A Parallelization Study of Storms of High-energy Particles

André Monteiro - 52337 Bruno Brito - 52361 João Martins - 52422

Abstract—In this report we describe our approach at parallelizing the code that simulates the effect of high-energy particles bombardment on an exposed surface, by explaining what we changed in the original sequential code and problems we faced while doing so. Next we show results, by comparing the sequential to our parallel version and the results we obtained whit different amount of threads. Finally we conclude by explaining the results, what we expected and possible improvements.

Index Terms—Concurrency and Parallelism, OpenMP, Gprof profiler, Time, Speedup, Efficiency , Cost, Amdahl's Law .

1 Introduction

This project we were provided with code that simulates the effects of high-energy particles bombardment on an exposed surface, based in the EduHPC'18 Peachy Assignment from the Trasgo Research Group of the Universidad de Valladolid.

For this implementation only the surface cross section was considered. For this the surface is represented by an array that stores the amount of energy in each point. The program will compute the accumulated energy of each point after the impact of the waves which are passed as an argument. In the end the program outputs the point with highest energy accumulated and respective energy.

Our objective for this project was to parallelize the code that was provided in order to obtain better performance as well as evaluate the impact of the parallelization. In order to achieve this we used the OpenMP [1] programming model, in which we were required to identify and remove code dependences before using OpenMP. To evaluate potential performance bottlenecks we used the Gprof profiler [2] to find and identify where the program was spending more time.

Finally we ran both the sequential code provided and our parallel code for different tests, the ones provided at the start and some we generated, in order to evaluate whether our code has the correct results and the impacts of our parallelization by testing in multiple machines with different amount of cores and threads, and by changing the amount of threads used in our program execution.

CP
June, 2021

2 Changes Made

2.1 Memory allocation and initialization (3 in base code)

In this section we opted by merging this loop into the second loop of Impacts energies to layer cells section. No dependencies were found while doing this whatsoever.

We decided to do it that way because layer and layer_copy are only first needed there and could save time by going through all the layer elements only once. Another reason that led us to merge the loop was the fact that by merging we could decrease by one the number of times that we would launch the threads, since it is faster for a thread to execute two more operations (layer[pos] = 0 and layer_copy[pos] = 0) than launching the threads again and go through all elements multiple times.

See loop parallelization details in section 2.3.

2.2 Storms simulation (4 in base code)

While studying the code provided we realized that the impact on the layer cells of each storm wave was dependent of previous storm waves.

Due to the nature of the problem, usually there aren't many storm waves unlike the number cells in the layer. We decided that it was worth more to parallelize the loop where the program goes through the cells of the layer. See more in section 2.3.

2.3 Impacts energies to layer cells (4.1 in base code)

After a group discussing we concluded that this was one of the best parts of the program to parallelize.

There are two loops in this section. The outer loop when parallelized gave us wrong results due to lack of precision. This happened because the associative property between floats is not granted and, later in the loop, the energy is summed into the respective layer cell,with the energy being a float.

This way we simply used "#pragma for omp parallel" to parallelize the inner loop, since we couldn't find loop dependencies, which gave us the same results as the sequential version. You can see more about how we compared parallel results with sequential results in section 3.1.

2.4 Copy values to the ancillary array (4.2.1 in base code)

This part was merged in the previous loop (Impacts energies to layer cells) just like we did in section 2.3.

We also did not find any dependencies here, although we added a condition check to execute the instruction only in the last run. We did it because it would be overriding it over the multiple runs of the Impacts energies to layer cells loop and a verification is faster than writing in memory.

2.5 Update layer using the ancillary values (4.2.2 in base code)

Initially we thought that not much could be done here aside from parallelizing it since if we merged this loop in 2.6 one, it would generate an anti-dependency. Later on, after debating ideas, we came up with a new solution.

The loop in the section 2.6 uses the previous, current and next value of layer_copy[pos]. Consequently, we could remove the anti-dependency by calculating those values before they are read.

