ENTREGABLE PARAL·LELISME

CINQUÈ ENTREGABLE

Username: par4105
28 de Desembre de 2018
Tardor del 2018

Àlex Valls
Roger Guasch

Table of content

Introduction

In this session we will apply data decomposition strategies in order to improve the performance of two different algorithms that simulate heat diffusion in a solid body. The first algorithm uses the Jacobi method, an iterative approach for determining the solutions of a system of linear equations, (https://en.wikipedia.org/wiki/Jacobi_method) whereas the second algorithm adopts the Gauss-Seidel method. Both methods are similar, although the Gauss-Seidel can be applied to any matrix with non-zero elements on the diagonals. (https://en.wikipedia.org/wiki/Gauss%E2%80%93Seidel_method)

First of all an analysis with *Tareador* software will be performed, describing the characteristics of each implementation, followed by a data decomposition implementation for both versions.

Sequential heat diffusion program

We will begin by displaying the results of Jacobi and Gauss-Seidel algorithms. By doing so we have complied the appropriate programs using the *make* directive and we have analyzed the differences obtained. First of all we are going to look the results of the Jacobi approach, as they can be seen in Figure 5.1.

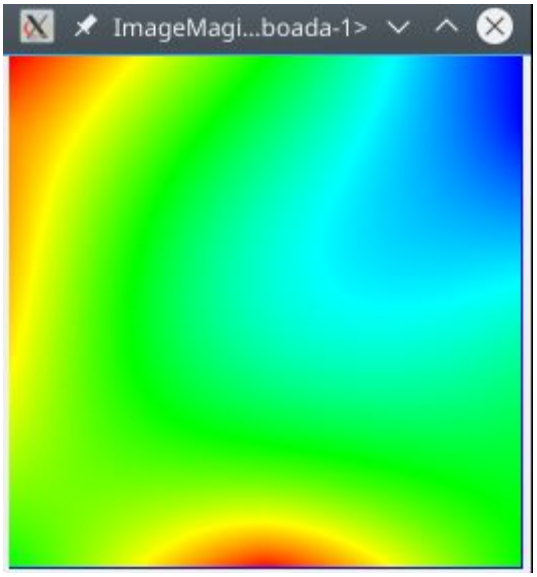

Figure 5.1 Showing the display obtained after executing Jacobi's method to represent the heat in each point of a 2D solid

Afterward we applied the same procedure but this time changing the value in *test.dat* archive, in order to display Gauss-Seidel solution. The display we have obtained can be seen in Figure 5.2.

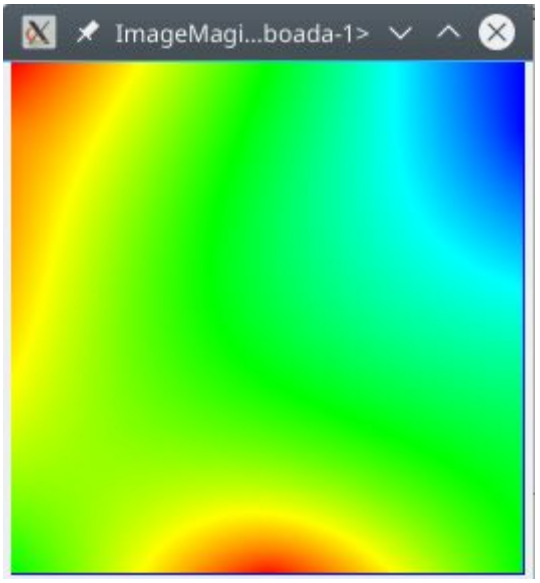

Figure 5.2 Showing the display obtained after executing Gauss-Seidel method to represent the heat in each point of a 2D solid

Differences between both solvers are hard to find but, at the same time, it is clear by looking at the top right of both images that are not strictly the same. In order to exemplify those differences we executed the diff command between to images, getting the output shown in Figure 5.3.

