Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona

Lab 5: Geometric (data) decomposition using implicit tasks: heat diffusion equation

Izan Cordobilla Blanco (par1304)

Alejandro Salvat Navarro (par1120)

Grau en enginyeria informàtica

Quadrimestre de primavera 2021-2022

INDEX

1. INTRODUCTION 3
2. SEQUENTIAL HEAT DIFFUSION PROGRAM AND ANALYSIS WITH TAREADOR 4
3. PARALLELISATION OF THE HEAT EQUATIONS SOLVER 11
   3.1. JACOBI.. 11
   3.2. GAUSS SEIDEL 14
4. Optional 1 18
5. CONCLUSIONS... 20

1. INTRODUCTION

In this last last laboratory assignment we will work on the parallelisation of a sequential code (heat.c) that simulates the diffusion of heat in a solid body using two different solvers for the heat equation: Jacobi and Gauss-Seidel.

In the first session we will explore the parallelism using Tareador and in the second one, we will parallelise the sequential code for the heat equation code considering the use of the Jacobi and Gauss-Seidel solvers.

We think that it's important to know before starting the session that the *heat.c* code generates an image of heat diffusion when two heat sources are placed in the borders of the 2D solid (one in the upper left corner and the other in the middle of the lower border). In the following images, we can see the images generated when using Jacobi or Gauss-Seidel solvers:

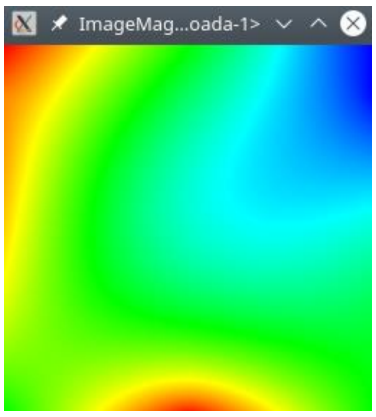


Figure 1: Image generated using Jacobi solver

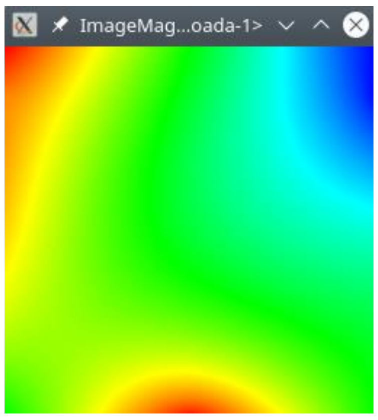


Figure 2: Image generated using Gauss solver

2. SEQUENTIAL HEAT DIFFUSION PROGRAM AND ANALYSIS WITH TAREADOR

We will use Tareador to study the parallelization by analyzing the task graphs generated when using the two different solvers. Taking a look at the heat-tareador.c code, we can see that the two solvers and the auxiliary function named "copy_mat" are identified as Tareador tasks.

We will compile this code and we will execute the binary using:

- **# ./run-tareador.sh heat-tareador 0** → Jacobi Solver
- **# ./run-tareador.sh heat-tareador 1** → Gauss-Seidel Solver

We can see the TDGs generated for both solvers in the following images:

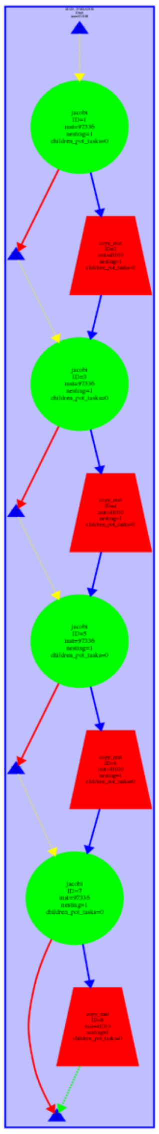


Figure 3: TDG using Jacobi solver

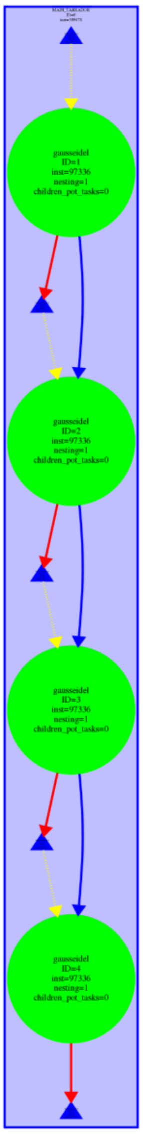
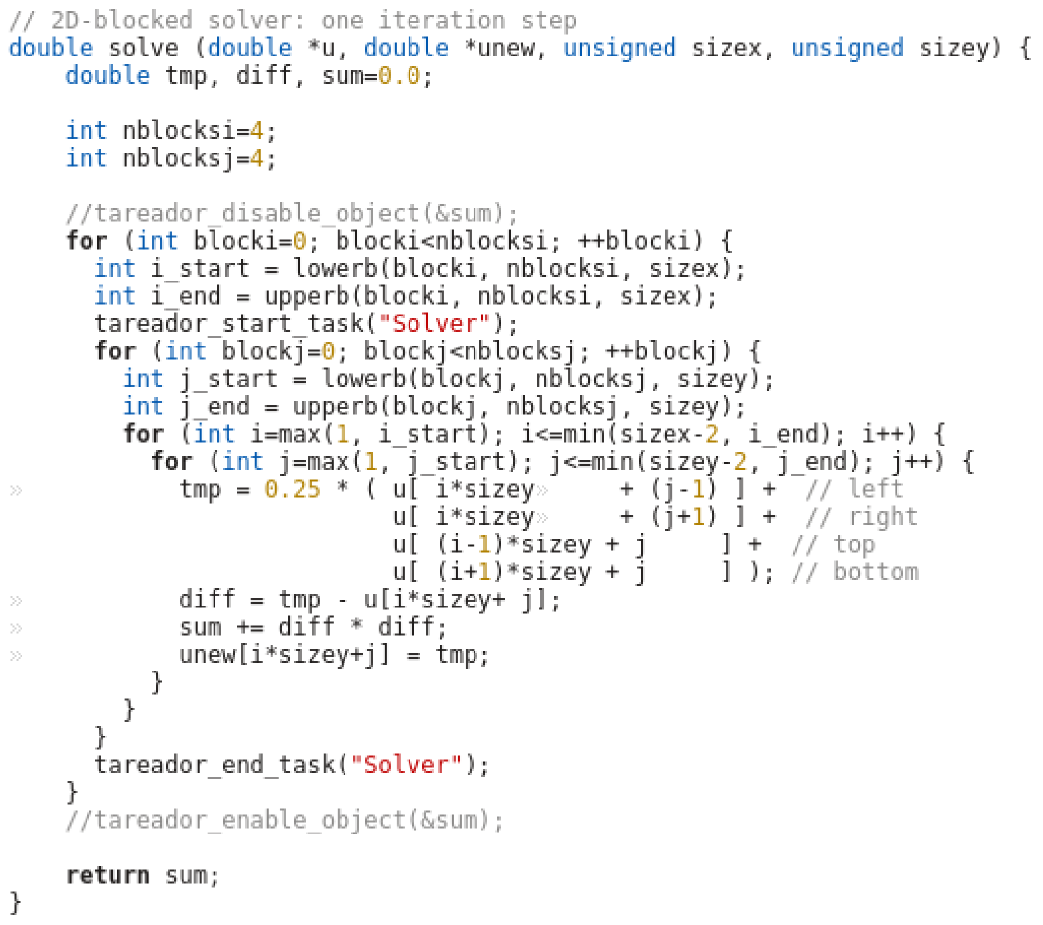


