# Fire Extinguishing Simulation Parallelization

Diogo Brás, Miguel França and Tiago Mendes

**Abstract**—Parallelizing a problem given its sequential solution can require some analysis and evaluation beforehand. We propose a possible parallelization solution for the problem presented by Arturo Gonzalez-Escribano, Jorge Fernandez-Fabeiro Group Trasgo, Universidad de Valladolid (Spain), with their Simplified simulation of fire extinguishing, and their sequential code written in C language. We also present some analysis using profiling techniques and generation of test cases which guided us in choosing what regions of the code to parallelize. We used OpenMP to parallelize the code and gprof as a profiler.

**Index Terms**—Parallelization, Concurrency, OMP, Speedup

## 1 Introduction

Parallelization of a program is usually defined as the break down of a larger problem, into smaller independent problems, to be ran by multiple processors, who can share pieces of memory and whose results are possibly merged at the end. There are two main goals justificative for the use of parallelization, to reduce execution time or to increase the amount of work done in the same execution time. In this assignment we focus on reducing execution time since the goal is to parallelize a fixed simulation, whose size wouldn't make sense to be increased within the same execution. This problem can prove to be quite challenging. In theory, when parallelizing a program, adding more problem division paired with extra processors would result in a faster execution. However, in practice, this hypothesis is not certain as the execution is always dependent on the hardware. Having a larger subdivision of the problem and asking for it to run in a larger amount of processors than the ones available, proves to, most times, not have a positive result based on the expectations. Furthermore, dividing the problem and distributing it to different processors during execution takes time, making it not worth to parallelize smaller problems. The amount of execution time reduction can be measured by the concept of Speed Up, which is defined as the ratio of serial execution time to the parallel execution time [4]. To determine which pieces of code to prioritize for parallelization (performance bottlenecks), a profiler can be used. Profiling is a form of dynamic program analysis, which can provide information on duration, time complexity and memory usage of function calls. [5]

June 7, 2022

### 1.1 The serial problem

The Agent-based Simulation of Fire Extinguishing is an assignment created by Arturo Gonzalez-Escribano and Jorge Fernandez-Fabeiro from the University of *Valladolid* to teach the approaches to the same problem with different parallel programming models [3]. In our case, we exploited this problem using OpenMP, a shared-memory parallel programming API.

In a very simplified way, this problem consists of the following features: activation of focal points, heat propagation, teams movement and the impact of each team on the heat surrounding them.

## 2 Method

We committed most of our work into parallelizing and optimizing the sequential code as best as we could. Our main goal was to get the highest speedup possible from the original code while not wasting too many resources and not trying to "overkill" it. Our solutions were mainly focused on basic OpenMP concepts such as simple parallelization of *for* loops and reductions. The code was divided essentially in two parts. The input reader and the actual simulation iterations and memory allocation. For the simplicity of analysis, on the subsequent sections we only measure the time it takes for the program to run the second part.

### 2.1 First Improvement

We started our analysis of the C code by performing two simple code optimizations, not related to parallelization.

The first optimization was to use a technique similar to double buffering. "A programming technique that uses two buffers to speed up a computer that can overlap I/O with processing. Data in one buffer are being processed while the next set of data is read into the other one." [6]. This way we can keep reusing the matrices we already have in order to avoid doing repeated work or transferring the data from one matrix to the other.

The second optimization was a simple change in the way the distance is computed on multiple iterations of the loops. Instead of using the square root function, *sqrt*, from the standard C library we compute the squares of both values that are being compared in order to avoid the call to that heavy computing function.

### 2.2 Profiling & Parallelization

To profile our code we used *gprof*, a profiling tool to gather statistics about C programs.

Profiling allowed us to identify where the program's time was being spent and which lines were being called the most while also taking the most time running.

Having different test cases for this part is crucial. So after creating a small program in the Java programming language that creates an environment for the simulation given some parameters (horizontal and vertical size of the grid, the

number of iterations, the number of teams and the total number focal points) we generated several test cases with each one focusing on potential slow down aspects of the code.

