Lab5: Geometric (data) decomposition using implicit tasks: heat diffusion equation Q2 2021-22

Integrantes:
Pol Pérez Castillo - par2116
Cristian Sánchez Estapé - par2118

Fecha de entrega: 31-05-2022

Índex

Introduction

In this laboratory, we will analyze the potential parallelization of two algorithms used to solve an equation related to the diffusion of heat on solid objects. To achieve it, we will approach such a problem using implicit tasks and other mechanisms that will be duly described in the present document. Two sections might be kept in mind: the first one will touch upon the Tareador instrumentation of the initial (sequential) code, which will give us information about the dependencies present in each solver, a necessary element that ought to be taken into consideration in section two. In regards to this second part, it will present the parallelization for both solvers through both efficiency plots and their corresponding *Paraver* simulations.

Sequential heat diffusion program and analysis with Tareador

Is there any parallelism that can be exploited at this granularity level?

Some brief explanation must be made in order to understand the obtained graphs: in regards to the Jacobi case, since the solution is computed in a temporary matrix, it is required to copy such result to the solution matrix, which is the reason for the relation between the *jacobi* and the *copy_mat* tasks observable in figure 1. For this particular strategy and granularity, there's no considerable parallelism that can be exploited (since the workload of each task hasn't been duly decomposed). In regards to the Gauss-Seidel case, since the solution is directly computed in the solution matrix, there's no intermediate computations (as in the previous case), which gives place to the graph presented in figure 2. For this strategy, too, there's no parallelism to be exploited, thus impeding the optimization of such tasks (and program).

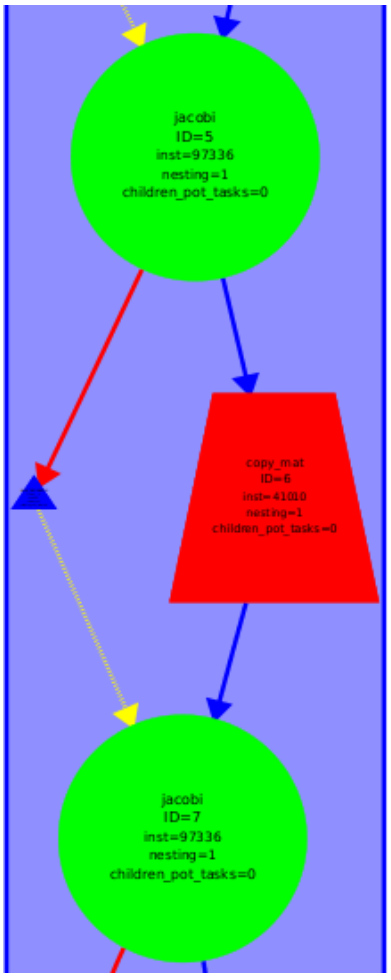

Figure 1: initial tareador instrumentation of the jacobi solution

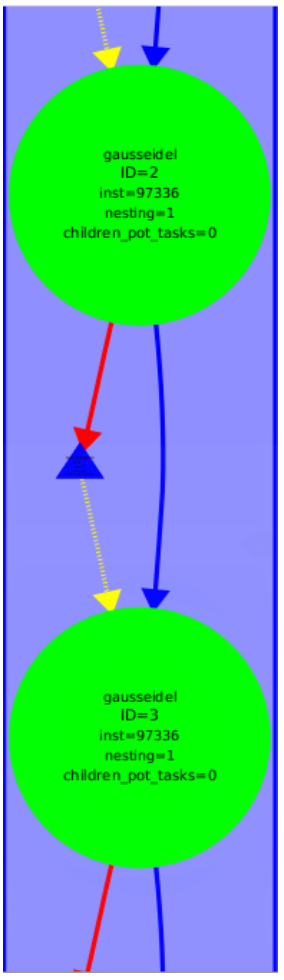

Figure 2: initial tareador instrumentation of the gauss-seidel solution

Which variable is causing the serialisation of all the tasks?

For both cases, the *sum* variable is the one which causes such sequentiality: for the *Jacobi* case, given that there's no dependency between rows and columns (we must bear in mind that such feature is taken into consideration by the *copy_mat* function, rather than the *solve* function itself), the *sum* variable forces such dependencies since the traversal of the matrix is done by columns. However, in regards to the *Gauss-Seidel* case, the dependencies (which in this case are: for each block, we must wait for the block in the i-1 row and the j-1 column) end up causing a different set of relations that are determined by *sum* nonetheless. It must be stated that both types of dependencies are observable in figures 3 and 4.

