Analysis on Time Improvements with Code Parallelization

Diogo Rebimba, *d.rebimba, 46731,* Hugo Lopes, *ha.lopes, 49873,* and Diogo Escaleira, *d.escaleira, 50054*

Abstract—The abstract goes here.

Index Terms—Computer Science, Parallel and Concurrent Computing, Time Optimization with Introduction of Concurrence

1 Introduction

This report concerns a project for the course of Concurrency and Parallelism consists in the parallelization of a provided sequential program that simulates the effects of high-energy particles bombardment on an exposed surface, using the OpenMP programming model.
The program receives as input several files that represent waves of high-energy particles. It then computes the accumulated energy on each point of the surface after the impact of those waves. The program calculates and reports, for each wave, the point with the highest accumulated energy, which presents the higher risk of being damaged.
The main goal of this assignment is to improve the time performance of the source code, in particular when handling large sets of data (big number of particles). The challenge was, not only to find a way to parallelize the various sections of the sequential program, but also to analyse whether the achieved parallelizations are in fact worth it. Our approach focused mainly on the identification and removal of dependencies in the various loops present in the source code, followed by parallelization using OpenMP and analysis of the possible performance improvements.

June 6, 2021

2 Development Methodology

The following strategy was defined for the development of the work assignment: identify all loop dependencies; check the possibility of merges between loops; in the loops that have removable dependencies, remove them in order to parallelize the loops; use a profiler to check which methods have the most calls and how that affects the performance of the program.
During the development and analysis stage, this process had to be repeated to check if everything remains correctly parallelized after all the changes.

2.1 Dependency Analysis and Code Improvements

2.1.1

As previously stated, we started by identifying all the for loops dependencies. This way we could check whether each loop could be directly parallelized (using omp parallel for). In this first step, we found that:

- the loops in 4.2 (4.2.1 and 4.2.2) and 4.1.1 (the one with the update function) did not have dependencies. So, they could be directly parallelized.
- the loop 4.3 had an output-dependency and flow-dependency.
- the loop 4 has all the three dependencies (flow, output and anti).
- The loop 4.1 has an output-dependency.

2.1.2

We then tried to look for loops with the same parameters to check the possibility of a merger. In this step, we chose the 4.3 and 4.2.2 for loops: we tried several approaches, but ended up concluding that this merge is impossible due to a flow-dependency and anti-dependency, detected after merging the loops: layer[k] $= \ldots$ and if (layer[k] $>$ layer[k - 1] && layer[k] $>$ layer[k + 1]) $\rightarrow$ anti dependency in k+1, flow dependency in k-1 (line 256) As such, we returned the code to its previous form.

2.1.3

As previously acknowledged, we detected an output-dependency on the 4.3 loop for. This dependency was resolved by splitting the loop in two: a parallelizable one, that uses two auxiliary arrays and the number of the thread to calculate local thread maximums, and the second one that compares the thread maximums to find the storm maximum and its position. With this change, we were able to improve the overall performance.

2.1.4

In the last step of our strategy, we used a profiler: we experimented with multiple profiling tools (like Valgrind with the Callgrind tool as well as the CLion IDE profiling tool) and maintained the same conclusions. In figure 1, obtained by using Kcachegrind visualizer for the Valgrind-Callgrind profiler using the sequential version of the program, we obtained an understanding of which methods have the most impact on the performance of our program. With this tool we can check the entire call tree which let us know which methods consume longer processing times. The majority of the calls were to the update method and the remaining were calls to procedures concerning the reading of the files which

wasn't accounted for the times obtained. Another point we noticed from the other profiler tool (CLion IDE profiling tool) was that one of the methods that had more samples (meaning, the one that was most "called"), together with the update method, were the processes concerning the calls to omp parallel. From this information we can determine that some operations are not worth parallelizing because the gains from achieved through parallelizing such operations are surpassed by the costs of the procedures concerning the parallelization itself (like opening and closing threads).

