Report on Parallelization of Agent-based Simulation of Fire Extinguishing

João Vargues 55185
NOVA School of Science and Technology
Email: j.vargues@campus.fct.unl.pt

José Murta 55226
NOVA School of Science and Technology
Email: j.murta@campus.fct.unl.pt

Diogo Rodrigues 56153
NOVA School of Science and Technology
Email: dt.rodrigues@campus.fct.unl.pt

***Abstract*—The present document is a report for the course of Concurrency and Parallelism. The goal of this project is to study and develop parallelization on a agent-based simulation of fire extinguishing using OpenMP. In this report it will be presented our implementation, examination of the results and discussion of the theoretical reason behind them.**

1. Introduction

The project consists on a agent-based that computes a simulation of fire emergencies and the behaviour of the extinguisher teams. A simple rectangular canvas is used to represent the simulation area, where several focal points and extinguisher teams are displayed. The focal points are precise points where a fire takes place at a determined coordinate and time frame, and when triggered, the temperature of the surrounding points will increase. The goal of the firefighter teams is to extinguish the closest fire. The simulation ends when there are no more active fires (the global heat residual is bellow a specific threshold) or it reaches the number of desired iterations.

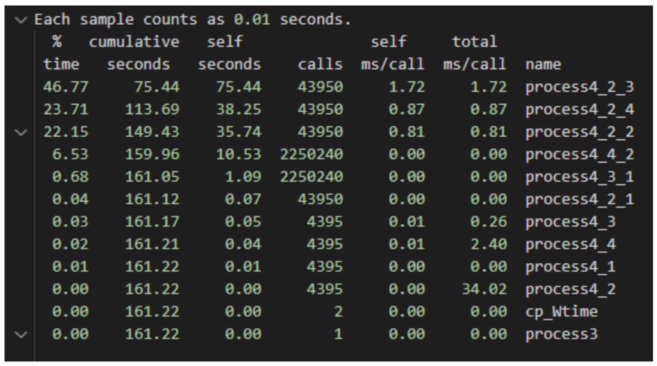
The provided sequential code is not performance optimal since it is implemented using several cycles to trigger the focal points and fire extinguishing teams behavior. The time to perform these cycles, grows when the number of these two entities increases and consequently the overall time also increases.

To solve this problem, the best solution is to parallelize the implementation. The API that we used to perform parallelization was OpenMP. During the development of our solution, we seek to improve the program's performance achieving the same output results by carefully analyzing dependencies and critical zones between running threads.

2. Looking for bottlenecks

To analyse and visualize bottlenecks (where the program is spending more resources) in the provided code we chose to use a profiler. A profiler displays information to better understand the program in a timing environment. Comparing to Java where it is available some IDES incorporated with profilers such as NetBeans, in C the process is more complex. We used *gprof* that was pre installed on our Linux distribution. To use it, the flag *-pg* must be used when compiling with gcc. After running the executable file, a file with profiling information in raw form is created. In order to create a readable file the gprof command must be executed using the previous files [1].

To be able to visualize properly the profiling information, it was necessary for us to isolate inside functions the different steps of the program. This was done in a different branch from the main one, with the goal to keep an available profiling tool throughout the entire development that enabled us to evaluate the quality of our parallelization choices. In Figure 1 it is possible to observe a snapshot before any changes were made on the code (Sequential version).



