Project Assignment Report: Storms of High-Energy Particles

Bernardo Baldaia nº49901, João Frazão nº51920 and Miguel Candeias nº50647

Abstract—This report has the purpose of explain our implementation of the parallel version of the storm of High-energy particles simulator, using OMP directives. The focus of this article is to explain our implementation, present and discuss the obtained results from our experiments. The work for this report started by studying the OMP API, and then study and analyze the provided sequential code. After the last step, we then proceeded to start our parallel implementation of the simulator. Finally making use of a shell script that automatically runs all the tests, we proceeded to the testing phase in the DI Cluster. Finally we will present plot charts and discussed to obtained values.

Index Terms—Parallelism, C programming language, Threads, OMP, Storms of high-energy particles

1 Introduction

The purpose of this paper is to analyze and provide insights on our implementation of the parallelization of the algorithm responsible for for simulating the effects of the bombardment of several high-energy particles on exposed surfaces, in this specific case, an array data structure. To solve this problem, we will employ the OMP API, which consists of a series of compiler directives, routines and environment variables that influence runtime behaviour, allowing, and facilitating multi-processed parallel programming. In this paper, we will divide the analysis in several sections:

- Section 2 will provide a brief overview on the algorithm, and the OMP API, without going into too much depth, but providing a clear explanation of the essential parts of the issued problem.
- Section 3 will give insights into the authors' implementation of parallelization techniques using OMP in order to improve the sequential algorithm's efficiency.
- Section 4 will discuss the evaluation of the implementation's results when running the project on the DI cluster. We will explain the methods used for evaluation, present the results that we obtained and then discuss them, providing an analysis and explanation for these results.
- Section 5 will provide closing thoughts and lasting observations that the authors find important to mention, as well as a personal opinion on the implementation's performance.

2 Related Work

In this section we will start by explaining the issued problem, which is a simulation of the effects of high-energy particles when bombarded to a given surface. We will start by explaining the sequential algorithm, and then we will briefly talk about the library that allowed to parallelize the algorithm, which is OMP.

2.1 Sequential Algorithm

For this project we were provided with a sequential algorithm which its purpose its to simulate the effects of high-energy particles when these collide in a given surface layer. In the algorithm this layer can be represented by an array of size n, where the amount of energy is accumulated at each index for each wave of particles. Each wave file is represented by a .txt file where particles and its respective energy values, can be found. Furthermore, the program calculates and reports for each wave the point with the highest accumulated energy values, which represents higher risk of being damaged. In the next section, a brief explanation of every single phase of the sequential algorithm.

2.2 OMP

Since the sequential algorithm takes some time to produce results, by using OMP, which was explained in the practical classes, we can take the advantage of parallel programming techniques to make the algorithm computation faster. OMP is an API for writing multi-threaded applications, by using a set of compiler directives and library routines. By including the OMP header file in our C code, we are able to use its directives. In OMP threads communicate by sharing variables, and in order to control race conditions, it makes use of synchronization, which is computationally expensive. We will proceed with a brief explanation of some of this directives, which will be further later used in our implementation. For example, by using the pragma omp parallel, we are defining parallel regions, in which this represent a piece of code that will be executed by an arbitrary thread. For loops can also be parallelized, when matching all the necessary requisites, such has for example not having loop carried dependencies. Parallelization can be done recurring to pragma omp for directive. Other directives that we used in our implementation, in order to write results in the critical region, were pragma omp critical, and pragma omp atomic update. The first clause, due to its strict restrictions when threads are accessing the critical region, imposes an execution time increase, while comparing with the second

directive. In order to impose synchronization, a pragma omp barrier directive was applied. [1]

3 IMPLEMENTATION

The algorithm is divided into 3 different stages, and we will address each of them separately, since our implementation for each stage required an understanding of the specific mechanics that each stage contains.