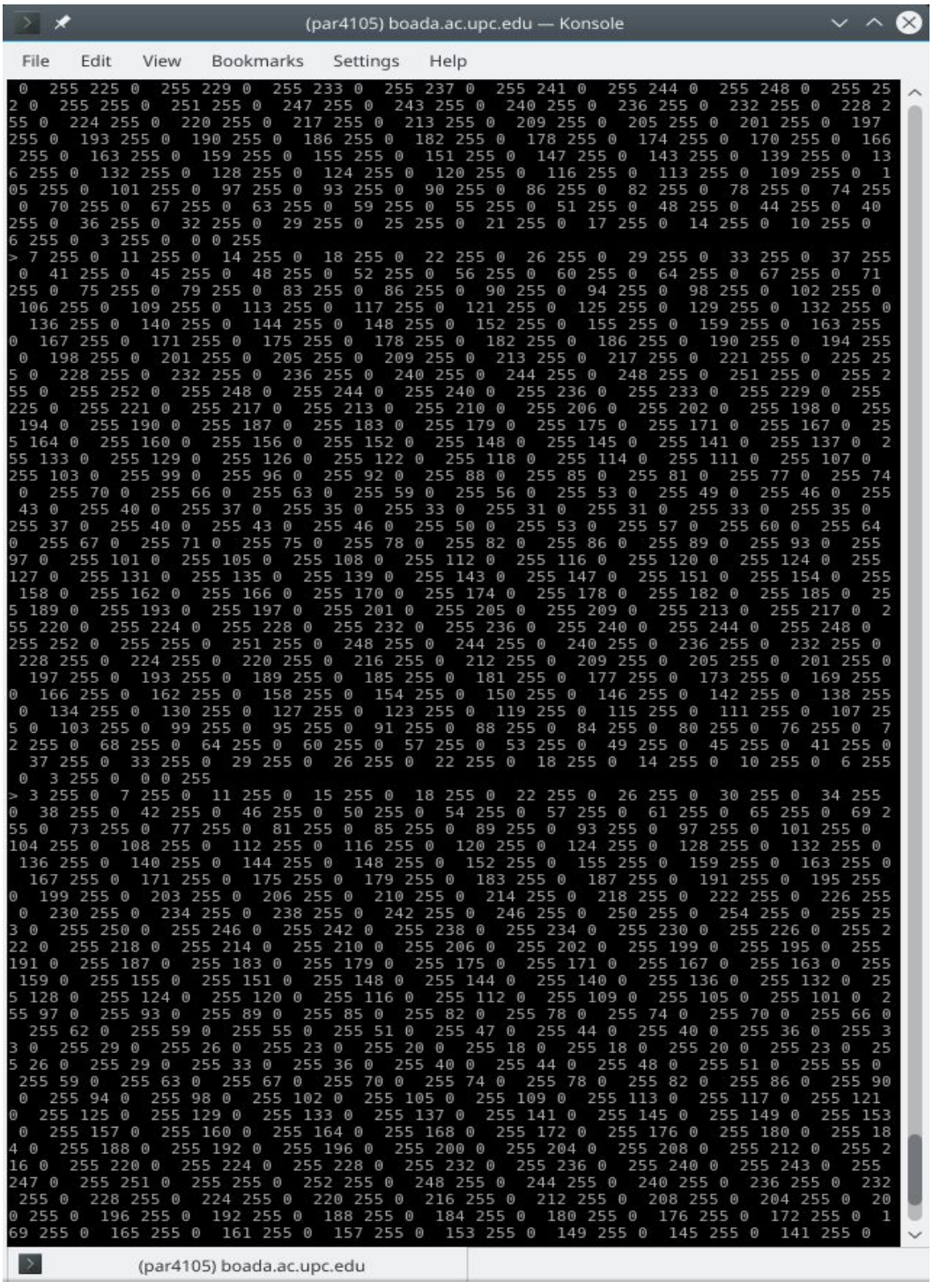

Figure 5.3 Differences between Jacobi and Gauss-Seidel algorithms.