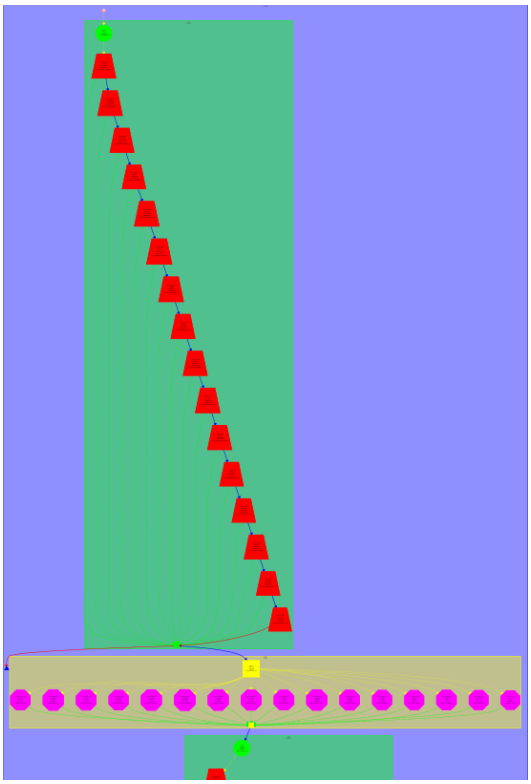

Figure 3: tareador instrumentation of the jacobi solution

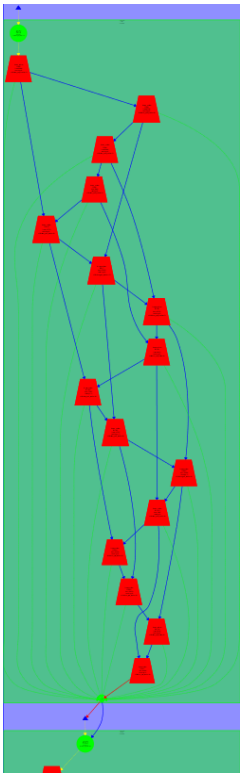

Figure 4: tareador instrumentation of the gauss-seidel solution

Are you obtaining more parallelism? How will you protect the access to this variable in your OpenMP implementation?

Both *Jacobi* and *Gauss* are achieving a better performance in terms of parallelism of the program (figures 5 and 6 are great examples of such improvement over parallelism, while figures 7 and 8 are the simulated execution of the program itself), given that we have procured to inhibit the dependencies provoked by the *sum* variable. Still, the *Gauss* TDG keeps presenting unpreventable dependencies provoked by the access order to the matrix (a problem which is inexistent for the *Jacobi* case, in which case we treat such dependencies in the *copy_mat* function, as it has been previously stated).
Moreover, one possible mechanism to protect the access to the variable for the *Jacobi* case would be the use of the directive *reduction(+:sum)*; as for the *Gauss-Seidel* implementation, the creation of an auxiliary data structure that allows threads to know which block has already finished would be necessary.