This approach does not generate an output dependency because the written value of each position in layer is the result of a calculation which always returns the same for each position. Although there are no dependencies, those values are going to be written in memory multiple times for no reason, but the fact of having one less iteration over those values and having to launch threads one less time, made us believe that this solution was preferable and it would run faster compared to the previous one.

Along this line, this part of the code was merged and parallelized in section 2.6.

2.6 Locate the maximum value in the layer, and its position (4.3 in base code)

After taking a look into this section, we noticed there was a flow-dependency. The program checks if the iterated cell energy is bigger than the maximum energy known so far and if so it replaces that same maximum energy by the iterated one.

The best way we found to remove this dependency was to create an auxiliary array where each thread would store its maximum found and later on, after all threads finish their jobs, sequentially going through that array and retrieving the maximum energy value among all maximums that the threads found.

After removing all dependencies we just used "#pragma omp parallel" to instantiate the threads, "omp_get_thread_num()" to get their respective thread ID and use it as an index of the auxiliary array to store its maximum energy value found and "#pragma omp for" to parallelize the loop.

We did not feel the need of parallelizing the later loop we introduced, which goes through all maximum energy values that the threads found because, since currently there are not machines with thousand or millions of threads, the auxiliary array will be small, and launching the threads would create an overhead which might not be worth it.

3 Results

3.1 Testing results

In order to test our results, we ran our parallelized version of the code against the sequential version to compare the results. This verification was done by running our version five times for each amount of threads and comparing the result that the sequential version gave, this way we could see if the program gives the right results and if there are inconsistencies due to some concurrency error. Regarding the correct results as previously discussed 2.3, the initial results were incorrect due to the lack of precision of our results, which happened because floats associative property is not granted. To avoid the concurrency errors we were required to evaluate and identify possible dependencies in the provided code before attempting to parallelize any part of the program.

3.2 Testing performance

To test performance, as previously mentioned 3.1, after confirming that we were obtaining correct results by testing against the sequential version, we used a script which ran each of the test we choose five times for each amount of threads. From there we removed the possible outliers, these are the biggest and the smallest time values, and we did the mean with the three remaining.

For our tests we used node 13 in the cluster which has 8 x AMD Opteron 8220 processors, 16 cores and 16 threads. As such we tested our code for 1, 2, 4, 8, 16 and 32 threads. This way we could see the effects of increasing the amount of threads used by the program until its maximum amount, and what would happen if we used more than the available by the machine.

The tests chosen for this purpose, given our time limitations, were the ones that covered different situations, the combinations of the size of the layer and the amount of particles in the waves 3.3. With this we hoped to cover the different possible results/outcomes our program would face. For the test with a small layer and a large amount of particle we were required to create a test file, as non of the provided ones covered this case. For this we used a script that generated test files, "wave-gen.py" which could generate files with random particle locations or with tendencies.

The results we obtained can be seen below.

3.3 Time

3.3.1 Test 8 - Big layer and few particles

This first figure we present is the result of one of the provided tests cases, test 8. In this test there's a big layer that is bombarded with one particle per wave for a total of three waves.

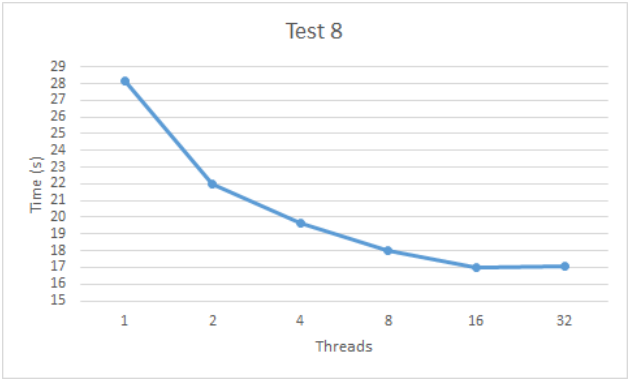

Fig. 1. Test 8 execution time.