Tareador analysis

In this section of the report we are going to share the results and conclusions we have obtained analysing with *Tareador* software both solvers. Both algorithms analyze a *4x4* matrix, in order to process the results quickly in *Tareador*. The API calls to Tareador functions can be seen in Figure 5.4, whereas the graph obtained is shown in Figure 5.5.

```
tareador_start_task("relax_jacobi");
residual = relax_jacobi(param.u, param.uhelp, np, np);
tareador_end_task("relax_jacobi");
// Copy uhelp into u
//tareador_start_task("Copy_mat");
copy_mat(param.uhelp, param.u, np, np);
//tareador_end_task("Copy_mat");
```

Figure 5.4 Tareador API calls in order to obtain the graph desired.

As it can be appreciated, between each *relax_jacobi* tasks, a copy of the matrix has to be done. We need to take into a consideration that performing a copy of the matrix has its impact over the average performance of the program.

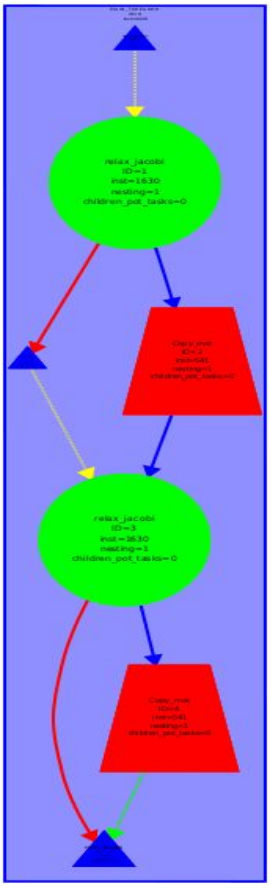

Figure 5.4 Graph obtained, showing the execution of tasks of Jacobi's algorithm.

To obtain the second graph we followed the same procedure, but changing the previous *Tareador* API calls to the ones shown in Figure 5.5.

```
tareador_start_task("relax_gauss");
residual = relax_gauss(param.u, np, np);
tareador_end_task("relax_gauss");
```

Figure 5.5 Tareador API calls in order to obtain the graph desired.

The main difference with Jacobi's solver is that, Gauss-Seidel algorithm, does not need to execute a copy of the matrix, achieving better performance.

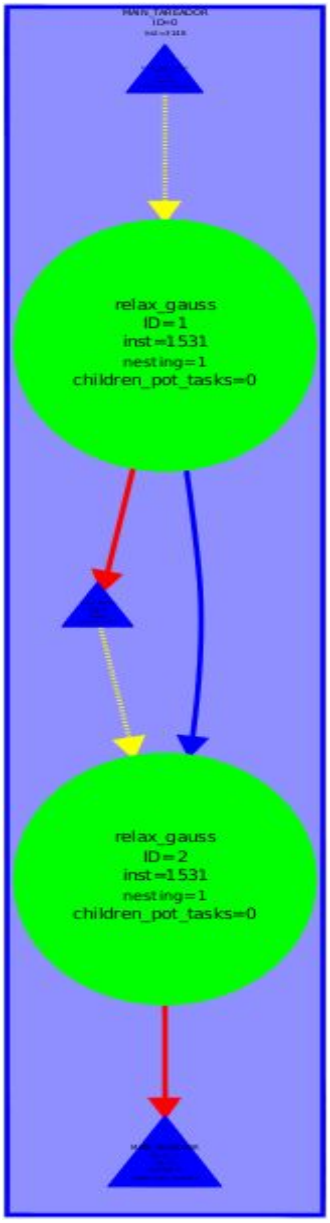

Figure 5.6 Graph obtained, showing the execution of tasks of Gauss-Seilder algorithm.