Figure 4: TDG using Gauss solver

Observing these two graphs we can see that they follow a similar structure. They have a little parallelism and unbalanced workload. Moreover, we can see that the Jacobi solver has more tasks due to the use of the copy_mat function.

After observing these two graphs we can conclude that at this level of granularity it is difficult to see any potential parallelism. Because of that, we will use the solver-tareador.c code.

We can see in the following image the changes that we have done to the code in order to explore a finer granularity:



```c
// 2D-blocked solver: one iteration step
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
    double tmp, diff, sum=0.0;

    int nblocksi=4;
    int nblocksj=4;

    //tareador_disable_object(&sum);
    for (int blocki=0; blocki<nblocksi; ++blocki) {
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      tareador_start_task("Solver");
      for (int blockj=0; blockj<nblocksj; ++blockj) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++) {
          for (int j=max(1, j_start); j<=min(sizey-2, j_end); j++) {
            tmp = 0.25 * ( u[ i*sizey     + (j-1) ] +  // left
                           u[ i*sizey     + (j+1) ] +  // right
                           u[ (i-1)*sizey + j     ] +  // top
                           u[ (i+1)*sizey + j     ] ); // bottom
            diff = tmp - u[i*sizey+ j];
            sum += diff * diff;
            unew[i*sizey+j] = tmp;
          }
        }
      }
      tareador_end_task("Solver");
    }
    //tareador_enable_object(&sum);

    return sum;
}
```

Figure 5: First changes to solver-tareador.c

Executing the program using both solvers generates the following results:

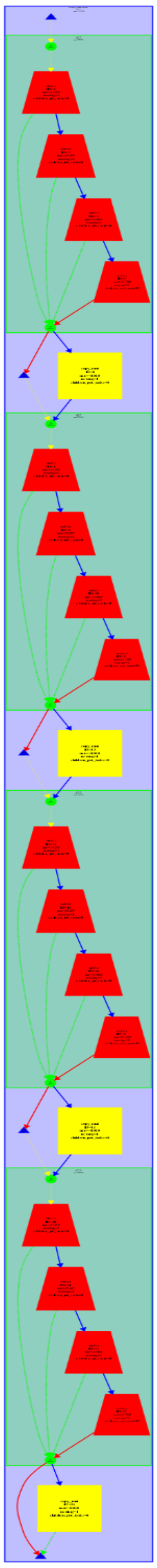


Figure 6: TDG using Jacobi solver

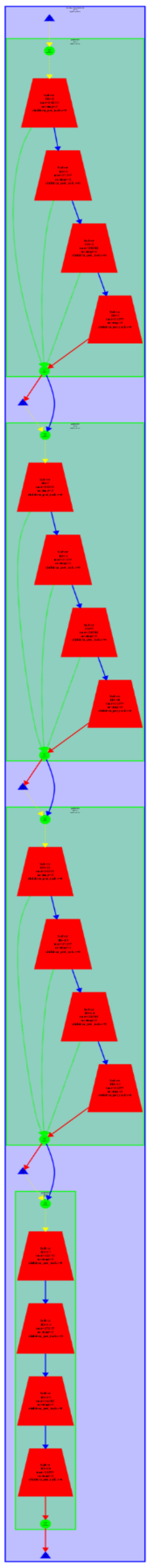


Figure 7: TDG using Gauss solver

As we can see, there are more tasks to execute but there is one problem that is causing dependencies. Using the Data View option in one edge of the TDGs, we can observe that the variable **sum** is causing the serialization of the tasks.

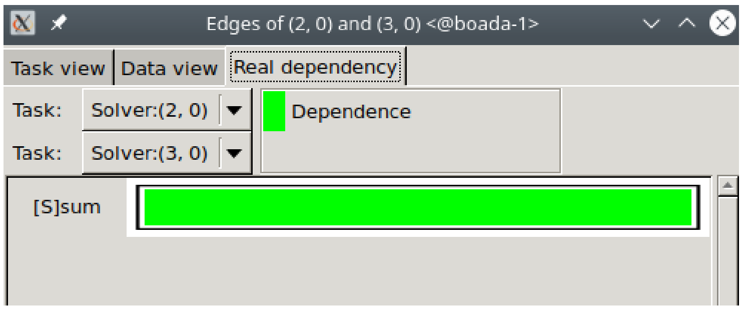


Figure 8: Variable that creates dependencies

Now that we have found the problem, it's time to fix it. In order to do that, we have done the following changes to the code:

```
// 2D-blocked solver: one iteration step
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
    double tmp, diff, sum=0.0;

    int nblocksi=4;
    int nblocksj=4;

    tareador_disable_object(&sum);
    for (int blocki=0; blocki<nblocksi; ++blocki) {
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      tareador_start_task("Solver");
      for (int blockj=0; blockj<nblocksj; ++blockj) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++) {
          for (int j=max(1, j_start); j<=min(sizey-2, j_end); j++) {
            tmp = 0.25 * ( u[ i*sizey     + (j-1) ] +  // left
                           u[ i*sizey     + (j+1) ] +  // right
                           u[ (i-1)*sizey + j     ] +  // top
                           u[ (i+1)*sizey + j     ] ); // bottom
            diff = tmp - u[i*sizey+ j];
            sum += diff * diff;
            unew[i*sizey+j] = tmp;
          }
        }
      }
      tareador_end_task("Solver");
    }
    tareador_enable_object(&sum);

    return sum;
}
```