3.1 Bombardment phase

The bombardment phase is where the program, for each particle in the wave, transforms their energies into values in Joules, and traverses the array positions whilst computing the accumulated energy values at each point, and in their neighborhood, since the impact not only affects the main position of impact in the array, but also the adjacent ones. This section starts off after the program reads the storm file(s) information, and prepares the array where the energy particles will impact. For an efficient job distribution, we started off by dynamically calculating the value of positions that each thread will process, henceforth named `split`. This will ensure that the threads of the program that run on the same data structure will share work efficiently and try to avoid race conditions where threads attempt to alter the same array position at the same time, leading to a deadlock situation. Since it consists on an imperfectly nested for loop, we can parallelize this with a simple `pragma omp parallel for`. However, we cannot use the `collapse` clause, since it's imperfectly nested due to the intermediate code between each for loop computing the energy value in Joules from the wave file's particle. Afterwards, we have another cycle, in which an `update` function is called. This function will update the layer with the previously computed energy values in each position. Due to the shared nature of the layer data structure, it is considered the critical region, for all purposes. As such, in order to avoid race conditions, we used `pragma omp parallel for` to divide the work between threads, and `pragma omp atomic update` to ensure that the layer updates are only accessed by 1 thread at a time. The rationale for not using `pragma omp critical` in this section is due to the fact that the critical region access is limited to a binomial expression, wherein a layer position is simply updated according to the previously calculated energy value. Using `atomic update` induces much less overhead than `critical`, and as such, it would not hinder the program performance as much. Knowing this, we now know that we have 2 for loops. The first goes through each particle in the wave, and the second goes through each position in the layer. Therefore, we can estimate the time complexity of this section, after parallelization, to be $O(\frac{stormSize*layerSize}{nThreads})$.

3.2 Relaxation phase

In this phase of the algorithm, the array distributes the initial impact from the particle to its adjacent cells. This is done by averaging the energy value of the initial impact, and its left and right neighbours. To do this, we employ an ancillary array, to which we copy the old values in order to read them, and compute the averages for the original layer. This section is comprised by 2 for loops, one with a nested for loop, and another, not nested. We define the parallel region once again using `pragma omp parallel` and we divide the work by the number of threads we have to work with. Using the `split` value that was mentioned in the previous bombardment section, we iterate, for each thread, the number of positions it will have to copy the values from the original layer, to the copy layer. We do this using a `pragma omp for`, which becomes parallel due to the previous directive. We make sure to add the `private` clause to the variables in this cycle so each thread is working within its own region in the layers. We employ the `pragma omp barrier` directive, in order to wait for all threads to have finished their jobs before proceeding to the next cycle. Here, we have another for loop in which we update the layer using the old values that are present in the ancillary array. We parallelize this using a `pragma omp for`. Finally, we can calculate the temporal complexity of this section. Since we have 2 for loops, running through the layer size (except first and last position), and one dividing the layer between threads, we have a final temporal complexity of $O(\frac{layerSize}{nThreads})$.

3.3 Maximum point location

This final stage is meant to pinpoint the position with the maximum value of energy stored after the wave has finished being processed. It contains a for loop, and a read access on the critical region, so there needs to be some special care when it comes to parallelizing this section. We define the parallel region using `pragma omp parallel` and we use `pragma omp for` to parallelize the for loop, running through the layer size. Then, since we must read the value of the layer positions in order to check for current maximums, we must surround this block of code with a `pragma omp critical` directive. We cannot use the `atomic` directive as mentioned above, since this is a multi line block of code and not a simple binomial expression or instruction. The time complexity for this section is $O(\frac{layerSize}{nThreads})$.