After this study, we have explore the dependences that appear when we define one task for each iteration of the body of the innermost loop. There is one dependence with *sum* and *diff* variable. *diff* dependence is easy to solve: we only need to declare the variable locally, inside each iteration. To avoid showing the dependency originated by *sum*, we simply deactivate it with *tareador_disable_object* call. The code in Figure 5.7 shows the implementation of the *relax_jacobi* function applying a point decomposition strategy.

```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey) {
    double sum=0.0;
    int howmany=1;
    for (int blockid = 0; blockid < howmany; ++blockid) {
      int i_start = lowerb(blockid, howmany, sizex);
      int i_end = upperb(blockid, howmany, sizex);
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
            tareador_start_task("Chill_jacobi");
                utmp[i*sizey+j]= 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
                double diff = utmp[i*sizey+j] - u[i*sizey + j];
                tareador_disable_object(&sum);
                sum += diff * diff;
                tareador_enable_object(&sum);
                tareador_end_task("Chill_jacobi");
            }
      }
    }
    return sum;
}
```

Figure 5.7 Point decomposition strategy for relax_jacobi function.

The results obtained with Tareador can be seen below.

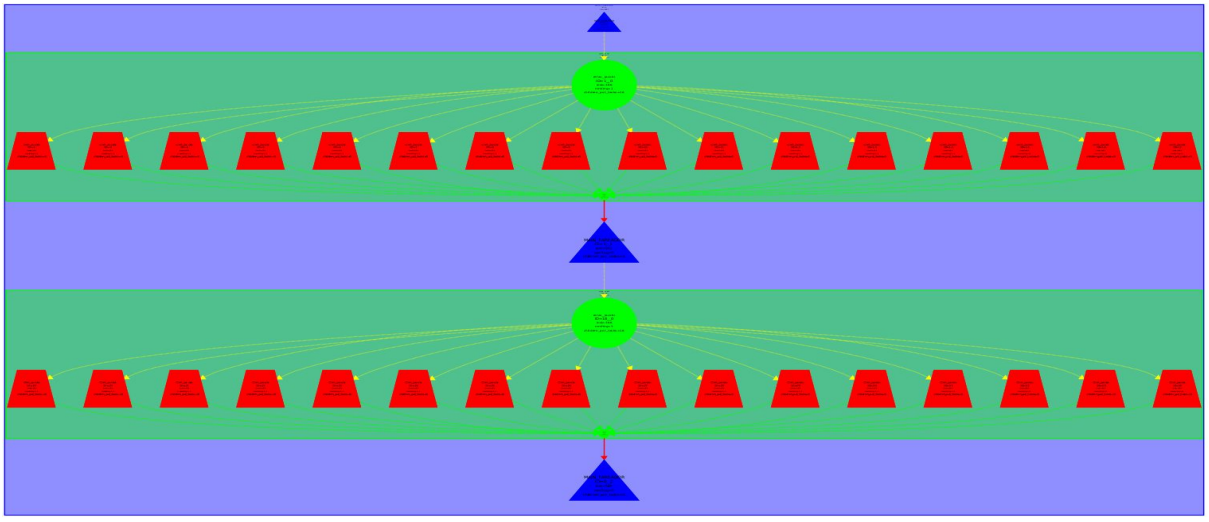

Figure 5.8 Point decomposition strategy for relax_jacobi function.

It is clear that Jacobi's algorithm is easily parallelizable, always respecting the dependencies we have found above, due to the independence of each innermost generated task. As we are going to see some figures above, this can not be said to Gauss-Seidel algorithm.
We have followed the same approach we took before: We have identified the *sum, diff* and the *unew* dependency and we have solved the by declaring locally *diff* and *unew* and disabling *sum* dependency, using the *tareador_disable_object* call. The code implemented for this analysis can be seen in Figure 5.9.