Figure 9: Second changes to solver-tareador.c

After executing the previous code with both solvers, we have obtained the following TDGs:

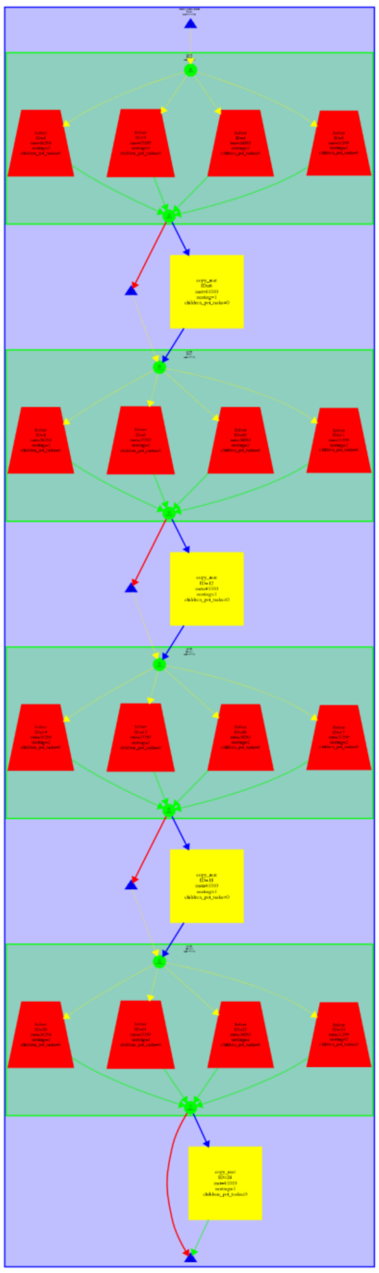


Figure 10: TDG using Jacobi solver

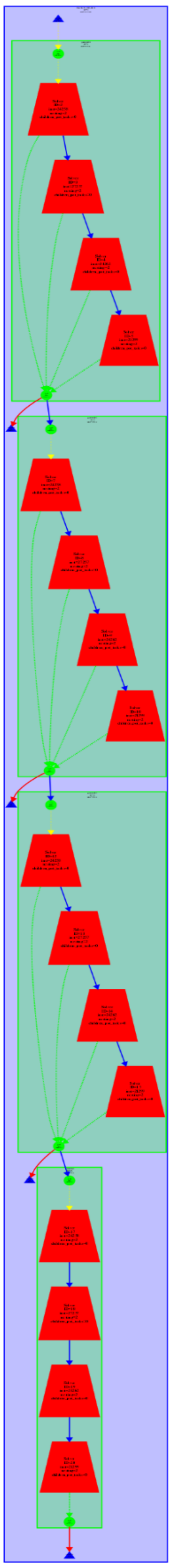


Figure 11: TDG using Gauss solver

Simulating the execution using 4 processors, we obtained the following results:

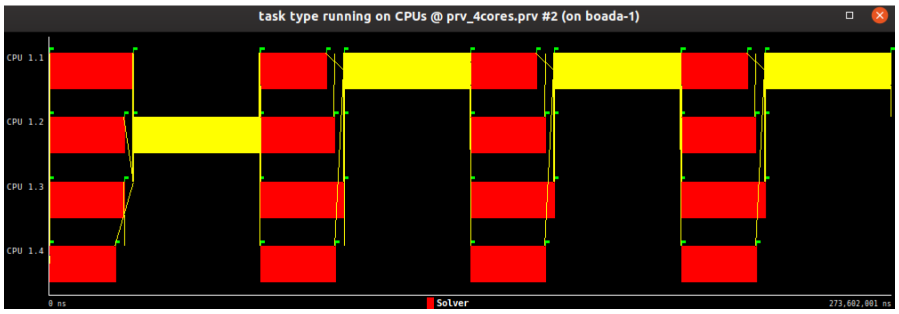


Figure 12: Jacobi timeline using 4 threads

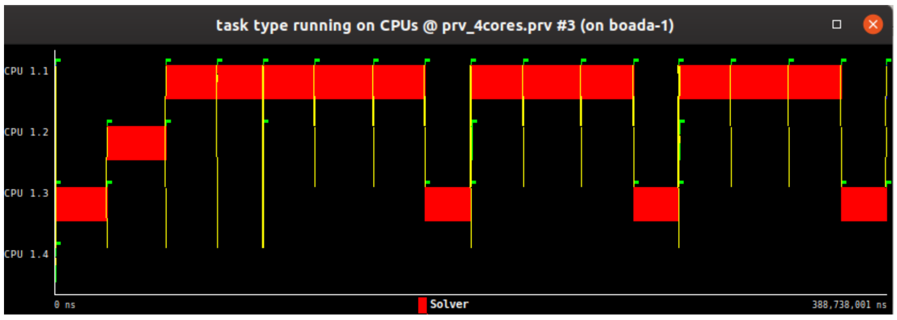
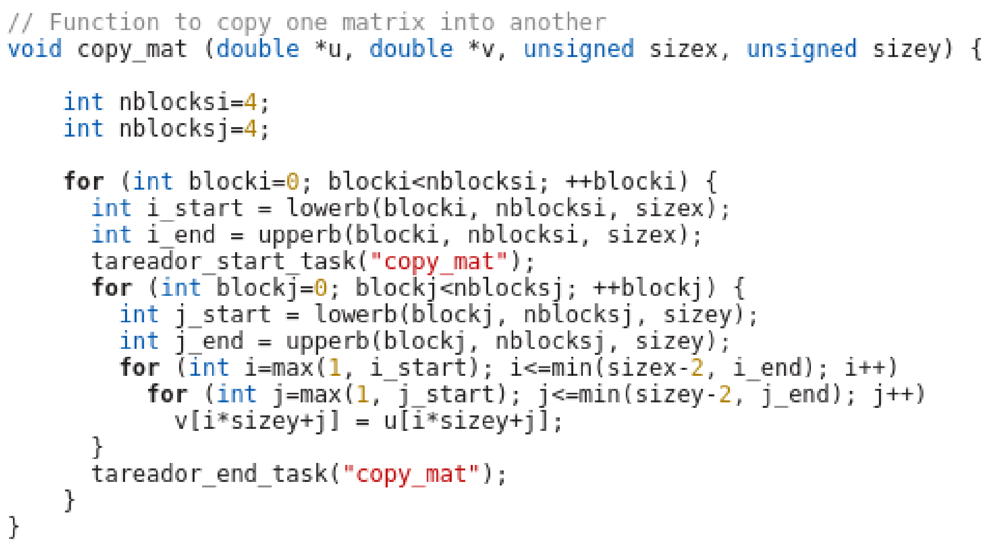