With each test case we found a new portion of the code that was taking the most significant time to run. Following these results there must be some analysis done, before trying to parallelize a certain loop. This analysis consists in evaluating the dependencies between the loops and iterations and loop carried dependencies. So in order to obtain a satisfying result in parallelization these must be well optimized.

### 2.3 Finding sequential cut-offs

After knowing how and where to parallelize the code, some more evaluation is needed in order to be able to generalize the program to run the most efficiently in the majority of cases. To do this we need to find the sequential cut-off for each parallelized code portion.

For this part we considered that the only possibilities were to either run each portion of code fully sequentially or parallelized always with the same number of threads throughout the program. So no dynamic management of the number of threads was done, in order to find the necessary/optimal number to run a certain parallelizable loop needs. However we tried using OpenMP's default dynamic adjustment of the number of threads and the results turned out to be worse as it usually doesn't find the best number of threads for a specific implementation.

In order to find the sequential cut-off we created another batch of test cases. One batch for each of the previous problems found. In each batch we tested different values, for the variables in which the loops depended, and discovered when it was worth parallelizing or running sequentially each code portion. We did this by analysing in what value did the time spent running the sequential code was starting to be worse than parallelizing it. From that point onwards, parallelizing the code was always worth it. Using the OpenMP's keyword *if* inside the *#pragma omp* statement we were able to decide at runtime if the code following that should be parallelized or not.

## 3 Experiments

After running the initially provided tests, several times, we constructed the following execution time table:

| seq | Time (secs) |
|---|---|
| test1 | 0,0016 ± 0,0001 |
| test2 | 88,1755 ± 1,1333 |
| test3 | 31,8986 ± 0,6057 |
| test4 | 40,0977 ± 0,9189 |
| test5 | 0,0041 ± 0,0002 |

TABLE 1
sequential version

This table shows us the average time that the initial sequential code takes on each test and the corresponding standard deviation.

After making the first improvements on the sequential version referred in 2.1, we proceeded to run the same tests, several times, again and constructed a new table:

| impSeq | Time (secs) |
|---|---|
| test1 | 0,0014 ± 0,0000 |
| test2 | 67,3883 ± 0,9312 |
| test3 | 25,2386 ± 0,0936 |
| test4 | 30,8513 ± 2,1725 |
| test5 | 0,0037 ± 0,0003 |

TABLE 2
improved sequential version

This table shows us that even minor tweaks on a sequential code can make a big difference in the final result, as a time costly function or routine can be called multiple times during a cycle.

After analysing multiple test cases and finding multiple performance bottlenecks 2.2, we proceeded to parallelize them. Several tests were used to verify correctness and the same initial tests were used to construct the same execution time table as before:

| omp | Time (secs) |
|---|---|
| test1 | 0,0037 ± 0,0001 |
| test2 | 36,6096 ± 0,1848 |
| test3 | 15,4412 ± 2,2126 |
| test4 | 25,1347 ± 0,2415 |
| test5 | 0,0067 ± 0,0001 |

TABLE 3
omp parallelized version

This table satisfies our initial expectations, where the parallelized version would be faster on larger and more time consuming problems but slower on smaller problems, due to overhead of threading resource consumption.

### 3.1 Sequential cut-off

With only the section 4.4 of the provided code parallelized, relative to the improved sequential version, we proceeded to run multiple tests where only the number of teams would vary. This made it so we could recognize the differences in execution time as an improvement or not of that same section. We obtained the following table:

| n teams | impSeq | 2 threads | 4 threads | 8 threads |
|---|---|---|---|---|
| 20000 | 5,411925 | 6,040235 | 5,734174 | 5,30323 |
| 25000 | 6,032743 | 7,165107 | 6,253542 | 6,222421 |
| 30000 | 7,588712 | 8,71174 | 7,415538 | 7,344133 |
| 45000 | 18,525552 | 15,910451 | 12,610695 | 12,695659 |
| 50000 | 20,554327 | 19,351132 | 14,885857 | 15,135402 |
| 100000 | 49,179453 | 41,850883 | 28,680078 | 29,119419 |
| 300000 | 85,603824 | 81,588832 | 54,032353 | 54,601846 |