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
   double sum=0.0;

   int howmany=1;
   for (int blockid = 0; blockid < howmany; ++blockid) {
     int i_start = lowerb(blockid, howmany, sizex);
     int i_end = upperb(blockid, howmany, sizex);
     for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
       for (int j=1; j<= sizey-2; j++) {
            tareador_start_task("Chill_Gauss");
               double unew= 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
               double diff = unew - u[i*sizey+ j];
               tareador_disable_object("sum");
               sum += diff * diff;
               tareador_enable_object("sum");
               u[i*sizey+j]=unew;
               tareador_end_task("Chill_Gauss");
       }
     }
   }

   return sum;
}
```

Figure 5.9 Point decomposition strategy for relax_gauss function.

As it was expected, the results we obtained differed from the ones obtained in Jacobi's analysis. Gauss-Seidel reuses the same matrix in order to avoid the copy. This allows the algorithm to perform a little bit better sequentially but causes some dependencies between tasks while running our tests. To implement a good data decompositions we should consider those dependencies in order to avoid bad parallelization results.
The graph we have obtained with the code above can be seen in Figure 5.10.

Finally we would like to comment that, in order to respect dependencies of the code, during the parallel execution, we would parallelize the code by block of rows, each of them assigned to a different thread.

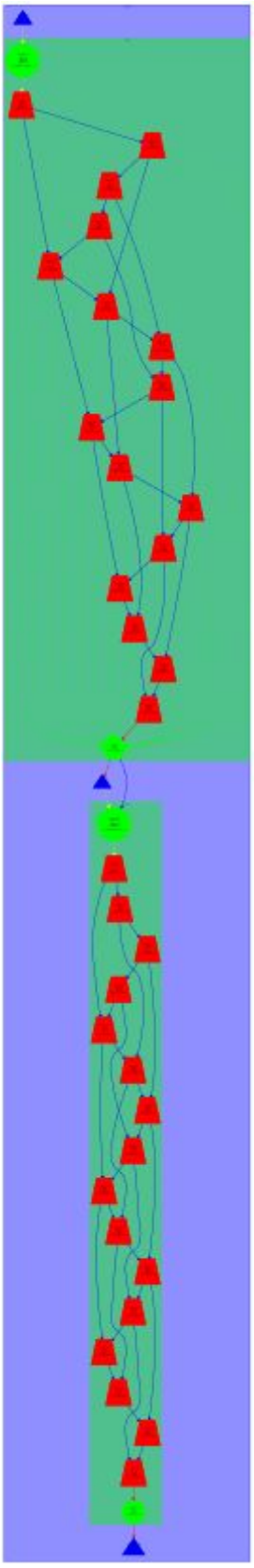

Figure 5.10 Point decomposition strategy for relax_gauss function.

Parallelization of Jacobi with OpenMP parallel

After understanding Jacobi's algorithm we proceed by parallelizing the code applying data decomposition strategies. We decided to apply the *#pragma omp parallel* directive, as it was the data decomposition strategy we have worked the most in class.
The first implementation was the one that can be readed in Figure 5.11. The part of the code in bold we are going to discuss it later on in this report.