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 46.77     75.44    75.44    43950     1.72     1.72  process4_2_3
 23.71    113.69    38.25    43950     0.87     0.87  process4_2_4
 22.15    149.43    35.74    43950     0.81     0.81  process4_2_2
  6.53    159.96    10.53  2250240     0.00     0.00  process4_4_2
  0.68    161.05     1.09  2250240     0.00     0.00  process4_3_1
  0.04    161.12     0.07    43950     0.00     0.00  process4_2_1
  0.03    161.17     0.05     4395     0.01     0.26  process4_3
  0.02    161.21     0.04     4395     0.01     2.40  process4_4
  0.01    161.22     0.01     4395     0.00     0.00  process4_1
  0.00    161.22     0.00     4395     0.00    34.02  process4_2
  0.00    161.22     0.00        2     0.00     0.00  cp_Wtime
  0.00    161.22     0.00        1     0.00     0.00  process3
```

Figure 1. Initial Profiler Output

3. Solving bottlenecks

When parallelizing a program it is crucial to reflect on what portions should be targeted, not only considering time consumption but also the code that can actually be parallelized and if that it is the case, if it shows an overall improvement.

Analysing the results from the profiler, it was possible to verify which of the steps were the most time consuming and probably a bottleneck. Those were our main candidates to parallelize and next will be described.

Heat Propagation

Inside every iteration of the simulation there are 10 steps where the heat propagation is calculated. Carefully analysing the code, it can be seen that inside those 10 steps there are four cycles calculating focal points update, copy of values between surface and the auxiliary surface, update of the surface points' heat values and compute the global residual. We decided to define a parallel zone (using the statement *#pragma omp parallel*) surrounding the 3 last mentioned cycles because it avoids launching and killing threads for each cycle, reusing the existing ones.

In the first two cycles that copy and update surface values there are no loop carried dependencies since values that are wrote are never read and vice-versa. To implement our approach, it was necessary to use the *#pragma omp for* and *private (j)* statement in order to privatize the iteration variable of the inner cycles, and consequently not be shared between threads and waste unnecessary time.

For the third cycle that calculates the global residual, it is being computed a maximum between the last surface and the new one. So it was necessary to use the same statement as before to parallelize the cycle. It was also necessary to define a critical zone on the global residual variable, using *#pragma omp critical* since this variable is shared between all threads. Our first approach was to place it inside the loop, but the performance was poor because the overhead of critical zones was added in each iteration. The final solution was to use a private variable for each thread and define a smaller critical zone just to check if that is the global maximum, adding no more than just one overhead from a critical zone.

Team movement

For every simulation's iteration, each team looks for the closest active focal point and moves one position towards it. A parallel zone was defined (using the *#pragma omp parallel* statement) wrapping the teams movement code. This behaviour was originally supported by a single cycle but we decided to divided it into two simpler ones, since we noticed a performance improvement implementing it this way.

In the first cycle that chooses the nearest active focal point from each team, no loop dependencies were identified and for that reason, it was only required to use the *#pragma omp for* to distribute the loop iterations between the threads in combination with the *private (j)* statement to privatize the iteration variable of the inner cycle.

To parallelize the second cycle, that controls the movement of each team, a similar process to the first cycle was used, however the privatization statement is not applicable since there is no inner loop.

Team actions

Inside every iteration of the simulation, the teams look to see if they got to the focal point in order to deactivate it and reduce the heat around themselves. Our first approach was to parallelize only the cycle that reduces the heat around the team, but this showed to not be the most optimal performance.

We choose to place *#pragma omp for* on the cycle that goes through each team in order to distribute the teams action for each thread. This with the combination of *private (i,j)* statement in order to privatize the iterations variables of the inner cycles.

4. Results

When analysing results it is important to have in mind two key factors: time improvement and the simulation continuing to produce the right results.

In Figure 2 it can be observed the time comparison between sequential and OpenMP versions on our local machine. The machine has 16GB of RAM and 8 threads. For the parallel version we are using a upper bound of 6 threads utilizing the *omp_set_num_threads* routine.

We can see that in most tests, a time improvement happened, being now about two to five times faster parallelized compared to sequential. Bigger tests that have a higher number of focal points and fire extinguishing team showed the best overall results, contrary to tests with a small number for the agents, that were barely better in the sequential version compared to the OpenMP one.

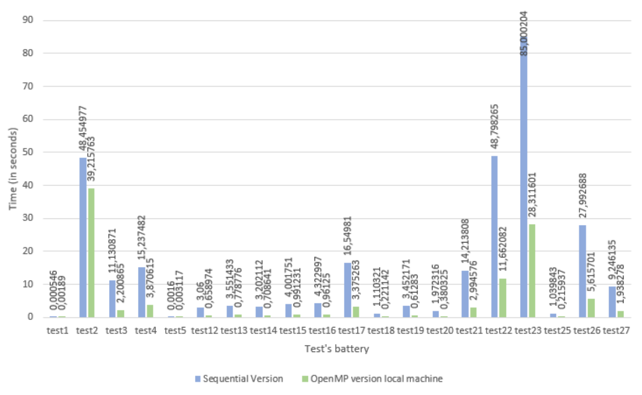


Figure 2. Average time comparison between sequential and OpenMP version - Local machine

In figure 3 it is possible to visualize a comparison between sequential and OpenMP version running on the provided cluster on the *Magikarp* node. This machine has 27GB of RAM and the 16 available threads were used in this case.

A smaller test's battery was used, however it is visible that the results are similar to the ones observed previously (executing on our local machine), and the time improvement is still noticeable. The motive why a minor number of tests are annotated in the graphic is that we spent our 120 minutes available on the cluster testing with the maximum code variations possible and try to achieve the best solution.

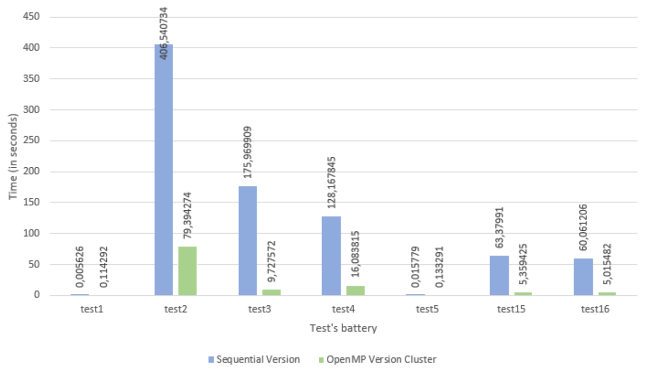


Figure 3. Average time comparison between sequential and OpenMP version - Cluster

5. Discussion

In this section we will explain the reason behind some of the choices we made and explain why some of the possible alternative solutions would not be optimal.

The cycles that initialize both surfaces are only done once before the iterations. Because of that, parallelization of this part would not bring benefits in the running code since the process of spawning and killing threads would be more costly that running it sequentially.

Parallelizing the iterations would be negative as well because it would give too much work on each thread, making this not the optimal solution. But the main reason that make it not possible to parallelize are loop carried dependencies, since this cycle would read and write in each iteration the values from the original and auxiliary surface.

The cycle that activates the focal points in each iteration could be parallelized but after some tests we decided not to. This is due to the few number of tasks that this cycle performs. A similar decision was made for the cycle that updates the heat values on the active focal points, and for that reason this cycle was left out the parallelization zone made to compute the heat propagation.

After a deep analysis and reflection, as well as several tests performed on different solutions that we have explored, we chose the implementation that presented the best time performance and achieving the expected output results.

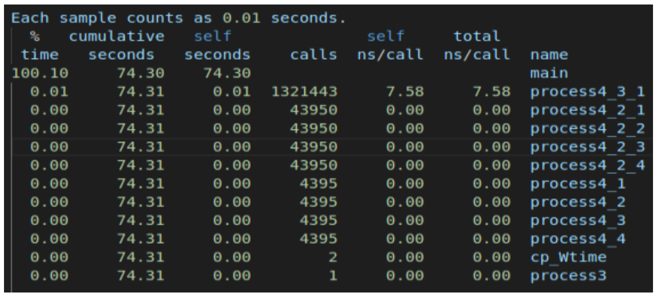

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
100.10     74.30    74.30                             main
  0.01     74.31     0.01  1321443     7.58     7.58  process4_3_1
  0.00     74.31     0.00    43950     0.00     0.00  process4_2_1
  0.00     74.31     0.00    43950     0.00     0.00  process4_2_2
  0.00     74.31     0.00    43950     0.00     0.00  process4_2_3
  0.00     74.31     0.00    43950     0.00     0.00  process4_2_4
  0.00     74.31     0.00     4395     0.00     0.00  process4_1
  0.00     74.31     0.00     4395     0.00     0.00  process4_2
  0.00     74.31     0.00     4395     0.00     0.00  process4_3
  0.00     74.31     0.00     4395     0.00     0.00  process4_4
  0.00     74.31     0.00        2     0.00     0.00  cp_Wtime
  0.00     74.31     0.00        1     0.00     0.00  process3
```