TABLE 4
Sequential cut off table for parallelization on 4.4

With this table we can conclude that for around 30000 teams the parellelized version starts to be faster. Even though the distinction between threads was not necessary in this step, it helps us strengthen the belief that for the particular machine where the tests were being ran, the ideal number of threads was 4.

Using a python ploting script, we are able to create a graph and calculate a estimation of the intersection point between the execution times:

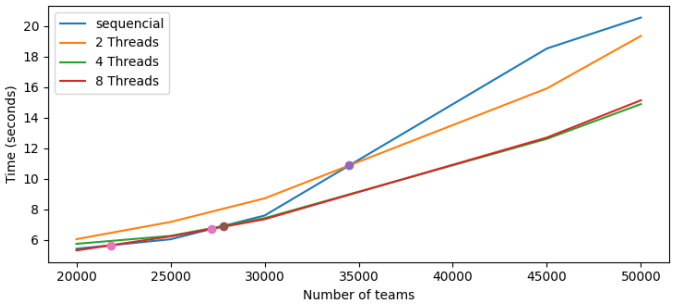


Fig. 1. Sequential cut off graph for parallelization on 4.4

In this case the intersection between the sequential and 4 threads lines was on X = 27802, leaving us with a good estimation for the cut-off on this section.

The same process was repeated for all other sections. Here we demonstrate the experimentation on section 4.3, just to show that the results can be very different from one another. Once again tests were made with only this section parallelized and differed only in meaningfully values.

| n teams x n focal p | impSeq | 2 threads | 4 threads | 8 threads |
|---|---|---|---|---|
| 10000 | 1,737427 | 1,744368 | 1,784506 | 1,780357 |
| 250000 | 1,695187 | 1,624315 | 1,698878 | 1,663076 |
| 640000 | 1,723713 | 1,654085 | 1,731241 | 1,6684 |
| 1000000 | 1,835826 | 1,721149 | 1,783247 | 1,735846 |
| 10000000 | 5,511364 | 3,955861 | 3,405557 | 3,325923 |

TABLE 5
Sequential cut off table for parallelization on 4.3

Here we have the multiplication between number of teams and focal points as the varying value. This was due to the results of tests with the same size being distinctive based on the difference between the two values, although maintaining the improvement ratio.

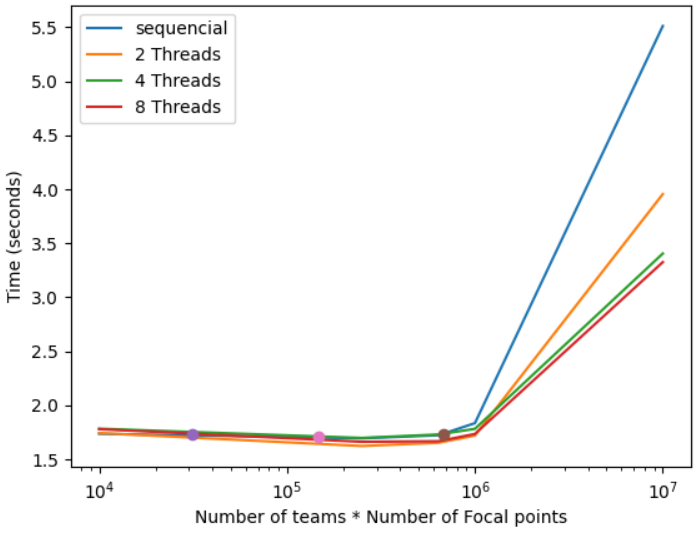


Fig. 2. Sequential cut off graph for parallelization on 4.3

In this case the intersection was on X = 685087, leaving us with a good estimation for the cut-off on this section.