```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
   double diff, sum=0.0;
   int howmany=4;
   #pragma omp parallel private(diff) num_threads(howmany) {
          double aux = 0;
          int blockid = omp_get_thread_num();
          int i_start = lowerb(blockid, howmany, sizex);
          int i_end = upperb(blockid, howmany, sizex);
          for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
                 for (int j=1; j<= sizey-2; j++) {
                                utmp[i*sizey+j]= 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 = utmp[i*sizey+j] - u[i*sizey + j];
                                aux += diff * diff;
                         }
                 }
          #pragma omp atomic
          sum += aux;
   }
   return sum;
}
```

Figure 5.11 First version of Jacobi's data decomposition.

Because we can calculate, inside each parallel region, which block are we working the outermost for loop is not needed in this implementation. Also we need to consider that *i_start* and *i_end* gives us the range of rows that are contained inside a block. It is a must to use *#pragma omp atomic* directive to protect the addition of the *sum* and *aux* variables. Note that, by doing so, we are avoiding using this directive once for every iteration, what could lead to a bad execution performance.

We submitted the code, with the appropriate scripts, to the *Boada's* execution queues, in order to obtain the *Paraver* screenshots needed for this report. The results obtained can be seen in Figure 5.12.

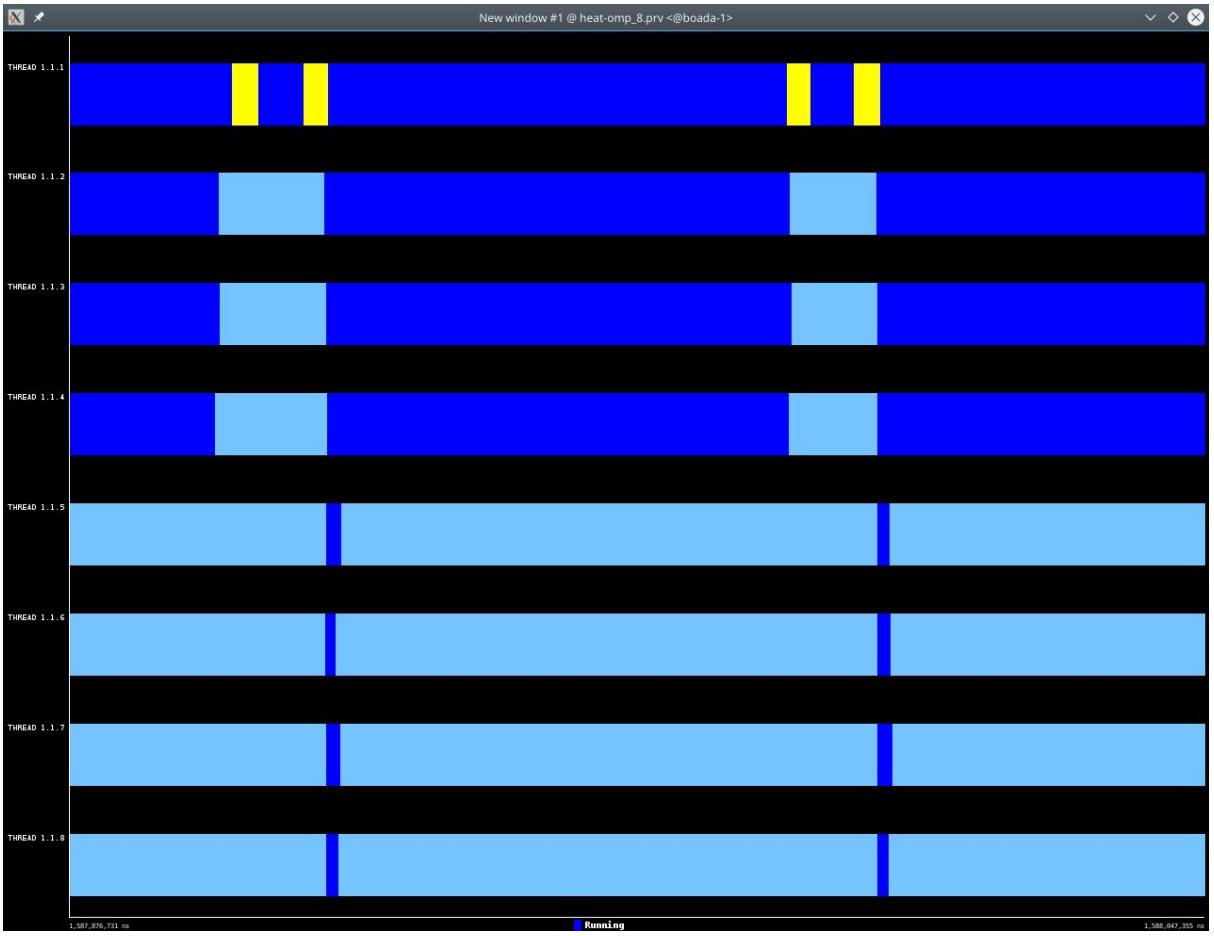

Figure 5.12 Zoom-in paraver results after submitting Jacobi's data decomposition above mentioned.

As we can analyze, although we have eight threads, only four of them are working to solve the problem. This is because, if we analyze the code in bold in Figure 5.11, the variable *howmany* is limiting the parallelization. Once we realized that, we changed the code to the version that can be seen in Figure 5.13, in order to avoid wasting threads and spreading the work more accurately.