Figure 13: Gauss timeline using 4 threads

In them we can see how the Gauss-Seidel solver is executing sequentially and the Jacobi solver has a part that is parallelizable: the yellow one (auxiliary function).

As we can see in the previous TDGs and simulations, there is an improvement in the parallelism. Although this is good, we have observed that in the Jacobi TDG the tasks of the auxiliary function have a lot of workload and because of that, we will include Tareador tasks inside the function (as you can see in the following image).



```
// Function to copy one matrix into another
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {

    int nblocksi=4;
    int nblocksj=4;

    for (int blocki=0; blocki<nblocksi; ++blocki) {
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      tareador_start_task("copy_mat");
      for (int blockj=0; blockj<nblocksj; ++blockj) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++)
          for (int j=max(1, j_start); j<=min(sizey-2, j_end); j++)
            v[i*sizey+j] = u[i*sizey+j];
      }
      tareador_end_task("copy_mat");
    }
}
```

Figure 14: Changes made to the copy_mat function

Executing the previous code, we have obtained the following TDG:

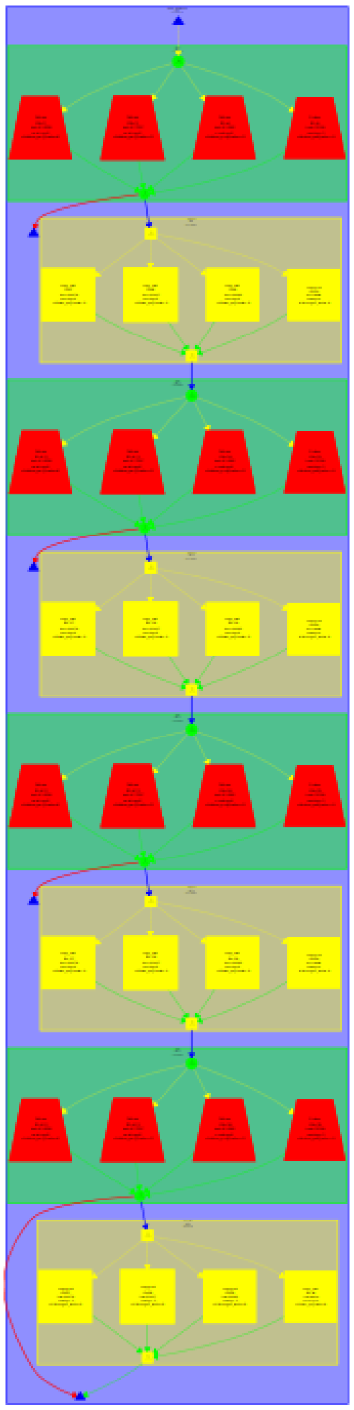


Figure 15: Jacobi TDG after changes in the code

After the parallelization of the copy_mat function, the newests resultas are the best ones yet. We can see that the time spent inside the auxiliary function is reduced because it is divided between 4 threads.

We only have put the TDG of the Jacobi solver because it is the one that uses this function (Gauss-Seidel don’t).

3. PARALLELISATION OF THE HEAT EQUATIONS SOLVER

3.1. JACOBI

In this section we will first parallelise the sequential code for the heat equation code considering the use of the Jacobi solver, using the implicit tasks generated in #pragma omp parallel, following a geometric block data decomposition by rows for 4 threads running on 4 processors.

With this strategy the computation that the implicit task executed each thread has to perform is determined by the data it has to access, either read or write. This may have clear benefits in terms of data locality exploitation because each thread will always execute implicit tasks that access the same data, whenever possible.

First of all, we will review the parallelization proposed in the solver-omp.c code. Once understood, we will complete the parallelization so that it fulfills the dependencies previously discovered by applying the Jacobi solver. For it, we will begin protecting the variables diff and tmp with a private clause, avoiding accesses to other threads that can give problems in the execution of the program. Then, to avoid the serialization caused by the variable sum, we will protect the variable with a reduction, making sure that each thread has a copy of it.

Once verified that the result is correct, we pass to execute the script submit-strong-omp.sh to observe the graph of strong scalability.

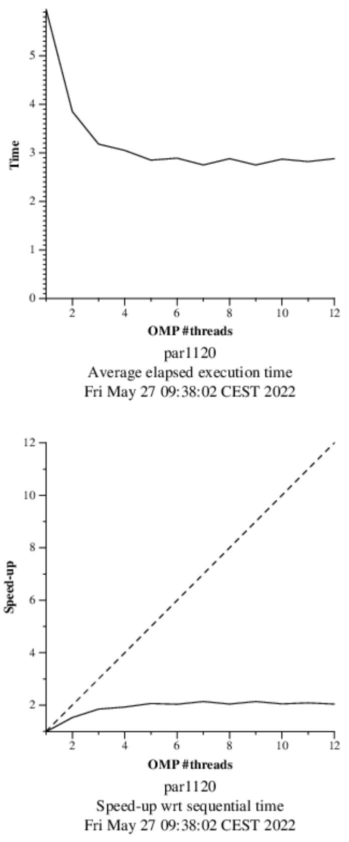


Figure 16: Scalability plots using Jacobi solver

As we can see, the speedup is minimal, so we will look at the program timeline to see if there is any way to improve it.