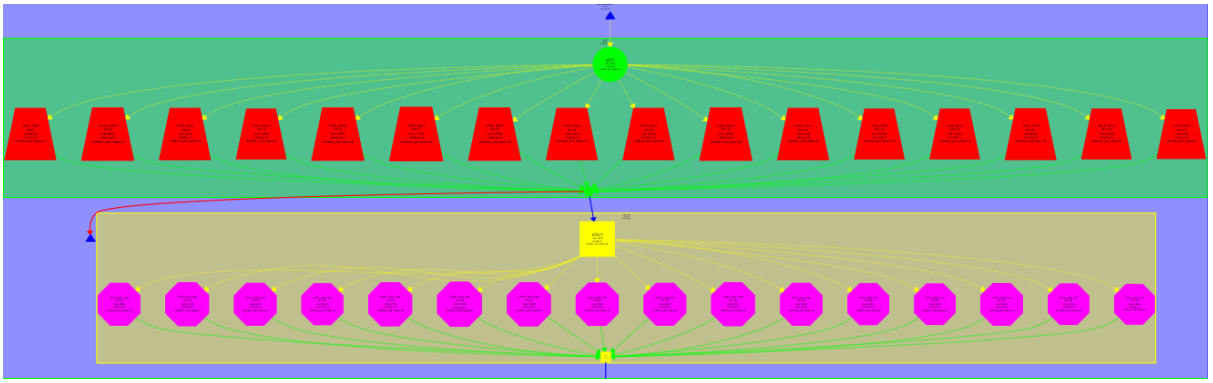

Figure 5: tareador instrumentation of the jacobi solution with sum disabled

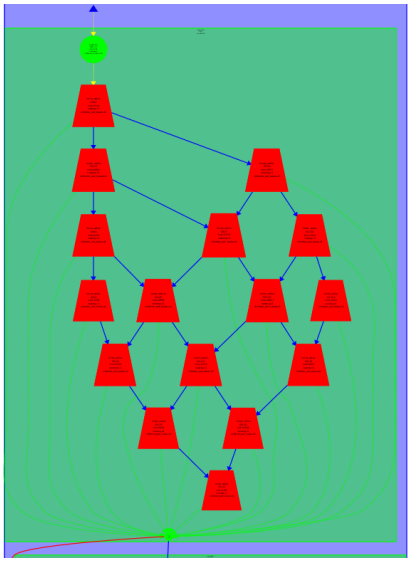

Figure 6: tareador instrumentation of the gauss-seidel solution with sum diabled

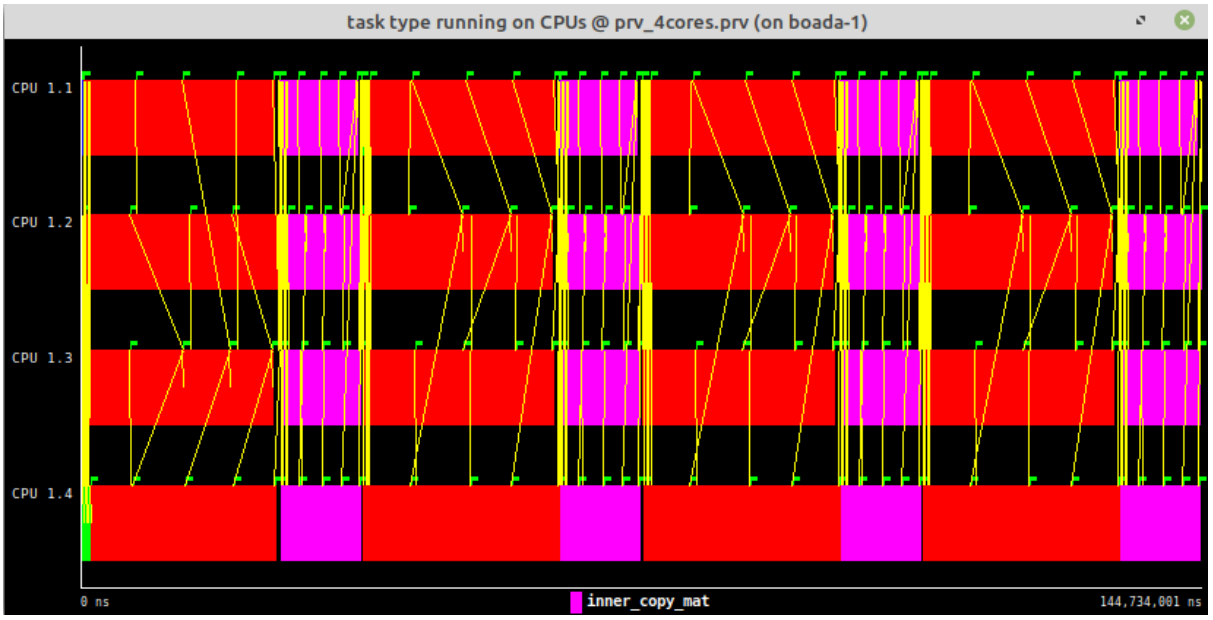

Figure 7: simulated execution with 4 threads for jacobi solver

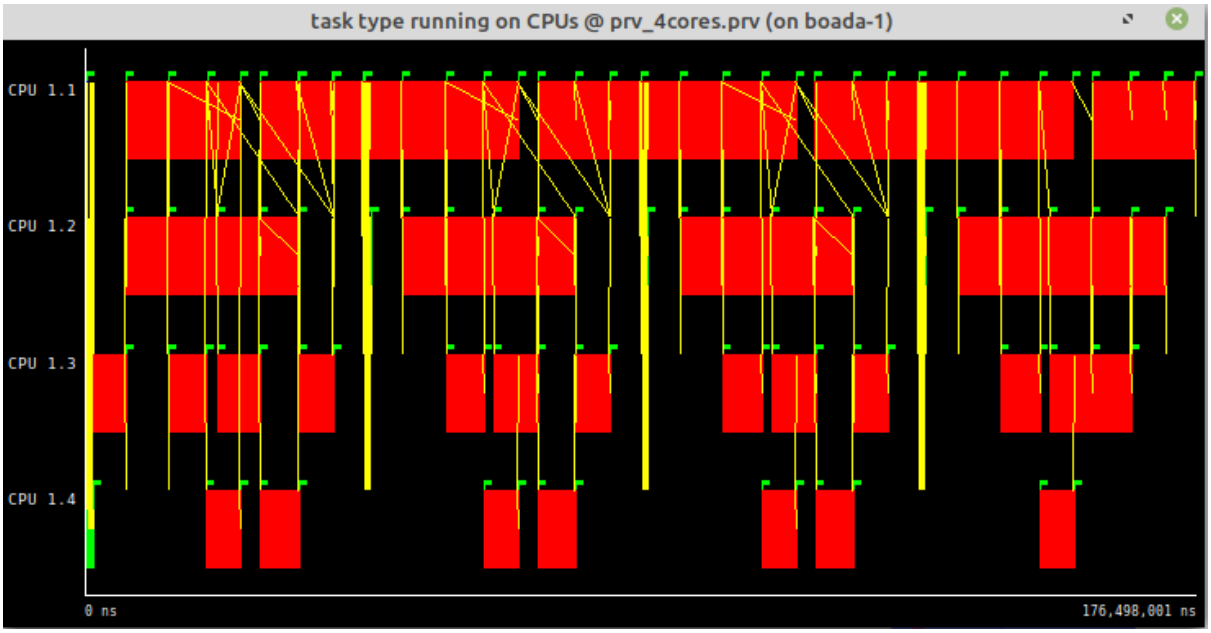

Figure 8: simulated execution with 4 threads for gauss-seidel solver

Parallelisation of the heat equation solvers

Jacobi solver

Is the scalability that is obtained with this initial parallelisation appropriate?

