Concurrency and Parallelism Project Report G06

Gonçalo Prates 55223, Ruben Belo 55967 and Yang Ji 55519

Abstract—This paper will cover the process and details of adapting a sequential simulation written in the C programming language to a parallel version using the OpenMP library. This includes the changes that had to be made to the existing program, the tests that were created to verify that the program functions like in the sequential version and also tests to verify if the parallelization improvements correspond to the expected results. It also discusses the decisions and choices that had to be made during this development.

1 Introduction

This report will describe the process of adapting the sequential version of a fire-extinguishing simulation written in C to a multi threaded version using the OpenMP library. It will cover the strategies used to decide what parts of the program should have priority in being parallelized, what changes were made, and how the correctness of the program was tested and confirmed.

May, 2022

2 Where to optimize first?

2.1 Using a profiler

To start, we had to find out where the hot spots were in the program's execution. To do this, we employed the use of Gprof, a performance analysis tool. The first scan didn't tell us much, since what Gprof does is showing how much of the execution time was spent in each function, and upon reading the code, we saw that there were no functions called from the main function. This meant we had to alter the program to include function calls, while changing nothing else.

2.2 Extracting methods

To make it easier to extract the functions from the existing code, it was recommended by the professor to use CLion, a C and C++ IDE. The original program was well commented, and had comments separating different sections of code, so we used these comments to help decide where to extract the methods. We extracted some methods, made a few changes to some variable types to keep everything working, and used Gprof to confirm that the execution times would be displayed. It was working, but the profiler was showing that most of the time was spent in a single function. So we divided this function further to see what parts were most critical to be parallelized. By this point, most functions were small and had simple loops. This also made it easier to add the OpenMP parallelization.

3 How to Optimize and Test

3.1 First Optimization

The biggest hot spot in the program was the part where it would update the surface values, this is, updating the float matrix called surface. What we did here was add a simple "omp parallel for", and with 2 threads, the profiler showed that the time spent in this section was reduced by about half. The results looked good, but we had to make sure that everything was working the same. So we created our first test.

3.2 First Test

Since the altered section of code changed the values of the 'surface' matrix, we had to make sure that with the same parameters, the sequential and parallel versions of the matrix always had the same values. So, for each iteration of the program we write all the values in the 'surface' matrix to a file, and we do this twice, once for a sequential execution and another for a parallel execution. In the end, if both files look the same with several tests, then there are no errors caused by this change. Our tests showed that both files were always the same.

4 Problems with Gprof

4.1 Strange Results

Even though Gprof seemed to be a helpful tool, we noticed that after introducing some OpenMP functions that we would get different results from what we were expecting. The profiler's evaluation seemed different, on a specific test, from the original one when using a single thread. We looked at older logs, and confirmed that back in the beginning, about 60% of time was spent on the function "updateSurfaceValues", and on the new log the same percentage of time was now spent in the "initSurfaces" function. This was very strange since they should be same with a single thread. We researched on the internet and found posts with similar problems when combining OpenMP and Gprof [1]. Some comments on these posts said that Gprof didn't work well with parallel programs, so we decided to look for a new way to evaluate the program.

4.2 Creating our own evaluations

The solution we thought of was to create our own program, which would tell us:

- The total time spent on each function.
- The average time spent on each function.
- The number of function calls.
- The percentage of time spent in each function.

So it should show us the important information that Gprof also analysed, but our implementation should work fine with OpenMP. Our previous work, where we separated all the functions, made it easier to work here due to the code being already divided. Since we would add more variables and more lines of code related to printing our report of the execution, we decided to create a copy of the project, to keep the original cleaner and more readable. We called this file "extinguishing_omp_report.c". This became the new main way to verify that improvements were being made.

4.2.1 Report Example

Time %	Total Time	Avg Time	Calls	Function Name
0.387601	2.190378	0.000050	43950	computeGlobalResidual
0.219762	1.241902	0.000028	43950	updateSurfaceValues
0.155377	0.878055	0.000020	43950	copyValues
0.107918	0.609859	0.000139	4395	teamActions
0.091764	0.518571	0.000012	43950	updateHeatOnFocalPoint
0.025684	0.145141	0.000033	4395	moveTeams
0.011763	0.066473	0.000015	4395	activateFocalPoints
0.000130	0.000733	0.000733	1	initSurfaces

5 Updating Tests

Our test to verify the correctness of the program was quite heavy. Comparing every value of the matrix in each iteration took some time, so we decided to only compare the last iteration, since the matrix is always based on it's previous version, if something went wrong, it's very likely to pass to next iteration. In any case, as a final test we will still use the original version of this test, just to be sure. We also made it so that this test was activated by a flag "-w". Another test was added to write the global residual of each iteration on a file, this was activated by a "-g" flag. A new program called "compare.c" was also made to compare 2 files. We used this to compare a multi threaded output file to a sequential output file. This was used this to compare both the matrix values and the global residual values, to verify that the parallel version is correct.