4 EXPERIMENTAL EVALUATION

4.1 Methodologies

In order to proceed with the evaluation of our results, we used the DI cluster to run our tests. We sought a node with sufficient processing capacity to withstand running 32 threads, which was the maximum number of threads we defined for our tests. We created a bash script that would run each test with different numbers of threads, namely, 4, 8, 16 and 32 and the results from these executions would be saved into separate files. For these executions, we decided to use the previously given test files that came with the initial project handout, as they went through both race condition testing and workload strain, and we found that these tests would suffice to gather the results that we wanted. Afterwards, these files would be read and organized into new files, using a python script implemented by the authors, in order to organize results for the final stage of data treatment. After the results are cleanly organized in a file, we plot these results using the python script. We found it important to evaluate the speedup and efficiency of

using different numbers of threads during parallelization, so for each executions, we calculated and plotted the speedup following Amdahl's Law, as well as the efficiency of using that specific number of thread for that array size, in that test. This analysis would give us a clearer insight on the true value of parallelizing a program, and would let us learn both proper and improper times to employ parallelization in our future programs. Another point we should mention is that during the development of the parallelization we used a profiler (we choose valgrind and used the tool callgrind). We used the profiler to check if there were any bottlenecks in our implementation.

4.2 Results

After analyzing the profiler results we noticed that a lot of the time in the sequential version was spent in the bombardment phase, which was expected (Appendix Figure 17). A good chunk of our time was spent parallelizing the impact phase to get the best results. The results we got from the profiler show that in tests that have a large number of particles and array size, such as test2, have shown a great improvement. On the sequential version around 99 percent of the execution time was spent on the impact phase, and after parallelization that number was 25 percent (Appendix Figure 18). The difference in the relaxation phase and finding the max value phase is neglectable since it was used a very low percentage of our time to start with. When we analyzed the results and the graphs generated by the python script we noticed a few patterns that we will now discuss. We can divide the tests provided in three groups: small storm small array, large storm large array and small storm large array. We also run the tests locally, in a machine with 4 cores, and in the cluster, using 32/64 cores, to see if there were any differences. On the first group (small waves small array), the results of parallelization were not great, both locally and in the cluster. Out of the tests in this group only one used more than 4 threads (test number 1 - Appendix Figures 1 and 2), and the rest (tests 3, 4, 5 and 6 - Appendix Figures 5,6,7,8,9,10,11 and 12) we only tested for 2 and 4 threads. Independently of the number of threads used, none of the tests benefited from parallelization, which is to be expected. The second group, which includes tests number 2 and 7 (Appendix Figures 3,4 and 13,14), had by far the best results. The results when ran locally benefited a speedup of up to 3.79 with 4 threads,with lower efficiency with more than 4 threads, and when ran in the cluster the speedup was as high as 10.14, with the efficiency falling with more than 16 threads. For the last group (test 8 - Appendix Figures 15 and 16), the results were not as bad as the first group, but they also did not benefit from parallelization, with the maximum speedup locally being 0.46 with 32 threads (on a machine that only has 4 cores) and 0.136 in a machine that has 32/64 cores.

4.3 Discussion

To explain the results we will also divide by storm size and array size. For the tests that have a small storm and small array size, we found that there was no positive impact on the performance and speedup of the program by adding more threads. The reason for this is that the time spent creating new threads (overhead) has a greater impact on the overall execution time of the program, since the array size is small, and there are fewer energy particles to be processed. Because of this, we theorize that the thread creation overhead takes up a larger portion of the runtime, which harms efficiency and speedup. On top of this, the speedup value registered locally was (mostly) lower than the one registered by executing our implementation in the cluster for the first test, since the machine that we used to run the project locally had only 4 cores, as mentioned in the previous section, versus the 32/64 cores that were available in the cluster node. It is important to note, however, that for other similar tests, namely the third, fifth and sixth test, the local speedup registered was higher than the cluster value. We suspect that it is due to the number of threads being specifically limited to 4 during these tests, and the creation overhead being slightly higher in the cluster [2]. In the case of tests with a large array and a large storm size, we verified that the cluster speedup was significantly higher than local speedup, as expected due to the higher number of cores, the thread scheduling was not swamped when testing in the cluster. Referring again to thread scheduling, the distribution of work among each thread contributes positively to reducing and optimizing execution time, since the amount of work each thread has to do is significant due to the high array size and high number of particles. The overload of the thread scheduler on the local machine is evident on the graph of test 7, representing that the machine does not have the capacity to handle the parallel computations any better, which means the thread scheduler is completely swamped. There was a specific test that deviated from these aforementioned norms. Test 8 has an extremely large array, but with only one particle per wave. We noticed that the results were actually negative when employing parallelization. This might be due to the time wasted by each thread on processing the entire layer. Although both local and cluster results were negative, we found that when using 32 threads, the performance actually increased when compared to the speedup registered when using only 16 threads. This behaviour can also be explained through the thread scheduling, since the array is easily divided between threads due to its large size, and we can therefore theorize that the better speedup value registered when using 32 threads is due to a better work distribution. Test 9 was a test specifically tailored to test race conditions in the program during a parallel execution. Although our outputs were correct when compared to the sequential version, the speedup results were quite negative. Once again, this is due to the fact that the array has a very small size, so the thread creation overhead once again, has a greater influence on the execution time.