The scalability obtained from the first parallelization of the program (only the *solve* function had been parallelized) performs poorly, given that a considerable amount of workload is still being computed sequentially (by the *copy_mat* function) and the parallelized portion isn't big enough to make an improvement in the overall performance. Such phenomena can be contrasted with figures 9 and 10, which are results obtained from such parallelization. Moreover, figure 11 displays the features of the execution of the parallelized program: there's a superior chunk of time dedicated to scheduling/fork in contrast with the time dedicated to the execution of the program itself.

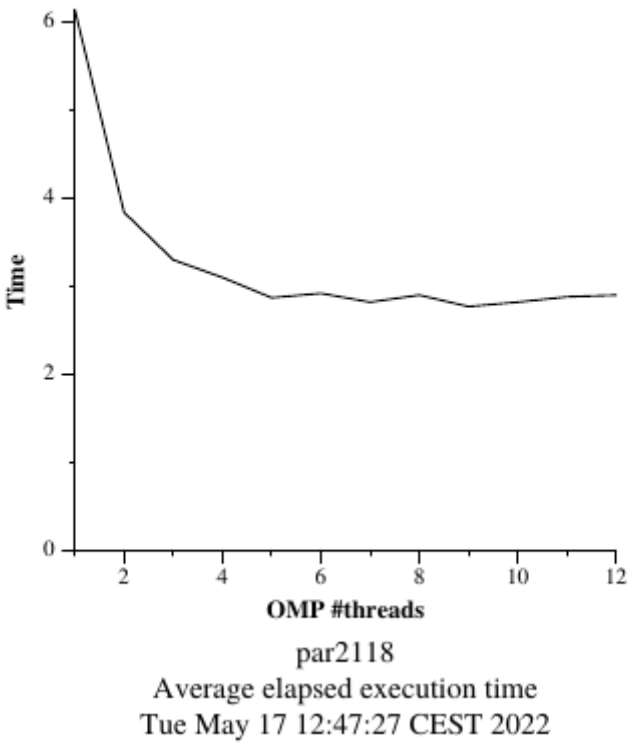

Figure 9: execution time of the program (solve parallelised)

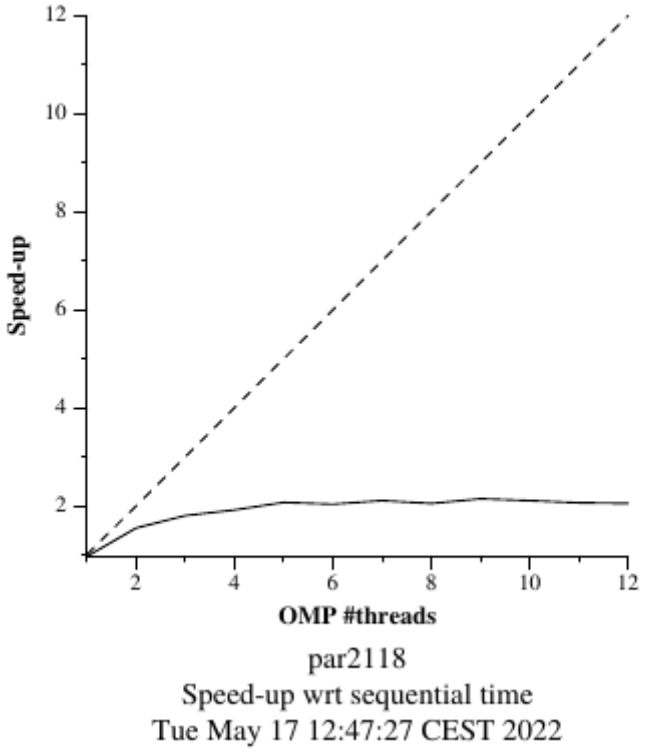

Figure 10: speed-up of the program (solve parallelised)

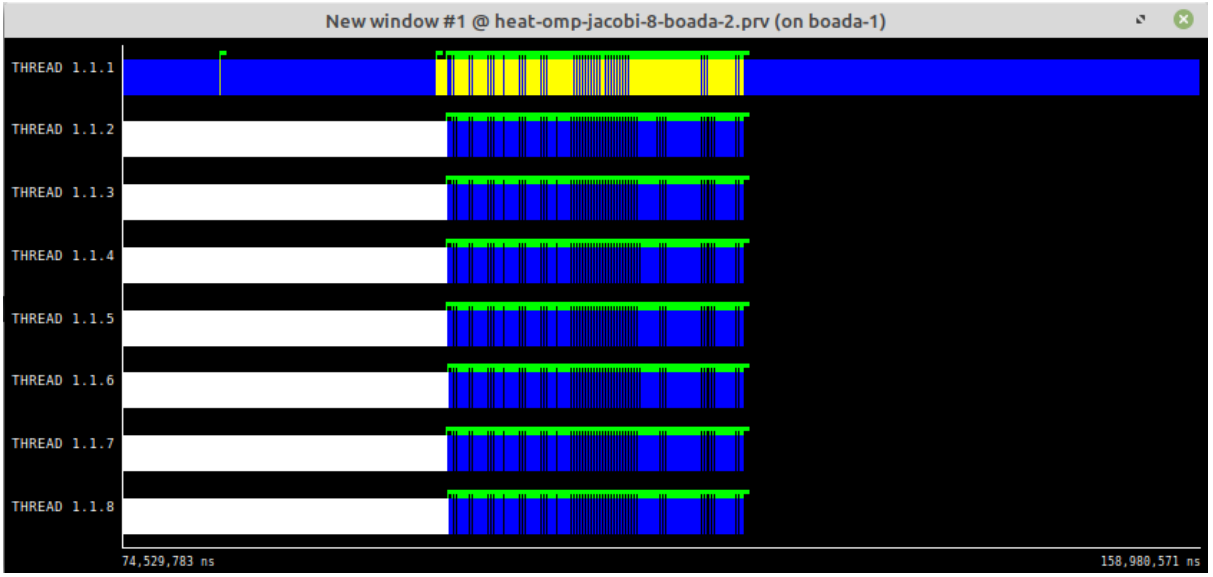

Figure 11: paraver window of the program (solve parallelised)