6 The Optimization Cycle

After parallelizing the first function, we had to repeat a similar process for more regions of the code, so that it could reach a state where more optimizations would only have a negligible impact. This process consisted of:

- Executing a test and analysing the functions where the most processing time was spent.
- Add parallelism to one of these functions.
- Test to see if the improvements are what was expected.
- Test to verify the correctness of the program.
- If all the functions where most time was spent have been parallelized, repeat the process with a test with different characteristics.

By the end of this process, we found that it was worthwhile to parallelize 5 functions. Some were very simple but the functions "teamActions" and "computeGlobalResidual" had some more nuances to them, these will be discussed in the next sections.

6.1 Global Residual

When we decided to paralyze the forsloops that change the global residuall variable we realized that it was not a trivial task, so first we decided to create a way to validate our solution and then choose the best way to paralyze the fors loops.

6.1.1 How to validate the solution?

We added a new functionality -g to the program that will write to a file GROmp.txt if the number of threads is bigger than 1 or GRSeq.txt if the program is running with 1 thread. When the results are in the files we can compare them with each other using a home-brewed compare program and if there are any differences the program will tell us the position of the error(s).

6.1.2 Solutions

- pragma omp critical
- reduction(max: global residuall)

Using both of this solutions give us the expected results but there is a catch when using pragma omp critical the time it takes to execute increases in a way there's not viable compare to the sequential time, but when using the option with reduction the time that the function takes is reduce around 66.7% with 4 threads. Other options like pragma omp critical, using arrays, etc... Are not discussed here because they don't offer a viable solution.

6.2 Team Actions

For this function many solutions were considered but after analysing and testing only one was viable. With a critical statement the problem is the same as the previous section 6.1, more standards solutions like shared(var) and reduction(action:var) are not a viable solution so we landed on an atomic statement that works wonderfully giving us the correct answer with the best time around 58.4% faster with 4 threads.

7 Best way to parallelize nested loops

There are several functions that include nested loops. In these cases we saw at least 3 different ways to parallelize, which are described in the next sections.

7.1 Outer Loop

This was our first option. It is the simplest, and consists of adding the OpenMP commands above the outer loop. In most cases this seems to be best option. What this does is splitting the outer loop amongst the given number of threads, the inner loop will execute sequentially for each thread.

7.2 Inner Loop

This option consists of adding the OpenMP commands to one of the inner loops. What this does is splitting the selected loop amongst the given number of threads, the outer loops will execute sequentially but new threads will be started in the inner ones. This generally creates more overhead with the creation and deletion of threads, and as such, is generally not the best option.

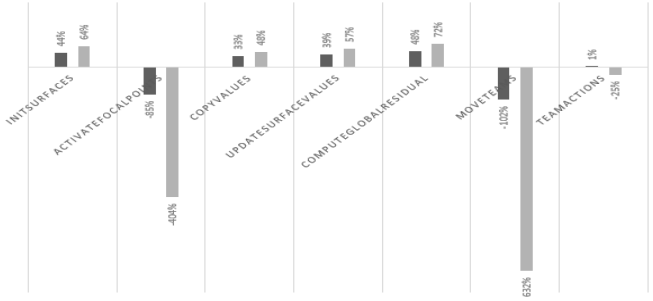
7.3 OpemMP Collapse Loop

Another interesting alternative is the collapse command that OpenMP provides. It basically joins nested loops into a single loop with the required variables. If there are no intermediate operations between nested loops, this may be a good choice. However, adding this option to our code showed a significant decrease in performance, and as such, we solely used the regular parallelization on the outer loops.

8 Some less useful optimizations

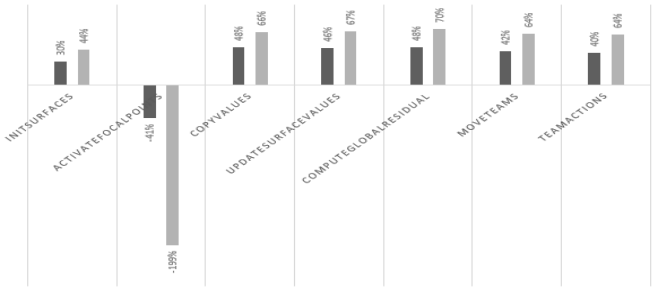
Some functions were not really using a lot of execution time, but it was stated in the assignment that some minor functions should be parallelized nevertheless. In our case these functions were: ”initSurfaces”, ”updateHeatOnFocalPoint” and ”activateFocalPoint”. The ”initSurfaces” function is only executed once, so in larger simulations it’s execution time becomes negligible. The ”updateHeatOnFocalPoint” and ”activateFocalPoint” functions only occupied a small percentage of the total execution time, because they were only loops with depth of one, while other functions had more nested loops. There is also a point to be made of the ”teamActions” and ”moveTeams” functions, these 2 functions’ impact depends on how many teams there are active in the simulation, in simulations with lots of teams they are definitely a bottleneck, so they shouldn’t be called less useful optimizations, it’s just that in simulations with few teams the performance actually decreases as shown in the following graph.