5 Conclusion

After carefully looking at the results, we came to the conclusion that parallelization, although very useful and necessary in some cases, should be used in an careful and intellegent way. Before parallelizing a program it's an engineer's obligation, and it's in his best interest, to meticulously analyse the problem, and decide if the program is worth parallelizing or not. In the case of our program, that decision would depend on the usage of the program. If the user only

had input similar to tests 3 and 4, parallelization would hurt the performance of the program, and it would be better to use the sequential version. If the user used mostly input similar to tests 2 and 7, he would benefit massively from parallelization.

6 Work Division

The work was evenly split from the start of the project, since all members of the group decided what to focus their work on together. Miguel was responsible for an initial implementation/stub of the parallelization of the bombardment and relaxation stage, while maximization was divided between Miguel and Bernardo, the latter of which also corrected bugs on the previous code and found other optimizations. Miguel and Bernardo also developed the shell script to run the test battery. João was wholly responsible for both the profiling and Python scripting, and also helped with bug squashing whenever available, as well as code refactoring and cleanup. All members also participated actively in running the tests locally, and helped with the development of the bash script. Therefore, the members all agree that the work was split 33.3 percent each.

References

[1] A "Hands on" introduction to OpenMP, accessed on June the 2th of 2021, URL: https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf

[2] "Why are 50 threads faster than 4?", accessed on June the 4th of 2021, URL: https://stackoverflow.com/questions/16268469/why-are-50-threads-faster-than-4