Figure 4. Final Profiler output

In the Figure 3 we can visualize the profiler of the final output. It is possible to observe that the times in each process improved by some margin when compared with the first image of the profiler and that now the sequential part of the project is responsible for almost 100% of the project running time. We just want to note that the images of the profilers provided in this report were done using the test 3 given along with the code. We used this test since it has a good amount of focal points and fire extinguishing teams, what enabled us to analyse better the results and tests done during the process of developing our solution.

The profiler we used was not compatible with the optimizer utilized when compiling. For that reason we did not used the optimizer and that is why we got some worse time results compared with the results when running with the optimizer.

In smaller tests, parallelization did not improve time. This happened because when parallelizing, threads are always launched and killed, and for tests that took little time running, this process did not compensate and although the times got worse, it was only by a few decimal points. This in a long run of very different simulations with varied surface sizes, number of teams and focal points, the percentage of time that got better is always going to compensate this few milliseconds loss. To solve this problem we tried to use the statement *if* to just carry parallelization whenever some Boolean expression is met. But after implementing and testing we concluded that after all, using it was not making a notable difference and decided to not keep it in the final solution.

6. Conclusion

Parallelizing is an approach that when correctly used, can present major performance benefits, being specially better when developing applications for large problems. The agent based simulation of fire extinguish, like explained in the introduction, had some problems of high sequential execution times. Our parallelization job definitely helped to achieve some better performance, having some tests, for example test2 when running on the cluster provided, that went from executing in almost 7 minutes to 1 minute and some seconds.

Considering all the possible implementations that we have done and tested, we believe that the one presented is the most ideal. The time spent implementing, testing and searching to achieve the best solution possible really helped us to put in practice all the information provided during the semester on this course.

Individual contributions

The work throughout the entire project started with a few days of code revision for each member, then after it, each presented implementation ideas and some doubts that appeared. The ideas were discussed and the program was distributed by the members. All in all, we agree that each member had the same amount of workload, complied with their tasks, participated in the final choices' discussion and report elaboration. So, João Vargues did 1/3, José Murta did 1/3 and Diogo Rodrigues, as well, did 1/3.

Acknowledgments

The authors would like to thank the professor for providing the use of the cluster, since that helped us to have a different point of view in our results and the group that shared their battery of tests in Piazza because it gave us more opportunities to test our implementation.

References

[1] GNU gprof Manual
https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html
Last accessed 5 June 2022