From the figure we can observe that the time the test took to run decreased with the increased number of threads. When the test reaches the number of threads of the machine it gets the best execution time. As expected when the number of threads exceeds the amount of threads of the machine

in the cluster, the performance decays due to the overhead of having more threads than processors.

3.3.2 Test 2 - Big layer and many particles

Test 2 is a workload test. It contains a decently sized layer and many particles. We chose to run this test because it took the optimal amount of time to run in the two hours we had reserved in the cluster.

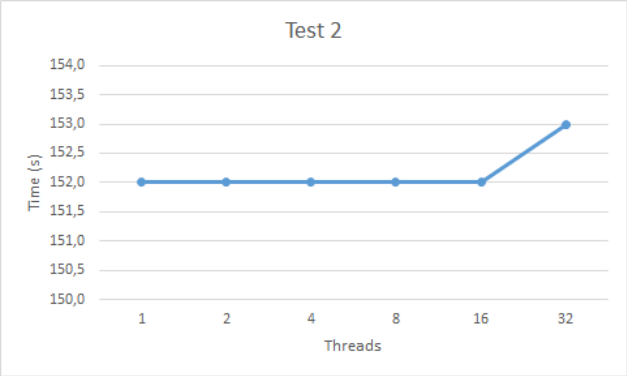

Fig. 2. Test 2 execution time.

In this test we observe some unexpected values. We observe that the execution time remained constant up to 16 threads. When the number of threads goes above 16, the number of threads of the machine, the performance decays and we believe this is given to the overhead of having more threads than cores.

3.3.3 Test 10 - Small layer and many particles

Test 10 is a test created by us which has different characteristics from the provided ones. This test has a small layer which is bombarded by many particles.

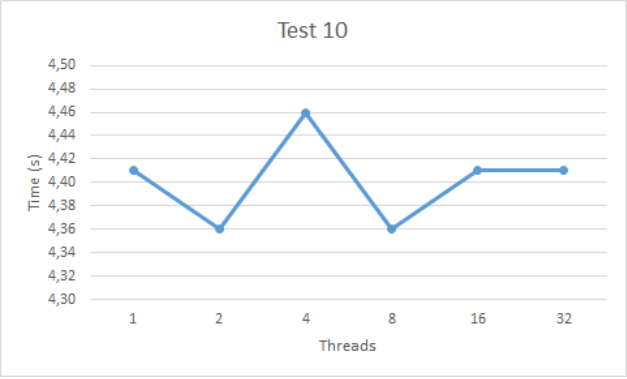

Fig. 3. Test 10 execution time.

In this test we can infer that our solution doesn't take much advantage from parallelization given the average execution times for different thread numbers is almost constant.

3.3.4 Test 9 - Small layer and few particles

This test is test 9 from the provided test files. We used this test just to observe how the performance would evolve when we have a small layer and a small number of particles.

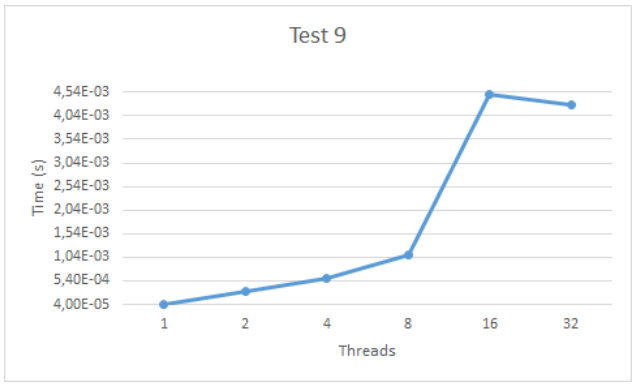

Fig. 4. Test 9 execution time.

As we can observe from the figure, the performance actually decayed. This was expected since the computation is really small. The performance loss with the increasing number of threads must be related with the cost of launching and managing threads.