Appendix
Test Battery and Profiling Results

Fig. 1. Test 1 Speedup

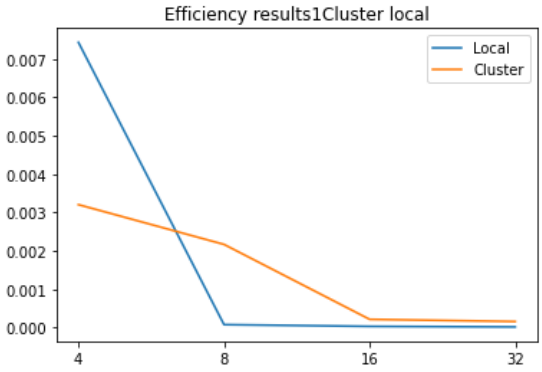

Fig. 2. Test 1 Efficiency

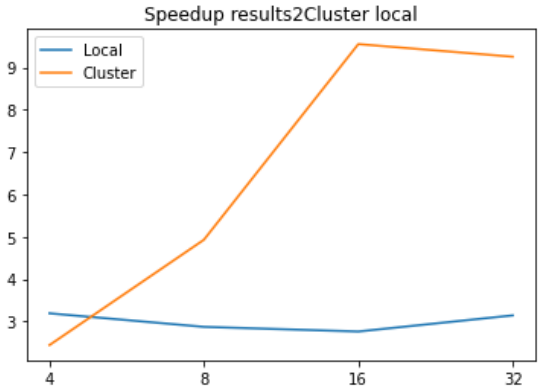

Fig. 3. Test 2 Speedup

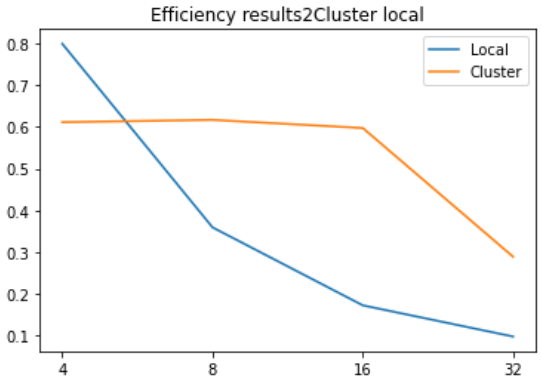

Fig. 4. Test 2 Efficiency

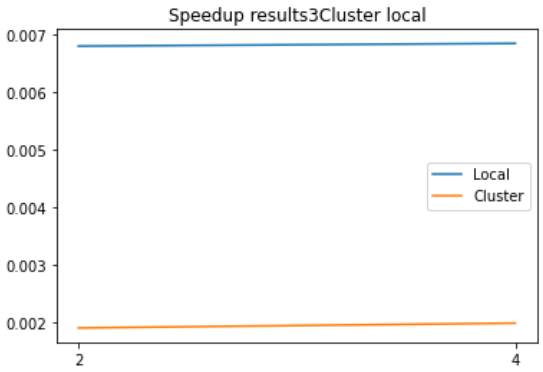

Fig. 5. Test 3 Speedup

Fig. 6. Test 3 Efficiency

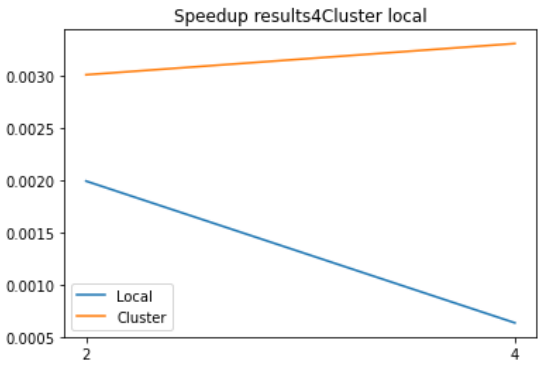