```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey) {
   double diff, sum=0.0;
   #pragma omp parallel private(diff) {
          double aux = 0;
          int howmany=omp_get_num_threads();
          int blockid = omp_get_thread_num();
          int i_start = lowerb(blockid, howmany, sizex);
          int i_end = upperb(blockid, howmany, sizex);
          for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
                 for (int j=1; j<= sizey-2; j++) {
                        utmp[i*sizey+j]= 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 = utmp[i*sizey+j] - u[i*sizey + j];
                        aux += diff * diff;
```

```
                    }
                }
                #pragma omp atomic
                sum += aux;
        }
        return sum;
}
```

Figure 5.13 First version of Jacobi's data decomposition.

By not limiting the parallel directive with *num_threads* OpenMP uses all the threads available, hence, the execution will be much better and we will not waste threads.

Figure 5.14 Jacobi's parallelization where it can be seen that it used all available threads.

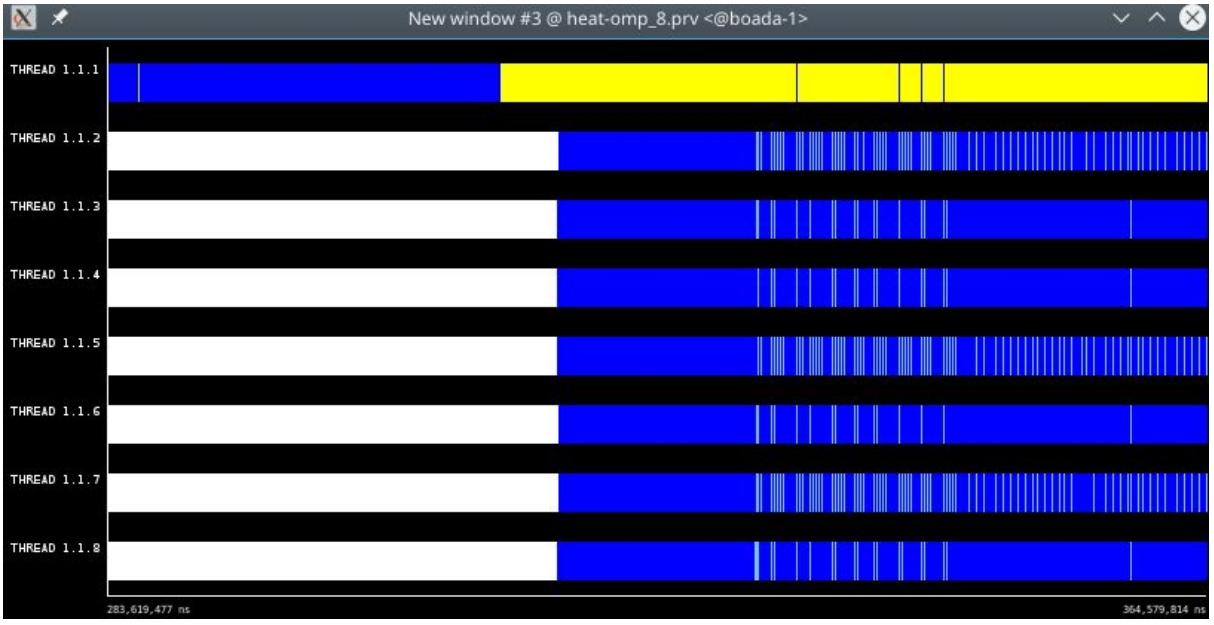

Figure 5.15 Zoom in of the paraver results, where it can be appreciated that the program is using all available threads.