Is there any serious serialisation in your parallel execution? Is the execution time reduced? Parallelise other parts of the code in order to improve the efficiency of your parallel code. Is the execution time reduced?

As it has been stated above, there is a serious serialization problem, since the *copy_mat* function still does its computations in a sequential manner. To treat such a problem, we should parallelize both *solve* and *copy_mat* functions.
As for the newer version, there are no more problems in regards to the *solve* and *copy_mat* functions given that, just as we had detected with the Tareador instrumentation, the two functions that, for their derived dependencies, forced a sequential execution, have been duly parallelized. Moreover, with this said parallelization, we achieve a most-considerable reduction in terms of the execution time of the program (both in regards to the execution of the functions themselves as well as the overall execution; for this matter, figures 13 and 14 might be checked).

Why the new code that you have parallelised makes the difference in the performance results?

Just as it's been stated, the previous versions of the program had a lower percentage of workload that had been parallelized. With this new version of the code (both functions involved in the computation of the solution) we achieve a considerable increment of the parallelized workload, thus giving place to results observable in figure 12, in which there's no predominant chunk of fork/scheduling tasks anymore, and the computation of the solution itself is what characterizes such implementation of the program.

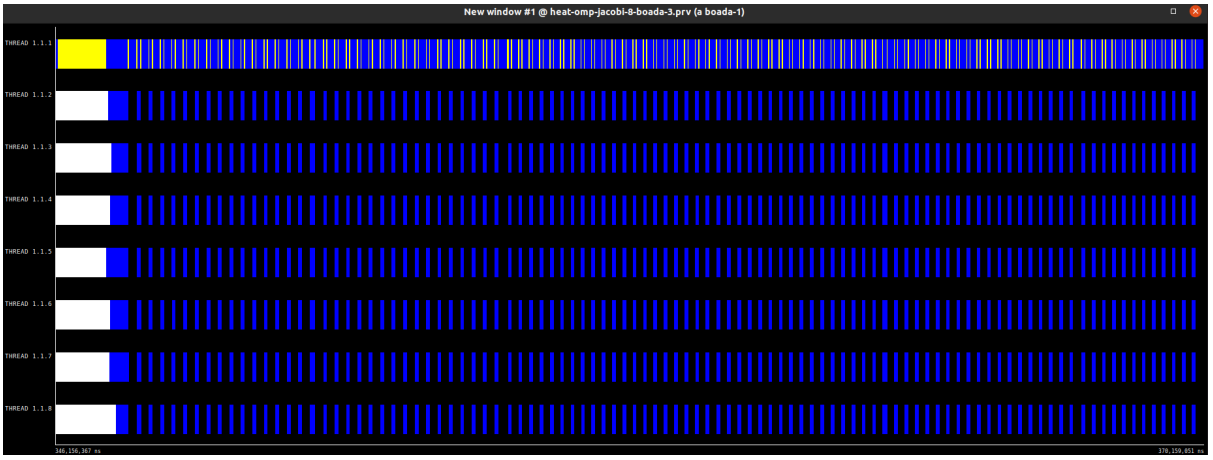

Figure 12: *paraver window of the program (solve and copy_mat parallelised)*

Reason about the scalability that is obtained.

In the scalability previously obtained, there was a minor improvement over the execution of the program. Such a small magnitude of the result had been attributed to the fact that the computed solution (for the Jacobi case) also required the *copy_mat* function to copy the solution matrix to another one, and without parallelizing said function, the execution of the program wouldn't yield the expected performance improvement.
However, once said function had been duly parallelized (on the grounds of the results obtained through the Tareador instrumentation, which can be checked in figure 3), a considerable improvement over the performance of the overall execution of the program is

obtained: one must bear in mind that, just as we had seen with Tareador (figure 1), the *copy_mat* function has a workload virtually equivalent to that of the *solve* function. So, with all of this in mind, and through the parallelization implemented with implicit tasks, we achieve an optimal result to solve the problem through the *Jacobi* method (figures 13 and 14 might be check, once again, to see the improvement over the performance of the solution computed by said method).