3.4 Speedup

To calculate our speedup we used the time of using just one processor and divided it by the time of n processors, being n the number of threads used in that test. As we can see in the figure 5 and by the times obtained previously 3.3, test 8 obtained a speedup to up to 1.6 for 16 processors, which is the amount the machine we tested had. Both test 2 and 10 did not benefit from the parallelization with more threads as the results are around 1, meaning time changed very little. Test 9 however as a speedup below 1 meaning the times actually got worse with more threads, this can be due the overhead of launching the threads is higher than if the program ran sequentially.

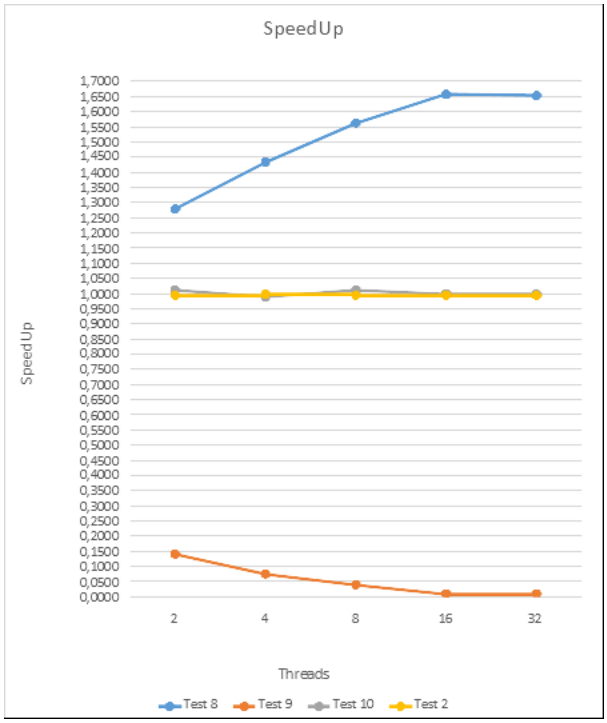

Fig. 5. Speedup Comparison.

3.5 Efficiency

Efficiency is defined by ratio between speedup and the number of processors. As such, the closer to 1 the more efficient the code is. However, as efficiency is directly dependent on speedup, means that the Amdahl's law still applies as such we cannot expect an efficiency of 1 unless the code is embarrassingly parallel, which is not the case. So as we increase the number of processors, since the speedup isn't increasing as much our efficiency will go down.

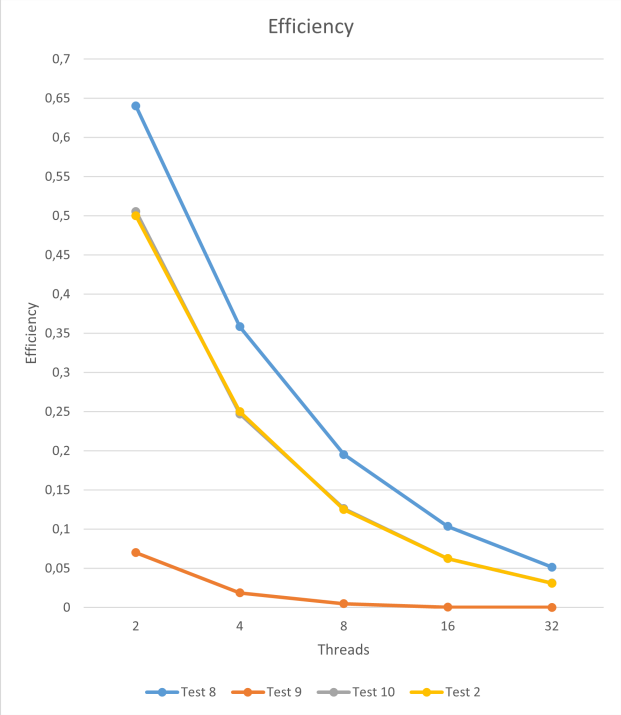

Fig. 6. Efficiency Comparison.