Fig. 7. Test 4 Speedup

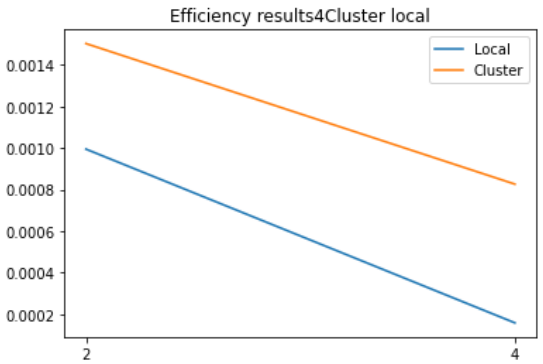

Fig. 8. Test 4 Efficiency

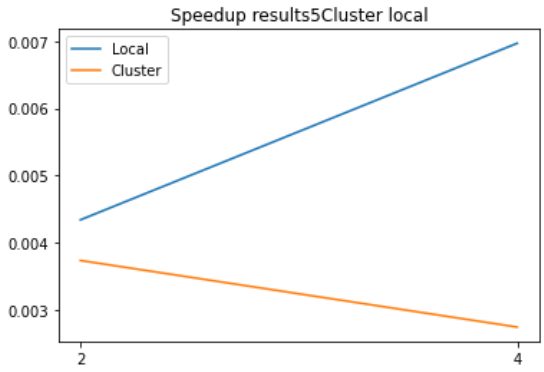

Fig. 9. Test 5 Speedup

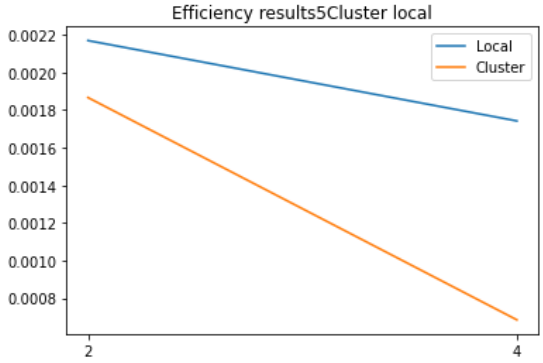

Fig. 10. Test 5 Efficiency

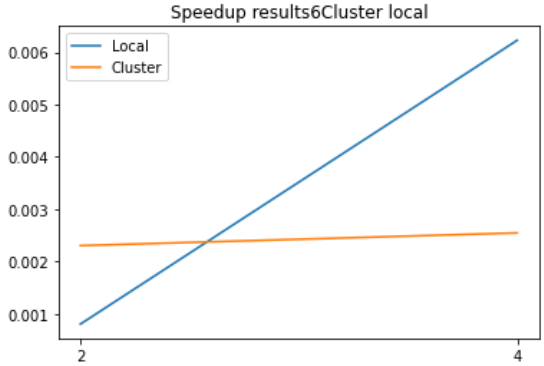

Fig. 11. Test 6 Speedup

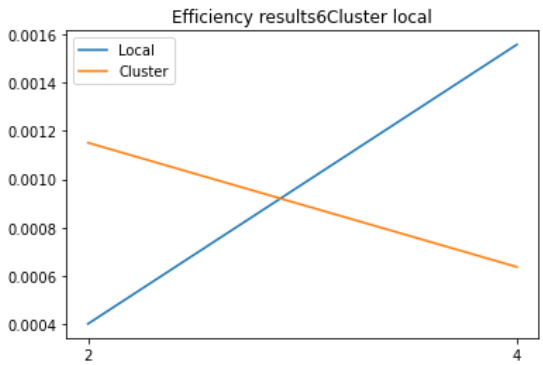

Fig. 12. Test 6 Efficiency

