where energy values throughout the layer positions would continuously increase, a maximum would never be correctly identified.

2.3.2 Parallel version implementation

In order to solve the output dependencies, we begin by declaring a variable which will hold the maximum value found in the loop. As it is found within a parallel code block, each thread will have its own private "maxk" variable, which will hold the maximum value for the corresponding thread work.

As per the original code, we utilize the "local maximum" condition to save potential maximums. As the threads finish their work, each will have a different value in their "maxk" variable, which will then be processed by the conditions written within the critical block of code; for each storm, a single maximum and position index will be occupied by the biggest value between all threads. This is and implementation of the **divide and conquer** parallel pattern, which breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

As an attempt to correct the original code, we include a condition to ensure that, if no change was made to the maximum value during the loop iteration, then that can only mean that this layers maximum lies within the first or last position.

3 Results and evaluation

With our implementation done, it was time to benchmark our implementation on the cluster provided by the Department of Informatics (DI) of the NOVA University of Lisbon and evaluate the parallel performance by studying some performance metrics, such as speedup, efficiency and cost.

In order to test and benchmark our solution, we implemented and used some python scripts for this effect:

- `TestsScriptBase.py` - is considered the base of the other scripts and contains all functions and variables necessary for the other scripts to run. The script has no effect by itself, and needs to be imported by the other scripts.

- `Benchmark.py` - like the file name implies, is used to benchmark our solution. After the benchmark, the script will export the results metrics (mean time, speedup, efficiency, cost, etc...) to the `seq.csv` and `omp.csv` files, which contains the metrics about the sequential (`energy_storms_seq`) and paralleled (`energy_storms_omp`) programs samples metrics, respectively.
- `RunCompare.py` - Used to test the paralleled version of the program with a specific number of threads by comparing the output with the original, sequential program. The script will run both programs once.
- `TestFilesScript.py` - Tests all test files individually and combined (example: test all test_02 files) in order to check the correctness of the paralleled program.
- `BuildPlot.py` - Produces a plot with the metrics about the `Benchmark.py` results. The script imports the `seq.csv` and `omp.csv` files produced by the `Benchmark.py` script and uses the Matplotlib python module to produce a plot.

By default, the `Benchmark.py` and `TestFilesScript.py` scripts will execute the original program and the paralleled program for each number of threads 5 times (for example: the script will execute 5 times the program with 1 thread, 5 times with 2 threads, etc...).

We have to remark that the benchmarks on the cluster were done with the energy storms programs compiled with assertions disabled. Although assertions are a very handy tool to debug our program by testing our assumptions, they are not intended to be included on the production code, since it may cause side effects on the performance, depending on how they are implemented.

Also, before doing the benchmark, we choose the adequate number of threads for the target node in the cluster given the number of cores, since executing the program with a number of threads above the available number of processing cores will not only not improve performance, but degrade it, as taught and shown in the practical classes of Concurrency and Parallelism.

Like mentioned before, in order to evaluate our solution after the benchmarks, it was used the following performance metrics, taught in the Concurrency and Parallelism classes:

- **SpeedUp** $S(p) = \frac{T_1}{T_\text{p}}$
- **Efficiency** $E(p) = \frac{S_\text{p}}{p}$
- **Cost** $C(p) = p \times T_\text{p}$

Where T_p is the mean time of all program samples executed with p threads. Unfortunately, we detected near the deadline that the the efficiency plot on both **Figures 3 and 4** were generated from a previous version of the `Benchmark.py` script that had a bug on the efficiency metric. This bug was fortunately corrected.

Originally, we expected that our energy bombardment optimization would improve considerably our program performance in comparison with the original, since we made the program ”smart” by only updating the layer range above the threshold. However, as you can see in **Figure 3**, this expectation wasn't met with the default threshold value, given that it was so low that generally all energy increments caused by the particle collision on all layer positions were above the threshold. The small mean time gap between the original (`SEQ` in the mean time plot) and the paralleled program with only one thread (`OMP-1` in the mean time plot) is explained by the removal of the `layer_copy` array, described in the energy relaxation phase section, which not only improved execution time by not initializing another array, but more obviously memory consumption, as we expected.

In order to fully test our energy bombardment optimization, it was necessary to increase the program threshold value to a value that was enough to show notorious results. Looking at **Figure 3**, we see that the parallel program mean time was massively improved with a threshold value of 50. However, it is assumed that it was not the goal of the project to tune the threshold value, given that it was previously defined as a constant in the original program. But given that the threshold usually depends on the material of the layer, as mentioned in the project assignment, we though it was interesting to tune the threshold value with a program argument (-h).