3.6 Cost

The cost of solving a problem is defined by the runtime and the number of processors.In this we see that for test 8,9,10 the cost until the number of threads in the machine is linear, with more it increases a lot more. For test 2 it appears to have an exponential growth with the increase of threads.

This happens because the test 2 execution times remain constant with the increased number of processors. Therefore, it is expected that the cost increases exponentially because the number of processors grown exponentially in the tests.

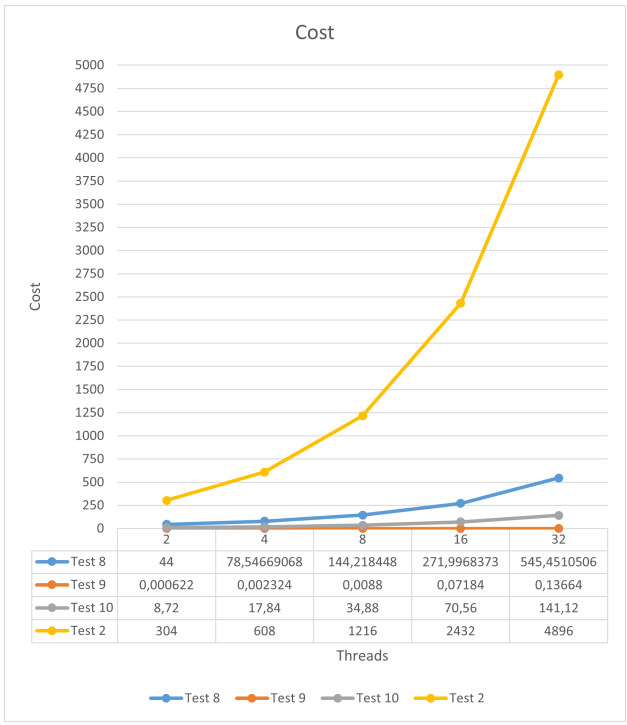

Fig. 7. Cost Comparison.

4 Conclusion

To conclude, from what we observed in the results we obtained 3.1, the profiler when ran spent 60 to 70 % of the time on the update function. Our solution did not parallelize this part. This means we are working with the remaining 30 to 40 %. According to the Amdahl's Law with 50 % parallelization we can only expect a speedup of 2 or lower. Since we have less than this amount parallelized we can only expect a speedup lower than 2 which we observed in all the tests 3.4

ACKNOWLEDGMENTS

The authors would like to thank to:

Post #73 on piazza by Hugo Lopes https://piazza.com/class/km3ev13xcu1fd?cid=73 [3], which provided a script which we based ours in order to run and test our program.

Post #72 on piazza by David Pereira https://piazza.com/class/km3ev13xcu1fd?cid=72 [4], whose script lead to a discussion in the comments and by having the teacher's suggestions in to account lead to our implementation of a wave generation script.

INDIVIDUAL CONTRIBUTION(S)

André Monteiro - 33%

This group member Made a script to generate new Wave tests. Calculated the speedup, efficiency and cost. Wrote section 1, 3.1, 3.2, 3.4, 3.5, 3.6 and 4.

Bruno Brito - 33%

This group member parallelized the code provided. Made the graphs presented on this report. Wrote the section 2.

João Martins - 33%

This group member made a script to test if the program was correct. Tested the parallelized code in the DI clusters. Made a script to process the output data from the cluster tests. Wrote section 3.3.

All

Analyzed the code and discuss how to better parallelize it. Discussed the results.

REFERENCES

[1] T. Mattson and L. Meadows, *A "Hands-on" Introduction to OpenMP**, https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf

[2] *GNU gprof* , https://sourceware.org/binutils/docs/gprof/

[3] H. Lopes , *test script*, https://piazza.com/class/km3ev13xcu1fd?cid=73

[4] D. Pereira , *wave gen script*, https://piazza.com/class/km3ev13xcu1fd?cid=72