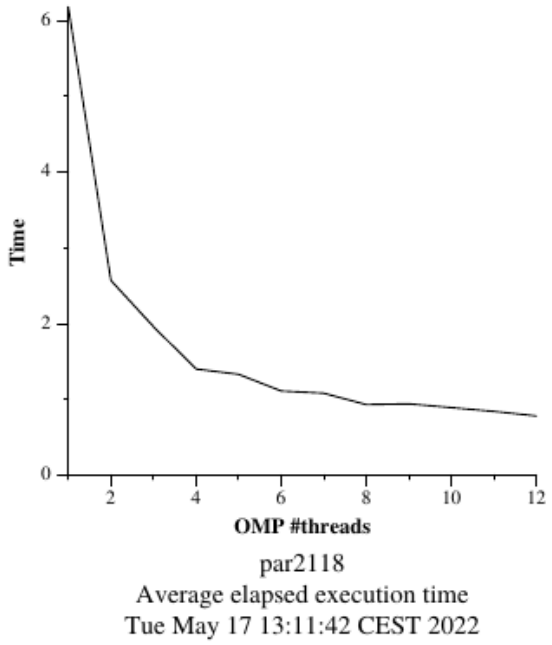

Figure 13: execution time of the program (solve and copy_mat parallelised)

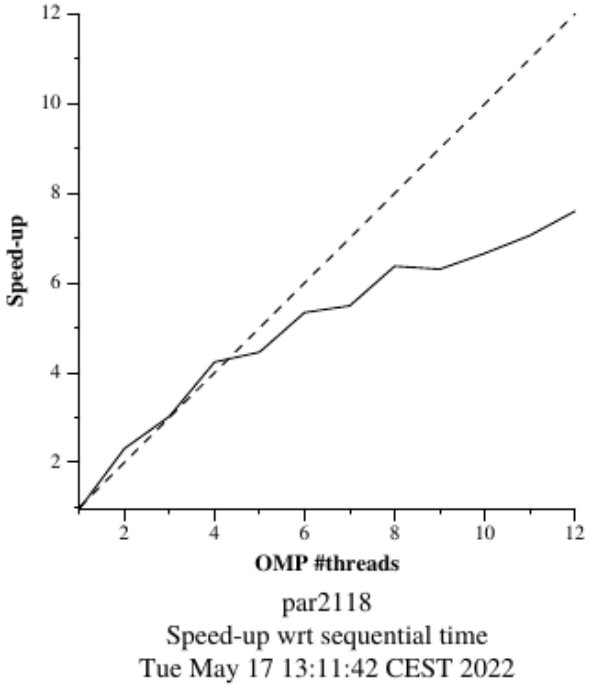

Figure 14: speed-up of the program (solve and copy_mat parallelised)

Gauss–Seidel solver

Does the parallel behaviour match your expectations?

For the *Gauss-Seidel* method, the obtained results match (in a relative manner) our expectations: on the grounds of the Tareador instrumentation previously presented (both figures 4 and 6), there are unavoidable dependencies that have to be taken into account no matter what parallelization one tries to implement. With that in mind, we approach the parallelization of said method in the following way: given that each block of *blocki* rows and *blockj* columns (which will represent a task) depends of the task in the previous column (that is, it depends on the *blocki* rows and *blockj - 1* columns task) and also depends of the task in the previous row (that is, it depends on the *blocki - 1* rows and *blockj* columns task), we will have to establish mechanisms that allow a task to know that its dependency condition has been fullfiled and that it is possible to continue (or begin) its execution. Now, having said all of this, and implementing such a mechanism with a matrix of [num_threads][16] (to evade problems such as *false sharing*, for example; something that must be stated is that we have 16 columns since 16*4 corresponds to the size of a cache line, thus allowing us to evade the already stated inefficiency), we will have to keep in mind that, even though our parallelization could be excel·lent, said unavoidable dependencies will most possibly imply that such method won't be as efficient as that of *Jacobi*.
Now, having said all of this, if we compare the performance of our implementation (figure 15) with that of the *Jacobi* method (figure 12), we can already see that, for the *Gauss* case, there's a slightly bigger amount of execution time dedicated to fork/scheduling, rather than to execution, which translates to a slight decrease in the execution time of the overall program, which is to be expected (and also verified) in figures 16 and 17: the overall execution time of the program is always superior to that of the *Jacobi* method (as it can be compared with figure 13) and the speed-up of the program scales less with the amount of threads used in its execution (which is not the case, again, for the *Jacobi* method, as it can be seen in figure 14).