## 4 Results

With the final version of the parallelized code we can finally estimate how efficient our work is. To do this we will use Amdahl's Law to calculate the maximum speedup possible and compare that to our current speedup. Amdahl's Law is a formula used to estimate the maximum speedup a program can have knowing it has a fraction that is sequential and another that is parallelizable.

The Amdahl's Law formula is:

$$S = \frac{T_1}{T_n} = \frac{1}{\frac{F}{n} + (1 - F)} \tag{1}$$

n = number of processors
F = fraction of the program that is parallelizable
$T_n$ = time it takes to run with n processors
S = maximum speedup with n processors

But since we don't know the fraction of the code that is sequential or parallelizable we must first estimate the value of F. Continuing the previous equation (1):

$$\frac{1}{S} - 1 = F\frac{(1-n)}{n} \tag{2}$$

solving for F we have:

$$F = \frac{\frac{1}{S} - 1}{\frac{1-n}{n}} = \frac{n\,(T_1 - T_n)}{T_1(n-1)} \tag{3}$$

Now we can make a rough estimation of the final F value if we compute for a given range of processors the F value of each by replacing the n in equation 3 with the number of processors being used, and taking the average over all of them.

$$F_i = \frac{n_i}{(n_i - 1)}\frac{T_1 - T_{n_1}}{T_1}, i = 2..N \tag{4}$$

$$\bar{F} = \frac{\sum_{i=2}^{N} F_i}{N-1} \tag{5}$$

We ran these next calculations for test2 because this test is for general purpose and we think it is a good performance checker. Our results for evaluating $F_i$, with $i$ up to 8 threads, are the following:

| Nº Threads | Time (secs) |
|---|---|
| 2 threads | 45,8781 ± 0,2627 |
| 3 threads | 40,0148 ± 0,7038 |
| 4 threads | 36,6096 ± 0,1848 |
| 5 threads | 39,2482 ± 0,5213 |
| 6 threads | 38,2533 ± 0,4624 |
| 7 threads | 38,6664 ± 0,4085 |
| 8 threads | 38,1343 ± 1,3244 |

TABLE 6
Results for running Test 2 with multiple threads

| Nº Threads | $F_i$ |
|---|---|
| 2 threads | 0.6384 |
| 3 threads | 0.6093 |
| 4 threads | 0.6089 |
| 5 threads | 0.5219 |
| 6 threads | 0.5188 |
| 7 threads | 0.4972 |
| 8 threads | 0.4961 |

TABLE 7
Computed F value for each number of threads

The resulting $\bar{F}$ value is **0.5558**, that is the average over all the values in table 7.

Now that we have an estimate for how much of our program is parallelizable we can measure the value for the maximum speedup achievable. Amdahl's Law can also give us the maximum possible speedup, which serves as an upper bound for the best possible outcome from parallelizing this problem even if we add more processors. We compute the maximum speedup by using the following:

$$S_{max} = \lim_{n \to \infty} S = \frac{1}{F_{\text{sequential}}} = \frac{1}{1 - F_{\text{parallel}}} \tag{6}$$

With this formula our result for the Maximum Speedup possible is **2.251**. Now we can easily compute our current best speedup by simply taking the run with the number of threads which we had the lowest execution time and computing the ratio between the time it takes to run the sequential program and that time, for the same test case. Our current best speedup is **1.8407**, with 4 threads.

If we now compute the ratio between our speedup and the maximum speedup,

$$1.8407/2.251 = 0.8177$$

this tells us that we were able to parallelize around **82%** of everything that we could have parallelized in the most efficient way possible.

If we then consider the very initial implementation of the sequential version, without any improvements, we reached a final speedup of around

$$88,1755/36,6096 = \textbf{2,39544847}$$

which is significantly higher. This result has no relation with the maximum speedup calculated as this was done for the initial problem without any code improvements.