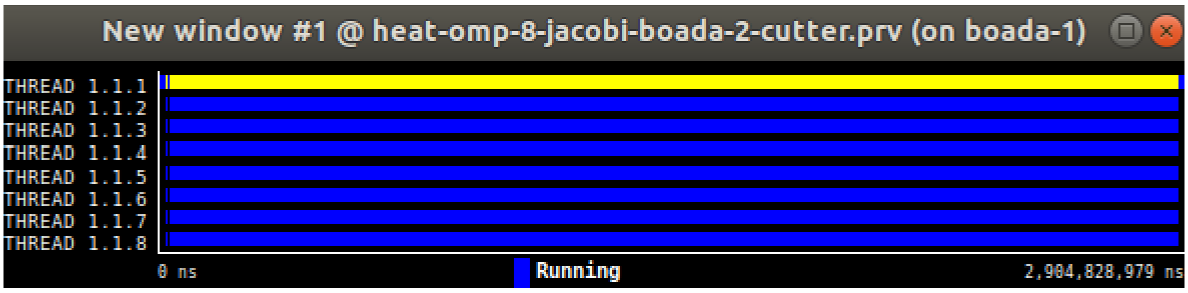


Figure 17: Timeline with 8 threads using Jacobi solver

We can observe that thread1 occupies practically all of its time in the creation of tasks, in addition to executing the other parts of the program that are not the solve function alone. To improve the performance of the program, we will parallelize these other functions as well.

We parallelize the copy_mat function, dividing its execution in different threads. Once done, we run the submit-strong-omp.sh script to observe the improvement in the strong scalability graph.

In the graphs we can see a great improvement in both the program execution time and scalability, confirming our hypothesis that the copy-mat function was a drag on program performance if it was not parallelized.

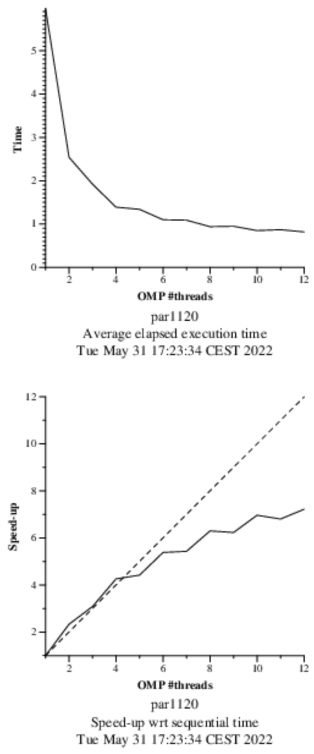


Figure 18: Scalability plots using Jacobi solver with improvements

We will now look at the paraver timelines, in order to observe the changes in execution.

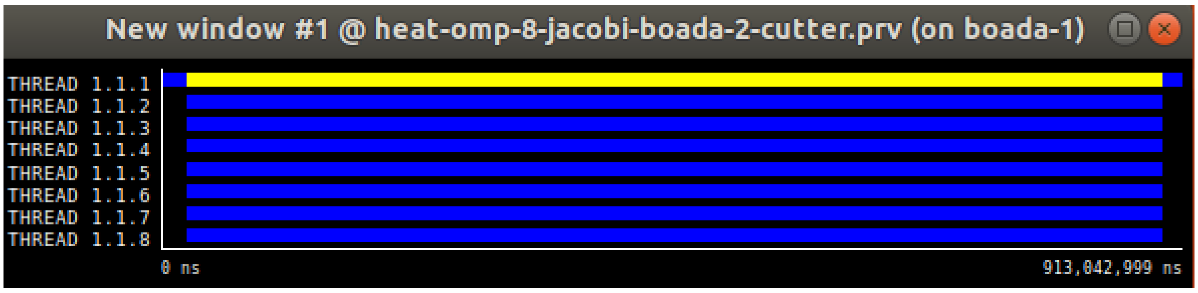


Figure 19: Timeline with 8 threads using Jacobi solver

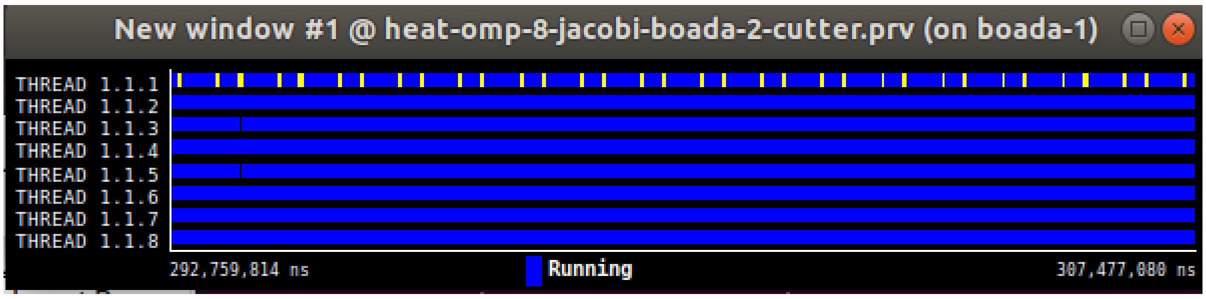


Figure 20: Timeline with 8 threads using Jacobi solver

As we can see, our first thread now interleaves program execution with task creation, which increases its performance.