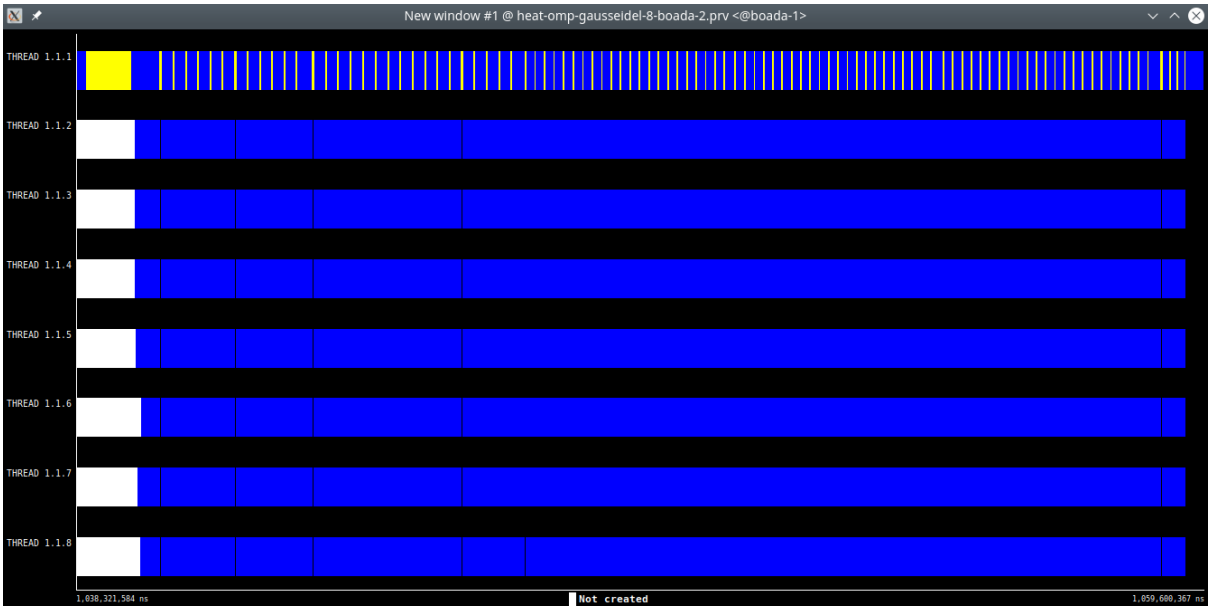

Figure 15: paraver window of the program (solve parallelised)

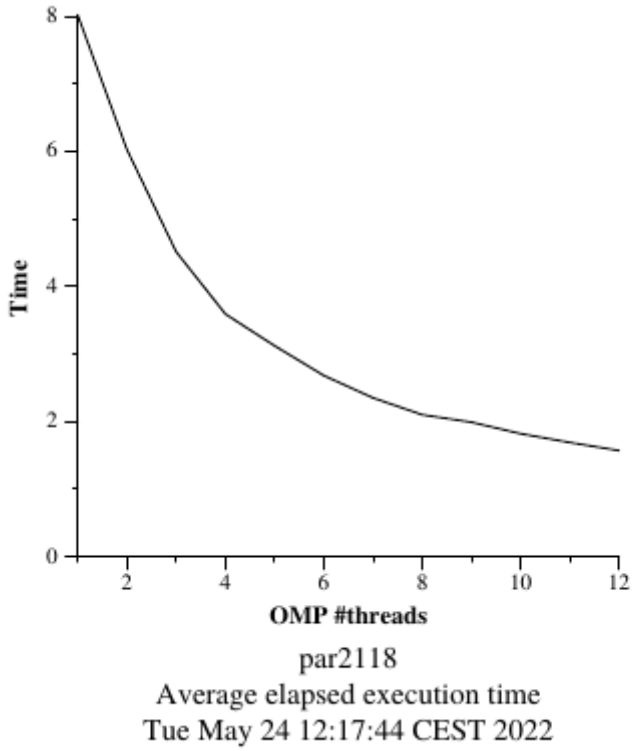

Figure 16: execution time of the program in relation to the number of threads used

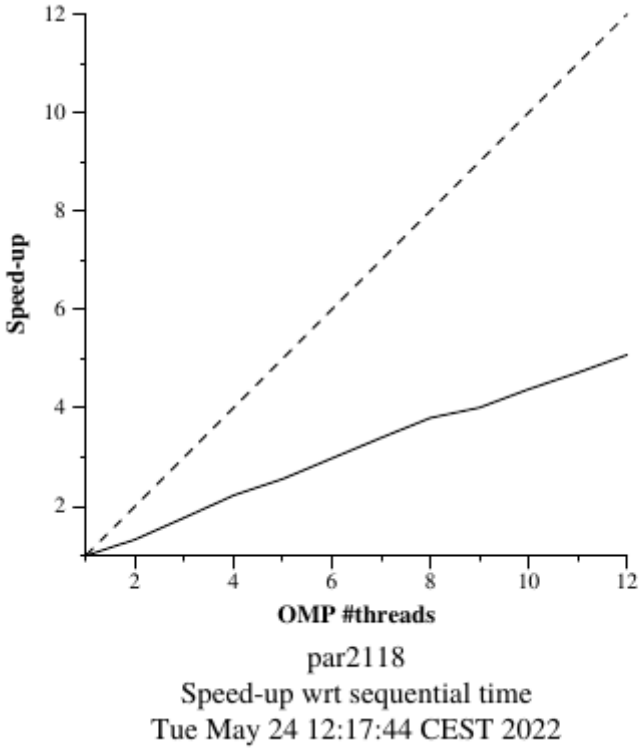

Figure 17: speed-up of the program in relation to the number of threads used

Changing the number of blocks in the `j` dimension changes the ratio between computation and synchronisation, why? For the execution with 4 and 8 threads plot how the execution time varies and explain the plot that is obtained.