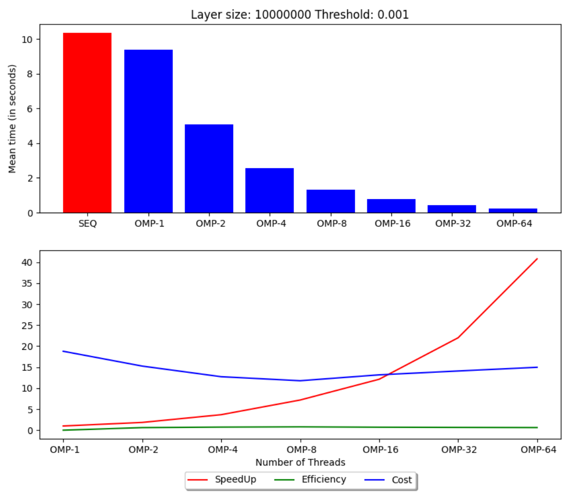

Fig. 3. Using all test_01 test files on node 12 with default threshold value

When it comes to the parallelism metrics shown in the second plot of both figures, it is noted that the speedup on the paralleled program is exponentially increased as the number of threads increases, which implies that the mean-time is decreasing, as shown in the first plot of both figures. This means that our program escalates well as the number of threads increases, which is a good thing and what we were expecting. However, evaluating program performance taking only into consideration speedup is not enough. Analyzing the cost metric, we can see that it is decreasing until 8 threads and increasing progressively after that. Besides that, although the efficiency plot values are too low and wrong, as stated before, we can compute the efficiency manually given

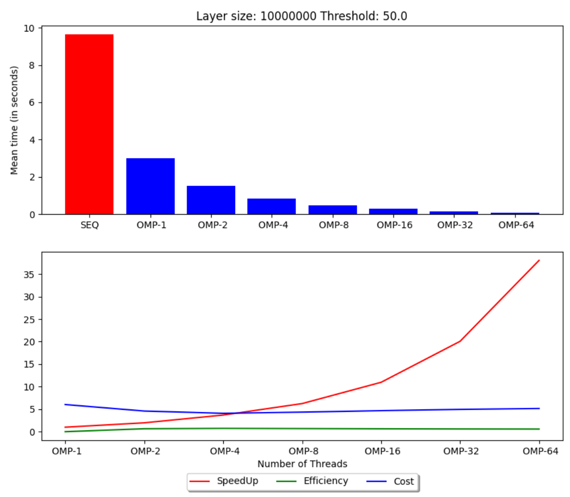

Fig. 4. Using all test_01 test files on node 12 with a threshold value of 50

the speedup, and we notice that it is progressively declining as the number of threads increases, which was expected.

4 Work division

At the end of project, we analysed and evaluated our work division and performance and we honestly conclude that there was some imbalance of our work division, as stated below:

- **Francisco Mateus** 50% - Was responsible for implementing the energy relaxation and maximum location sections of the program, implementing the python and bash scripts, doing the benchmarks on the cluster, fixing some bugs and writing the report.
- **Dionisie Tarlev** 35% - Was responsible for implementing the energy bombardment section, giving useful insights about the energy bombardment and relaxation sections of the program, fixing some bugs and writing the report.
- **Diogo Silva** 15% - Was responsible for fixing some bugs on the maximum location section on both the original code and the paralleled one, and writing the report.

Acknowledgments

As for acknowledgements we want to thank professor João Lourenço for lecturing CP class with great materials (recommended book and slides) and having a high availability to clarify questions online. Our colleagues from the piazza platform that made questions about similar dilemmas that we came across throughout the project and finally to the great descriptive documentation available online for OpenMP that helped us brainstorm on how to build our parallel program.

We specially want to thank Dionisie Tarlev 52612, who gave wonderful ideas about the optimization and parallelization of the Energy Bombardment and Relaxation phases. We also thank Group 02, more specifically Alexandre Correia 53298, for brainstorming ideas on how to parallelize some parts of the program.

References

[1] M. McCool, M. Arch and J. Reinders, *Structured Parallel Programming: Patterns for Efficient Computation*, Morgan Kaufmann (2012). ISBN: 978-0-12-415993-8
[2] Tim Mattson, Larry Meadows, A ”Hands-on” Introduction to OpenMP https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf
[3] *OpenMP Application Program Interface* https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf
[4] *Guide into OpenMP: Easy multithreading programming for C++* https://bisqwit.iki.fi/story/howto/openmp/