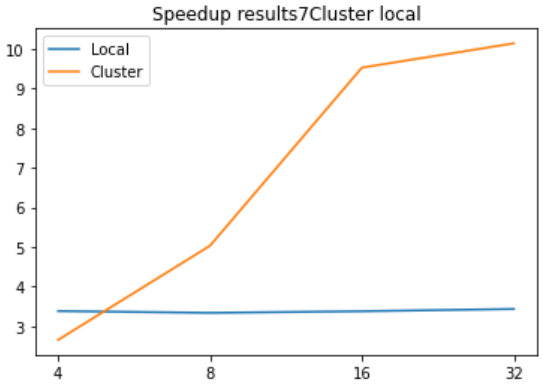

Fig. 13. Test 7 Speedup

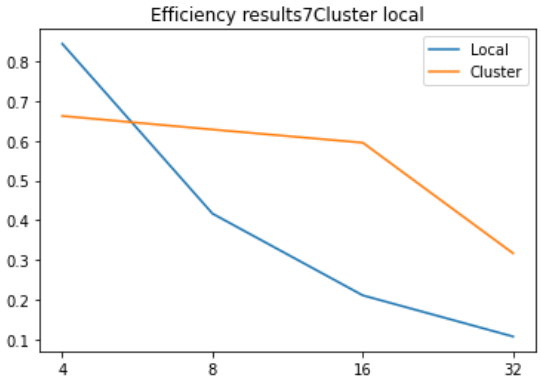

Fig. 14. Test 7 Efficiency

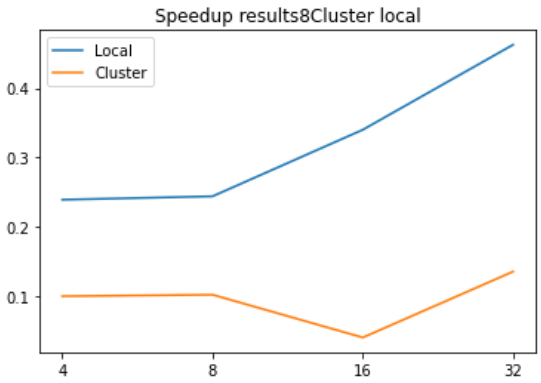

Fig. 15. Test 8 Speedup

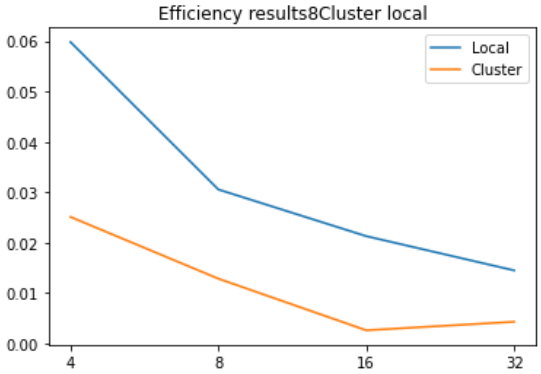

Fig. 16. Test 8 Efficiency

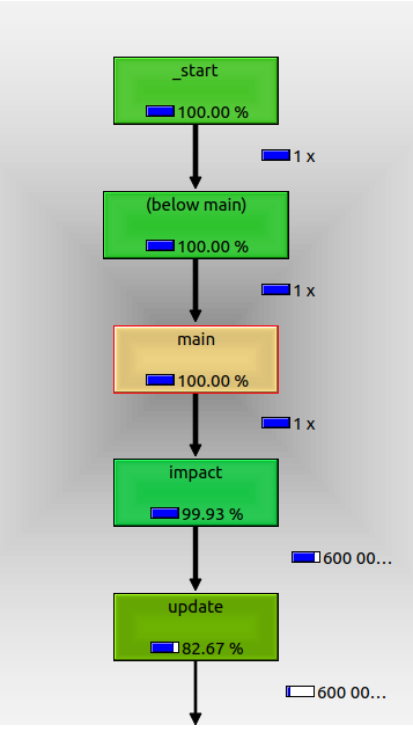

Fig. 17. Sequential Profiling

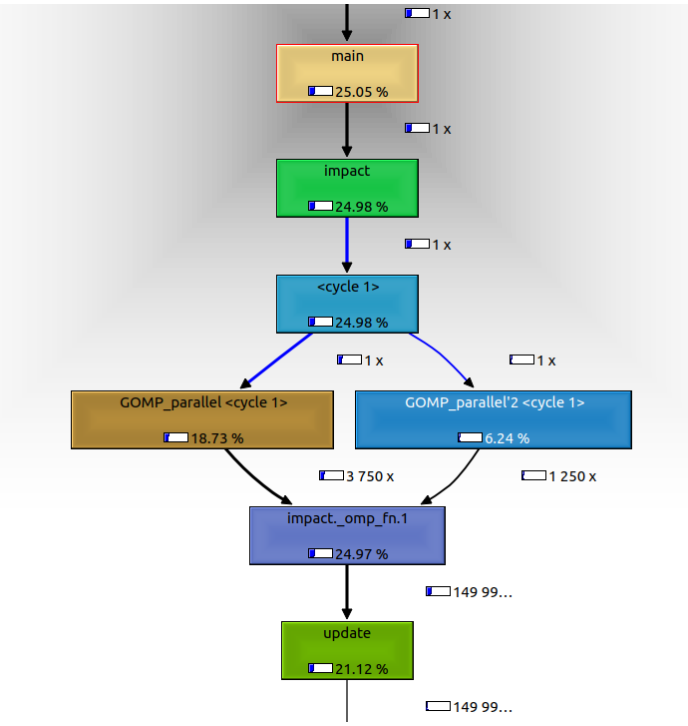

Fig. 18. Parallel Profiling