8.0.1 Performance changes with 2 and 4 threads on Test 2



This test only has 3 teams. As we can see the ”moveTeams” and ”teamActions” functions show a decrease in performance.

8.0.2 Performance changes with 2 and 4 threads on Test 3



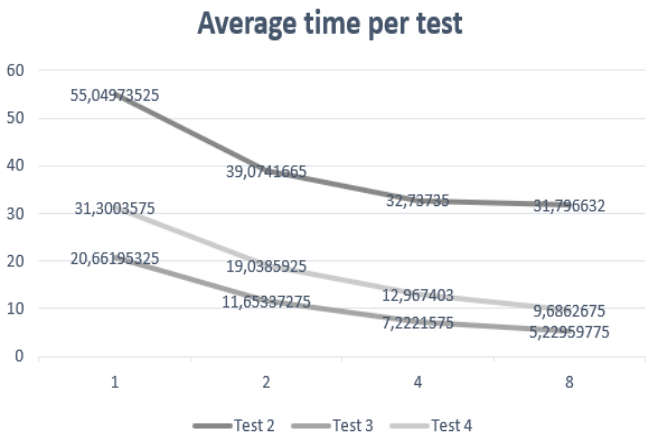
This test has 512 teams. The test results now show an increase in performance.

9 Improvements and Results

The result of our changes in the code, was that every loop we could parallelize was parallelized, this left out one loop. This means that most of the program has the ability to execute in parallel and as such, the percentage of time spent in parallel execution is very high, which should allow for great performance improvements. We didn’t manage to test with more than 8 threads, but we still have some significant results. Tests 3 and 4 showed a speed up of 4 when using 8 threads, while test 2 only has a speed up of 1.7, this is probably because test 2 doesn’t have many teams or focal points. This shows that having more teams increases thread efficiency.

The following graph shows the execution time in function of the number of threads.

9.0.1 Benchmark



10 Conclusion

To conclude, some comments could be made about our solution. To have the best performance for varying inputs we could implement a way to use statistics to choose if the function is going to execute in parallel or not. For example, when running the program, it would store the execution time for each function and when running the program again it would look at the stored data and choose what functions would run in parallel. This process is akin to cost based optimization. Since the purpose of the project was parallelization and not explicitly optimization, we think that we have made a solid solution.

REFERENCES

[1] Stack Overflow, https://stackoverflow.com/questions/36919518/gprof-on-both-openmp-and-without-openmp-codes-produces-different-flat-profile. Last accessed 23 May 2022.
[2] The Floating Point Guide, Rounding Errors, https://floating-point-gui.de/errors/rounding/. Last accessed 23 May 2022.
[3] Tutorials Point, C program to compare two files and report mismatches, https://www.tutorialspoint.com/c-program-to-compare-two-files-and-report-mismatches. Last accessed 24 May 2022.
[4] Microsoft, OpenMP Directives, https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170. Last accessed 31 May 2022.

ACKNOWLEDGMENTS

We would like to thank our colleague João Gomes for supplying some tests that helped to confirm our program's correctness.
We would also like to thank Sudhir Sharma from www.tutorialspoint.com who supplied the base for the compare program.
Finally, we want to thank the teacher João M. Lourenço for clearing our doubts in some aspects of the project.

INDIVIDUAL CONTRIBUTIONS

The division of work was not explicit, and instead, whoever had time available would make progress in some area that required more work. The only limitation was Gonçalo having a weaker computer with only 2 threads, this made it more appropriate for Ruben and Yang to run more tests in their machines, and since Gonçalo had better experience at writing he handled most of the report. The final division of tasks is presented:

Gonçalo - Wrote most of the report, made the first test (write surface matrix values to a file, to compare to sequential version) parallelized the functions "activateFocalPoints","updateHeatOnFocalPoint" and "updateSurfaceValues" and created our evaluation tool (the "extinguishing_omp_report.c" file, that prints to screen a report with details of the execution)

Ruben - Made all the extra functionality to be activated with arguments also made the -g and -t functionalities and upgrade the -w functionality.
Made the compare program that allows to compare the files produced by the -g and -w functionalities or any other 2 files as the arguments of the program.
Participated in the debug sections and helped to solve some bugs.
Made the parallelization of the rest of the functions.

Yang - Tests the correctness of the program in sequential and parallel. Changes the program in case of error.

We consider the work was shared fairly and everyone contributed around 33% of the work.

COMMENTS AND SUGGESTIONS

We think the project should have a lite more emphasis on writing code and some extra credits for different implementations, for example parallelize the program without the help of OpenMP.