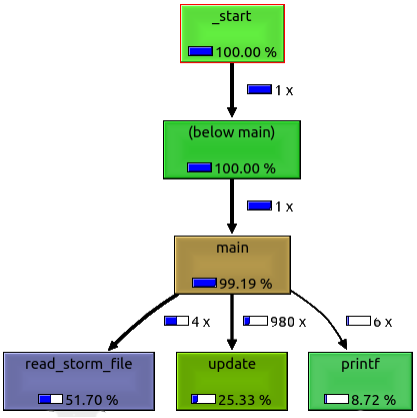

Fig. 1. Output of Valgrind profiler for sequential version

3 Testing and Results

Before running the tests in the cluster in order to get results to withdraw conclusions, we run the tests in a day-to-day use Linux machine. In this pre-tests we realise that the average standard deviation in 5 runs is very low (< 0.09), furthermore, the environment on the cluster machine is much more controlled and free of interferences (regarding timing operations) so, in order to be able to execute more tests and once the time available on the cluster was limited, we decided to run each test only 2 times.
The cluster's machine used has 4 x 8 processors and 32 hardware-threads. The tests were executed using a variable amount of threads from 1 (corresponding to sequential execution) to 32.
All the executions of the parallelized code have proven to be accurate in terms of output compared with the unchanged sequential version. So the results never changed, only the execution time. Once the main purpose of this course is the study of concurrency and it's performance advantages, the results of the executions will no be presented although they were used to check the referred accuracy.

3.1 Tests and Expectations

Test 0: Using a test file provided by another group which we duly acknowledge, the execution of this test purpose is debugging and not performance analysis. For this reason it is not executed and timed in the test batch.

Test 1: The most basic and small test, mainly used for reference timing and debugging.
Due to the small size, the anticipated results in therms of performance are not good since the times are so low they may not worth the cost of thread opening.

Test 2: This is a small workload test where we expect some gains on small amount of thread opening. The increase in the number of threads would cause too much of an overhead cost (due to creation and management) that may not be softened by the increased processing speed.

Test 3 to 5: This group of tests are presented for completeness once their main goal was to find errors in race conditions. As suggested by the professor who provided the tests, they were executed only for at most 4 threads so it was decided to execute them just in our local machine.
Due to their very small size,the workload for each thread would be very small and it probably will not make up for the thread management costs.

Test 6: Originally this test was just like the test 3 to 5. We then decided to make some changes. It still uses small size wave arrays (however, bigger then the previous ones) but the main difference is that there are a considerable amount of waves.
This test is expected to be the worst in performance given that all the optimizations made are in the calculations executed in each wave. This causes threads to be created several times to perform very small workloads.

Test 7: Decent size test. The main goal with it is to find if the optimizations done actually produced some effect on performance. It can be seen as the most balanced and close to reality test.
Before running predictions are that the increase of the thread number will produce speed-ups and, therefore, performance gains.

Test 8: Used to test the reduction efficiency of the solution, this test uses very big arrays (a lot of possible positions to be hit in each wave) but only one point (particle) per wave.
Since the solution goes goes through the entire array, it's expected that the execution uses as much time as if it had a lot of points. And consequently of the same time pattern, similar speed-ups.

Test 9: We used this test either with 16 and 17 array positions as instructed. Since this test, like test 0, is focused on checking results and finding bugs in our development, it was decided not to run it in the cluster machine to make performance evaluations.

Test 10: This big size test was knowingly created to provide the perfect conditions for speed-ups according to the changes made in the parallel version. It uses a big array with several points but in single wave.
Once there is a single file with a lot of points, it should be possible to see speed-ups every time the number of threads increases.

Test 11: This is the exact same test as *test 10* but with the wave file used twice in order to create two waves. The goal is to check how the performance degrades with the number of waves by comparing these results with the previous ones.
We expect the the drop in performance to be linear as the number of waves increases.