### 4.1 Running in a cluster

It is always good to test the same program in different machines to evaluate the results, and if you have a cluster at your disposal its even better. We were given the chance to run our work on the faculty cluster.

The results were the following for the test2: .

| test2 - cluster | Time (secs) |
|---|---|
| 2 threads | 53,3515 ± 0,2619 |
| 4 threads | 49,2901 ± 4,4044 |
| 8 threads | 34,0447 ± 2,6857 |
| 16 threads | 28,2745 ± 2,2506 |
| 32 threads | 24,9491 ± 1,3416 |
| 64 threads | 34,0447 ± 2,6857 |

TABLE 8
Results of running Test 2 on cluster

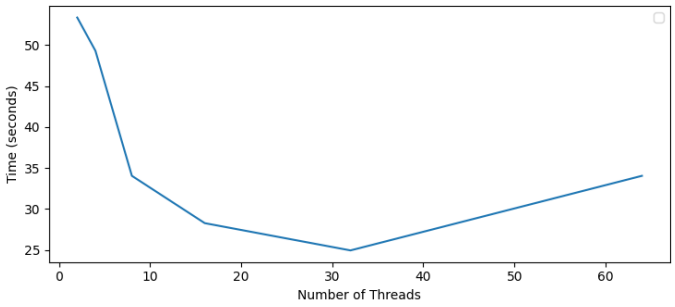


Fig. 3. Results of running Test 2 on cluster - Graph

As expected the results for running the same parallelized program in the cluster were better overall. As the cluster has a larger amount of processors than the machine we initially tested on, the problem can be subdivided even more into smaller problems, making it possible to run even faster than before, using a larger number of threads. These results have a speedup for the cluster of around 3.489.

## 5 Conclusion

After following the professor's recommended methodology we found that maintaining a good work ethic and rhythm was fairly easy.

Analysing the results, we are confident to assume our parallelization solution was adequate to the hardware it was tested on. Achieving 82% of the esimated maximum speedup, we assume our parallelization was successful and met the assignment's expectation. We are aware experimentations and optimizations were done for a single machine and values as number of threads and sequential cut-offs would need to be tweaked depending on the hardware used.

We were able to meet our expectations on the majority of accomplished experimentations, only during sequential cut-off testing did we have to rethink some possibilities, as the initially obtained results were confusing.

Hardware exists.

## Acknowledgments

We would like to acknowledge group 23 and 25 as they were very helpfull to discuss and compare all the experimentation results between us, including profiler outputs, section parallelization and sequential cut-offs.

## Individual contributions

We followed the recommended work methodology by the professor. No specific tasks were assigned to each group member, as the work evolved, each member knew what to do next and continued where the previous member had left. Sometimes work was done in parallel aswell, but following the same idea.

**Relative contributions:**
Diogo Brás (35%), Miguel França (35%), Tiago Mendes (30%)

## REFERENCES


[1] GNU gprof, The GNU Profiler
https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html
Last accessed 6 Jun 2022

[2] OPENMP API Specification: Version 5.0 November 2018, 2.19.5 Reduction Clauses and Directives
https://www.openmp.org/spec-html/5.0/openmpsu107.html
Last accessed 6 Jun 2022

[3] Agent-based Simulation of Fire Extinguishing: An assignment for OpenMP, MPI, and CUDA/OpenCL
https://tcpp.cs.gsu.edu/curriculum/sites/default/files/ws_eduhpca107s2-file1.pdf
Last accessed 6 Jun 2022

[4] A. Grama, A. Gupta, and V. Kumar, "Isoefficiency Function: A Scalability Metric for Parallel Algorithms and Architectures", IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice, Volume 1, Number 3, pp 12-21, August 1993.

[5] Profiling in computer programming
https://en.wikipedia.org/wiki/Profiling_(computer_programming)
Last accessed 6 Jun 2022

[6] PC Mag, Double buffering
https://www.pcmag.com/encyclopedia/term/double-buffering
Last accessed 6 Jun 2022