Changing the number of blocks in the J dimension has an effect on the ratio between computation and synchronization, since the size of a task (which we already stated that was defined as a set of *blocki* rows and *blockj* columns) depends on the value of both *i* and *j* dimensions, which also translates into the amount of time that subsequent threads have to wait to compute their assigned tasks; for smaller *j*s, we should expect a decrease in the execution time of the program (until the overhead of creation of both threads and tasks starts to provoke inefficiencies), and for bigger amount of *j*s, we should expect an increase.
To check if such expectations are really as stated, figures 18 and 19 represent the commented ratio, for values 4 and 8: for a *userparam* of 4 (bear in mind that *userparam* now represents the value of *nblocksj*, which defines the total amount of columns to be traversed), we can detect that the initial amount of time is smaller than that of the plot for a *userparam* of 8, and it keeps being that way until 5 threads are used in the execution of the program, in which the execution time is smaller for a value of 8 than for the other case. Such plots end up stagnating when reaching the execution of the program with 12 threads, but one thing is clear: it is optimal to have a considerable amount of threads for whatever value one chooses for *userparam*, but for bigger values of it, one will get a smaller execution time, thus confirming that the change in the *j* dimension affects the ratio between computation and synchronization, as well as verifying our expectations in regards to the effect of having a bigger or smaller value for *j*.

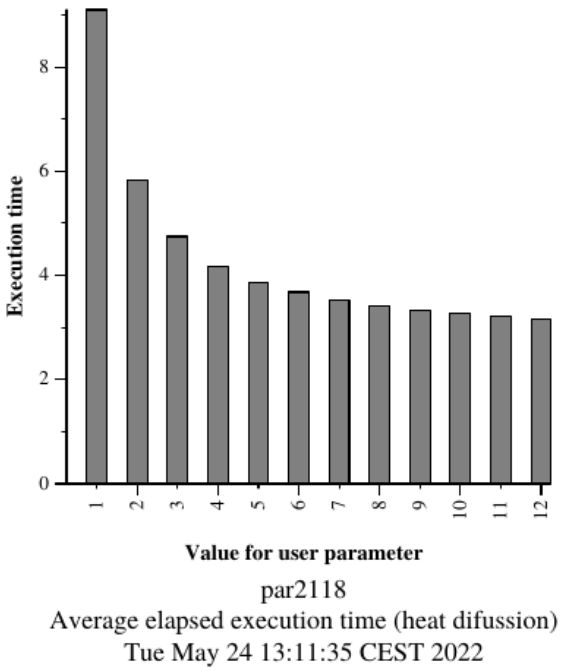

Figure 18: Execution time with 4 threads depending on the user parameter

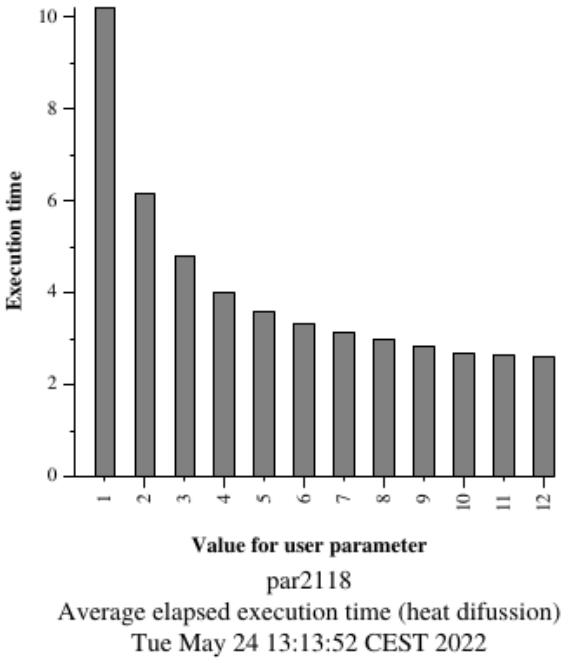

Figure 19: Execution time with 8 threads depending on the user parameter

Conclusion

First of all, in this lab we've seen (once again) how the preliminary Tareador instrumentation is necessary to comprehend the character of a given code and the phenomena that surrounds it (the dependencies and relations that arise with a given instrumentation, and how all of it comes together once one gets to parallelize the code with such instrumentation in mind). For this particular case, it's been both for the *Jacobi* and the *Gauss-Seidel* solvers, since both of them make a particular use of a variable that, in the end, could heavily slow down the execution of the program, as well as the relation between the solving functions (for the *Jacobi* method in particular).

Second of all, we've touched upon the parallelization of code with implicit tasks, which is something that we hadn't really trained seriously until now. With such cases, one had to use the dependencies detected in the Tareador instrumentation to ensure the correctness of the execution of the program itself. And not only that: parallelizing the code on the grounds of the observed phenomena in the Tareador instrumentation wouldn't be enough, for (as we've seen) various problems might arise, such as false-sharing or data-race conditions (both for accessing the *sum* variable and for the implementation of a mechanism that evades that problem itself, thus giving place to another type of problem…). Now, as our implementation tried to achieve, according to the solver used, we had all those questions already covered by the parallelization, thus achieving the efficiency presented in the sections of this document.

Lastly, the fact that we had to deal with two different solvers that had such different computing policies was certainly interesting, for we got to see (both in the *Paraver* windows and in the generated plots) that it is not intuitive to detect which solver is universally better, and that even though it might depend on the chosen instrumentation and parallelization, sometimes the methods that have the biggest workload are also the most efficient ones (once again, keep in mind that such a statement does not necessarily hold universally).