3.2. GAUSS SEIDEL

We will continue the parallelization process considering the dependences that appear when the Gauss-Seidel is used. The parallelisation will follow a geometric block by row data decomposition and we will only use implicit tasks in parallel regions. Moreover, we will take into account the annex of the lab assessment, which helps to solve memory consistency problems. This is the resulting code:

```
// 2D-blocked solver: one iteration step
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
    double tmp, diff, sum=0.0;
    int nblocksi=omp_get_max_threads();
    int nblocksj=1;
    int mat[24] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

    #pragma omp parallel private(diff) reduction(+:sum) // complete data sharing constructs here
    {
      int cont;
      int blocki = omp_get_thread_num();
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      for (int blockj=0; blockj<nblocksj; ++blockj) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);

        if ((u == unew) && blocki != 0) {
            do {
                #pragma omp atomic read
                cont = mat[blocki-1];
            } while (cont <= blockj);
        }

        for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++) {
          for (int j=max(1, j_start); j<=min(sizey-2, j_end); j++) {
»           tmp = 0.25 * ( u[ i*sizey»     + (j-1) ] +  // left
                           u[ i*sizey»     + (j+1) ] +  // right
                           u[ (i-1)*sizey + j     ] +  // top
                           u[ (i+1)*sizey + j     ] ); // bottom
»           diff = tmp - u[i*sizey+ j];
»           sum += diff * diff;
»           unew[i*sizey+j] = tmp;
          }
        }

        if (u == unew) {
            #pragma omp atomic write
            mat[blocki] = mat[blocki] + 1;
        }
      }
    }

    return sum;
}
```

Figure 21: Implementation of Gauss solver using implicit tasks

In order to see if the previous code is generating a good result we have executed “sbatch ./submit-omp.sh heat-omp 1 4” to see if the image generated is the same as the image generated with the sequential code. As we can see in the following images, they are the same and that indicate us that the parallelisation has been done correctly:

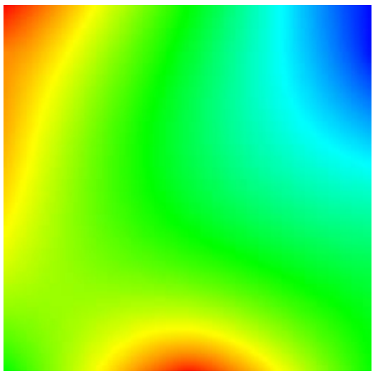


Figure 22: Image obtained with sequential Gauss solver

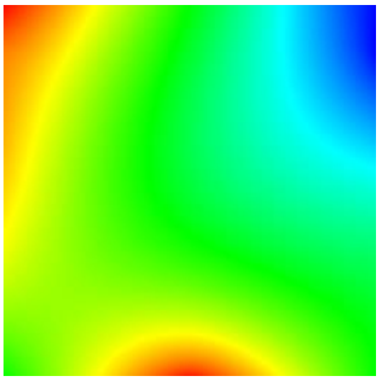
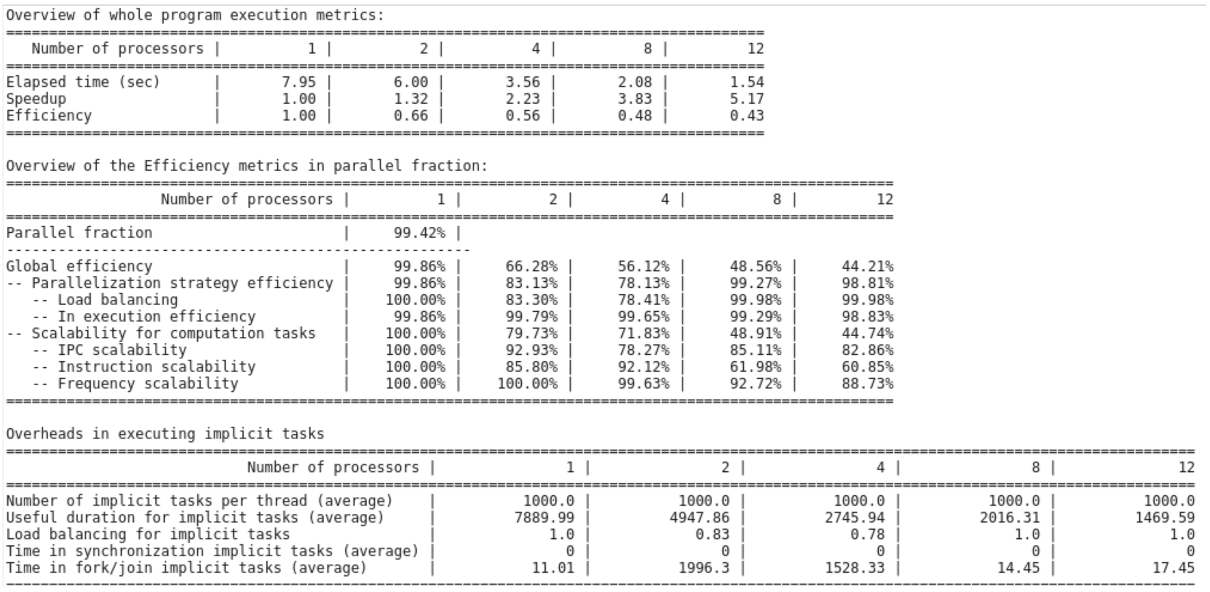


Figure 23: Image obtained with the new Gauss solver

In addition, we have submitted the execution using the ./submit-strong-extrae.sh in order to obtain the modelfactor’s table and some timelines.
In the following table, we can see the modelfactors.out:

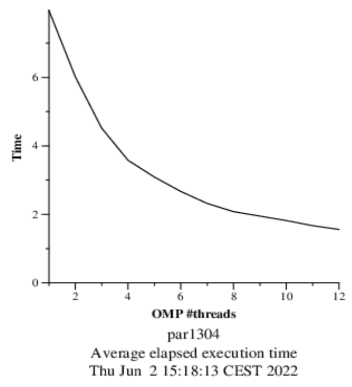


```
Overview of whole program execution metrics:
===============================================================================
   Number of processors |          1 |          2 |          4 |          8 |         12
===============================================================================
Elapsed time (sec)      |       7.95 |       6.00 |       3.56 |       2.08 |       1.54
Speedup                 |       1.00 |       1.32 |       2.23 |       3.83 |       5.17
Efficiency              |       1.00 |       0.66 |       0.56 |       0.48 |       0.43
===============================================================================

Overview of the Efficiency metrics in parallel fraction:
==================================================================================================
                  Number of processors |          1 |          2 |          4 |          8 |         12
==================================================================================================
Parallel fraction                      |     99.42% |
-----------------------------------------------------------
Global efficiency                      |     99.86% |     66.28% |     56.12% |     48.56% |     44.21%
-- Parallelization strategy efficiency |     99.86% |     83.13% |     78.13% |     99.27% |     98.81%
   -- Load balancing                   |    100.00% |     83.30% |     78.41% |     99.98% |     99.98%
   -- In execution efficiency          |     99.86% |     99.79% |     99.65% |     99.29% |     98.83%
-- Scalability for computation tasks   |    100.00% |     79.73% |     71.83% |     48.91% |     44.74%
   -- IPC scalability                  |    100.00% |     92.93% |     78.27% |     85.11% |     82.86%
   -- Instruction scalability          |    100.00% |     85.80% |     92.12% |     61.98% |     60.85%
   -- Frequency scalability            |    100.00% |    100.00% |     99.63% |     92.72% |     88.73%
==================================================================================================

Overheads in executing implicit tasks
========================================================================================================================================
                             Number of processors |                  1 |                  2 |                  4 |                  8 |                 12
========================================================================================================================================
Number of implicit tasks per thread (average)     |             1000.0 |             1000.0 |             1000.0 |             1000.0 |             1000.0
Useful duration for implicit tasks (average)      |            7889.99 |            4947.86 |            2745.94 |            2016.31 |            1469.59
Load balancing for implicit tasks                 |                1.0 |               0.83 |               0.78 |                1.0 |                1.0
Time in synchronization implicit tasks (average)  |                  0 |                  0 |                  0 |                  0 |                  0
Time in fork/join implicit tasks (average)        |              11.01 |             1996.3 |            1528.33 |              14.45 |              17.45
----------------------------------------------------------------------------------------------------------------------------------------
```

Table 1: modelfactors.out of the Gauss solver

In it, we can see that the main problem is the scalability for computation tasks. This reduces the global efficiency of the execution and leads us to a worse performance.

We can see this problem in the following plots:



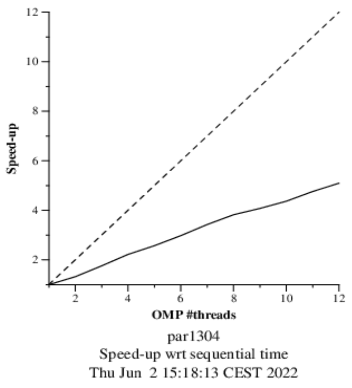


Figure 24: Scalability plots using gauss solver

The results obtained are worse than the ones obtained using Jacobi. We can see that the execution time decreases with the increment of threads but, with every increase of threads it reduces less.

Moreover, comparing the speedup obtained with the "ideal" speedup we can see how slow and bad it increments.

In order to understand what is happening, we will take a look at the generated timeline using 8 threads

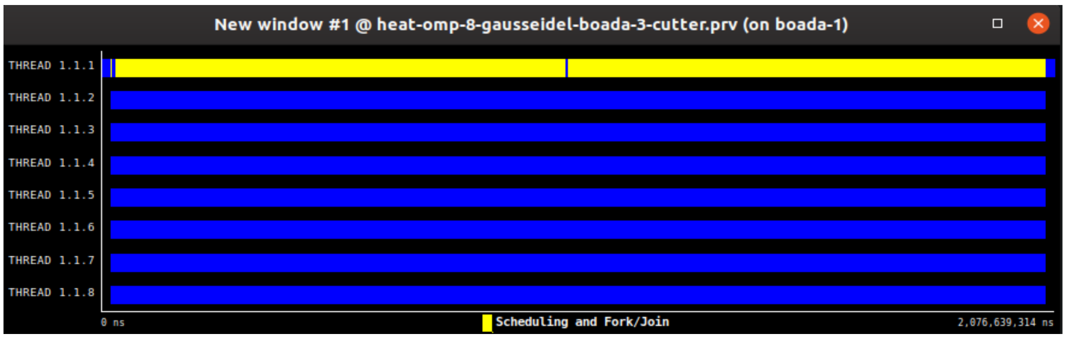


Figure 25: Timeline with 8 threads using Gauss solver

As we can observe, the first thread creates all the tasks and the other ones are executing them.

In order to exploit more parallelism we have changed the number of j blocks and we decided to do this instead:

```
// 2D-blocked solver: one iteration step
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
    double tmp, diff, sum=0.0;

    int nblocksi=omp_get_max_threads();
    int nblocksj=userparam;
    int mat[24] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
```

Figure 26: Second changes to the Gauss solver

Executing the program using the **./submit-userparam-omp.sh** script we have obtained the following plots:

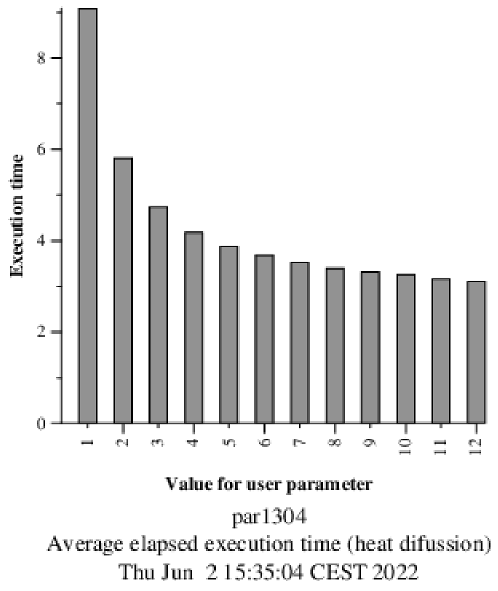


Figure 27: Execution time while using 4 threads

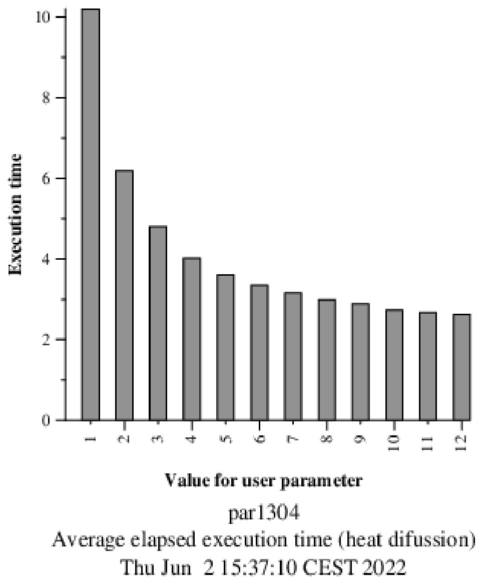


Figure 28: Execution time while using 8 threads

If we compare and analyze both plots, we can observe that the value for **userparam** that decreases more the execution time is 12. In addition, when we are using more threads, the execution time also increases. We can conclude that we have obtained better results because if we have more *blocksj*, we have more loops and the program can be parallelized even more.

4. Optional 1

In this optional section we will implement the Jacobi solver using explicit tasks and following an iterative task decomposition and compare the results with those obtained previously.

In the code we will change the previously used clauses to #pragma omp parallel and #pragma omp single, in addition to using taskloop in the outermost for loops (which will cause the explicit tasks of our program).

Once the code is modified, we run the submit-strong-omp.sh script to observe the strong scalability graph.

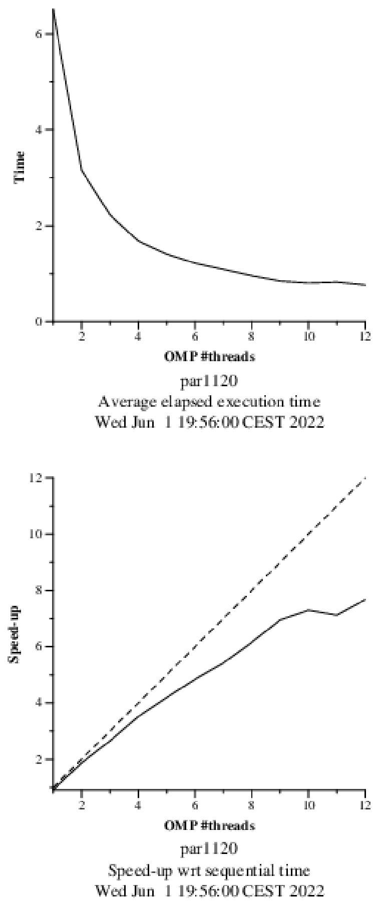


Figure 29: Scalability plot of heat-omp.c

As we can see in the graph, the results obtained are practically the same as those obtained with the configuration using implicit tasks. The difference is noticeable in the smoothness of both graphs, since the previously seen graphs presented very sharp scalability growths.

Once the strong scalability has been analyzed, we will proceed to analyze the paraver timeline.

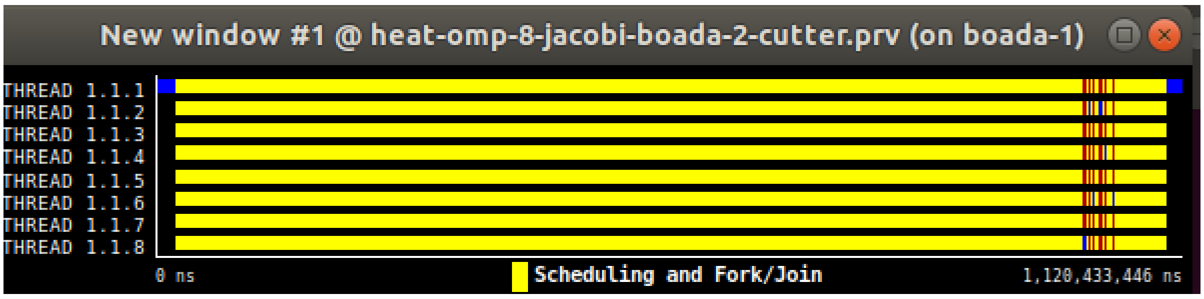


Figure 30: Paraver with expicit taks and without copy_mat

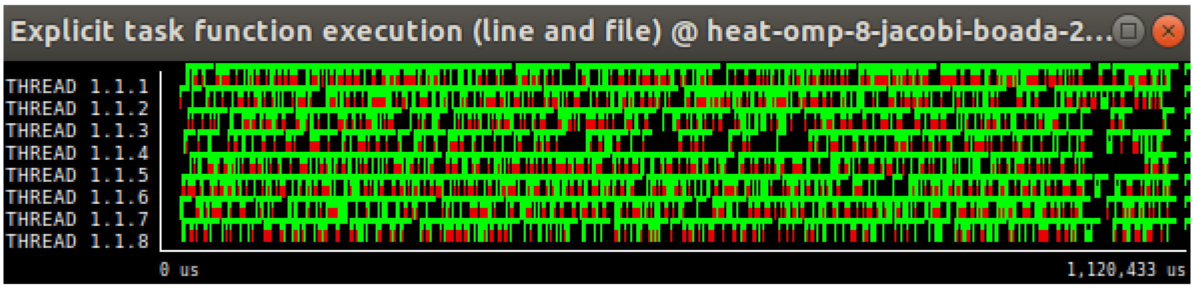


Figure 31: Paraver with expicit taks and with copy_mat

As we can see, now all threads spend time creating tasks and the execution of the program takes a smaller percentage of time. In addition, we see that the explicit tasks cover the whole area influenced by our solver script.

5. CONCLUSIONS

In this laboratory session we have seen the simulation of heat diffusion in a solid body using 2 equations: Jacobi and Gauss-Seidel.

As we have been able to observe in the results obtained, the simulation using the Jacobi equation presents a much higher potential parallelization than the simulation with Gauss-Seidel.

On the other hand, we have seen that by changing the implicit tasks by other explicit ones, in some cases we can smooth the strong scalability graph, making it smoother and avoiding an escalation with steep slopes.

Finally, we have found that no matter how many threads are used to parallelize a process, in case the parallelizable part is small, the scalability of our program will be very inappropriate.