3.2 Results

In table 1, downwards, are the results of test execution on cluster. These are the main focus of the analysis being made

in the work.
The results obtained are consistent with those which were expected. Except for test 2 where was expected the workload to be too small to present improvements in every thread number increase but it ends up to be big enough to show some improvements.
The times of test 11 are approximately the double of the ones on test 10. This shows that, although the prediction that test 11 was going to be worst than the 10 one, it gets worst in a linear way with the number of wave files considering the files are all the same size.
An interesting and unexpected result is the similarity between the test 7 and 11. Like exposed in the previous section, these two tests are totally unrelated and neither the number of points per wave neither the number of files are the same. The only similarity is that the array size is the same. It looked acceptable to extrapolate that the size of the array is not the main cause of variance in time of execution, instead, it's probably an equilibrium point where the the bigger amount of smaller files in 7 increases the time cost by the amount of threads created for small workloads and the bigger and less amount of files in 11 have less thread management costs but more work to be done for each one o them.

TABLE 1
Output of Valgrind profiler for sequential version

Average Execution Time (on Cluster)						
	Number of Threads					
Test #	1	2	4	8	16	32
Test 2	118,16935	66,70687	31,66971	17,63705	11,22094	8,39114
Test 6	0,00004	0,00043	0,00063	0,00099	0,00245	0,00485
Test 7	641,60140	341,70539	168,95200	87,95393	49,23048	32,91953
Test 8	19,37347	11,09786	5,03233	3,07526	1,72486	1,01824
Test 10	322,04200	166,57550	84,89914	44,48637	28,86720	20,64572
Test 11	643,52096	341,23911	169,33727	87,65991	46,89829	31,70972

Bellow it's presented in table 2 the results of the tests which were run in the local machine. Like it's mentioned earlier, these are mostly presented for completeness. As registered in the cluster for test 6, all the tests chosen to be run locally see their execution times worsen as the number of threads increases because this tests' main purpose is not to evaluate the performance of the program but debugging and testing its correctness. Since the data to be computed is relatively small the parallelization costs outweigh the parallelization gains.

4 Analysis

With the time values resulted from the tests, presented in table 1, it's possible to generate de graphics presented next. These graphics allow a better visualization of results and consequently an easier way to conduct an analysis.
All tests see their execution times reduced significantly as the number of threads increases. The exception is the test 6. Beware that however the graph 2 is really good to see the time evolution among the thread number, it doesn't fully reveals how badly the increase of the thread's number

TABLE 2
Output of Valgrind profiler for sequential version

Average Execution Time (on Local PC)						
	Number of Threads					
Test #	1	2	4	8	16	32
Test 1	0,00003	0,00022	0,00160	0,00337	0,00613	0,01404
Test 3	0,00001	0,00012	0,00181	-	-	-
Test 4	0,00001	0,00008	0,00011	-	-	-
Test 5	0,00000	0,00017	0,00011	-	-	-
Test 9_16	0,00000	0,00008	0,00276	0,00040	0,00111	0,00278
Test 9_17	0,00001	0,00008	0,00084	0,00129	0,00115	0,00264

makes the execution time of test 6 to increase. Also, due to the similarity of time values for the test 7 and 11 the lines appear overlapped.

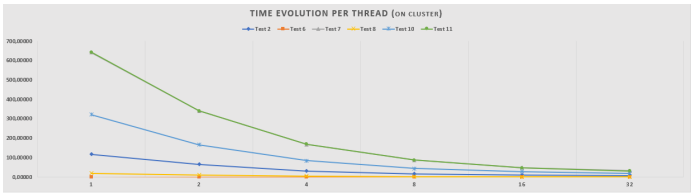

Fig. 2. Evolution of execution time for test per number of threads

Being the speed-up Sp given by the formula presented below where $T1$ is the time of the sequential execution and Tp the time of the parallel version with p threads, the following graph was built.

$$Sp = \frac{T1}{Tp}$$

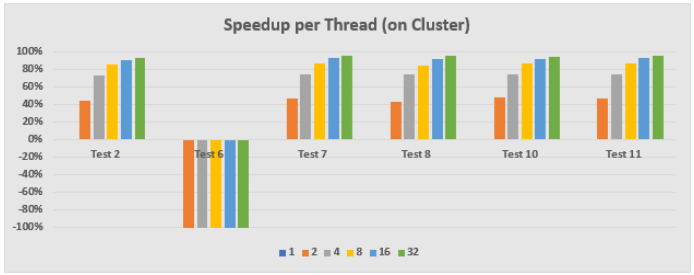

Fig. 3. Speed-up achieved with each number of threads pre test

As illustrated below in image 3, all tests (except test 6) achieved great improvements, especially when increasing the number of threads from 1 to 2 (40-50%) and from 2 to 4 (20-30%). After that, as the number of threads increases, the speed-up gains become progressively less pronounced.
In test 6 the speed-ups are really bad as predicted. They actually achieve values of around -10000% not presented on the graph for mathematical correctness. This is caused by a large number of thread management operations (creation and join/destruction) related to the number of files used as well as a small workload to be carried out by each one of them. This occurs because the paralellisation strategy was focused on the most time consuming methods (update) and

the wave/file computation wasn't made parallel.
Once the speed-up for each test-number of threads pair was calculated, it's possible to use that to estimate the efficiency *Ep* according to the number of threads *p*

$$Ep = \frac{Sp}{p}$$

TABLE 3
Computation efficiency accordingly to the number of threads

Efficiency per Thread (on Cluster)						
		Number of Threads				
Test #	1	2	4	8	16	32
Test 2	-	89%	93%	84%	66%	44%
Test 6	-	5%	2%	1%	0%	0%
Test 7	-	94%	95%	91%	81%	61%
Test 8	-	87%	96%	79%	70%	59%
Test 10	-	97%	95%	90%	70%	49%
Test 11	-	94%	95%	92%	86%	63%

It's clear that great efficiency values are achieved with 2 and 4 threads, the latter's being the best values of the whole batch. Besides that, increasing the number of threads further than 8 is quite inefficient as the speed-up gains do not measure up.
As expected and mentioned earlier, the test 6 presents very bad results.

5 Conclusion

Based on the analysis presented before, it can be concluded that it's not always a good approach to make a sequential program parallel.
The purpose of the computation as well as the kind of data to be used are two important aspects to take in consideration when paralelizing a program. The tests carried out shows that the strategy used in this work is adequate to very large files but in small amounts. If the files were a lot and very small a different approach must be taken or else it would be better to use the sequential version.
Also, the availability of powerful machines with lots of processors isn't always good. In this case with the type of data used, never seems to be profitable to use more then 8 threads given that the efficiency will only degrade and the speed-ups achieved are not that significant.

References

[1] R. Rocha, "Programação em Memória Paralela com o OpenMP," 2009. [Online]. Available: https://www.dcc.fc.up.pt/ ricroc/aulas/0910/ppd/apontamentos/openmp.pdf
[2] J. Lourenço, "Parallel Programming Models and Dependences," 2021

Acknowledgments

The authors would like to thank...

- Rúben Barreiro (42648) for asking a question on piazza which talked about a profiler that we later use
- David Pereira (52890) and his group (G28) for the python script base to generate test files
- The group who shared the *layer35_maximums_0* test file (not properly identified in the piazza post) for the test file which helped us find bugs and dependencies.

Individual Contribution

Diogo Rebimba - 25%

- Parallel testing and analysis of loop 4.1 and merge of 4.3 with 4.2.2.
- Profiler results analysis
- Data treatment from test results
- Excel graphs production for better visualization
- Report production

Hugo Lopes - 50%

- Team management
- Analysis and paralelization of loops 4.1, 4.2.1 and 4.2.2
- Creation of test scripts
- Cluster test and use
- Report production and merge

Diogo Escaleira - 25%

- Parallelization, testing and analysis of loops 4, 4.1 and 4.3
- Report production

Comments, Critics, and Suggestions

The size of the work seemed adequate to the time given as well as the difficult seemed appropriate. We wish we could have had a bit more time in the DI Cluster but we do understand the the limitations imposed.