After this small change we submitted the code, with the appropriate script, in order to obtain the execution and speed up plots. The results obtained can be found in Figure 5.15.

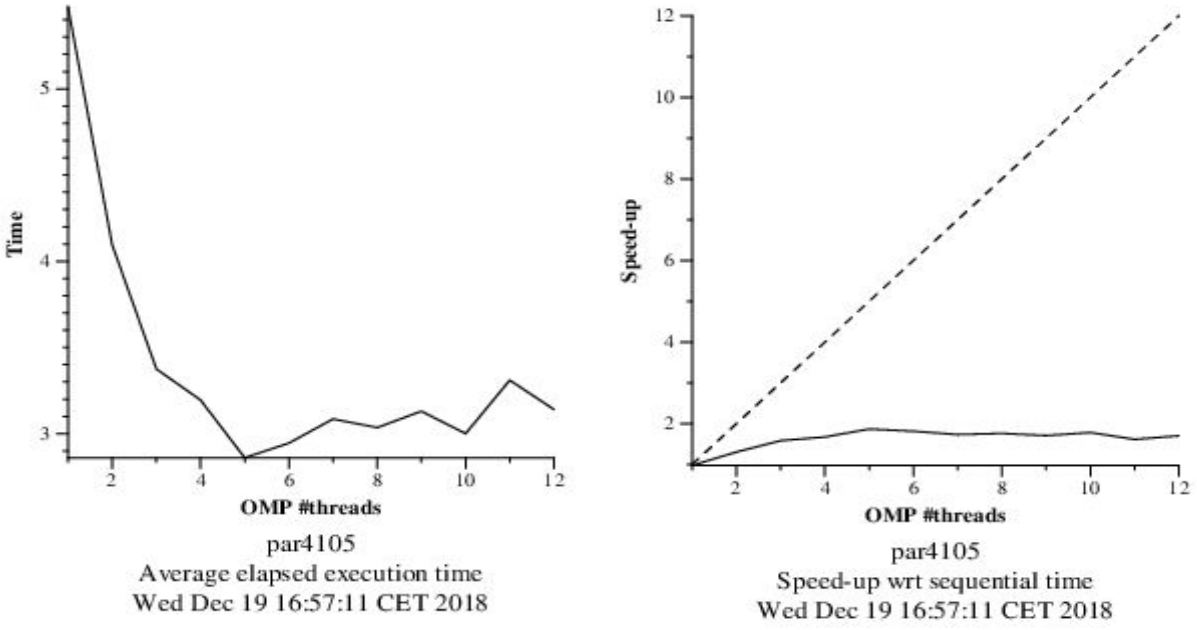

Figure 5.16 Average elapsed execution time and Speed up plots obtained after submitting our data decomposition.

At this point in our study we realized that more improvements could be done, in order to obtain better performance. In particular we found that, every time we were executing Jacobi's algorithm, a copy of the matrix was needed, subsequently reducing our execution time. Then we decided to work with pointers to reduce the time needed for every copy, although other strategies could have been applied such as parallelizing the loops needed for executing the replica.
In Figure 5.17 can be seen the snipped of the code implemented in order to optimize this part of our program.

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey)
{
   memcpy(v, u, sizex*sizey);
}
```

Figure 5.17 Improvement of the copy of the matrix, needed by Jacobi's algorithm.

Then we decided to submit again our code, in order to get the timing plots, hoping that our performance had been improved.

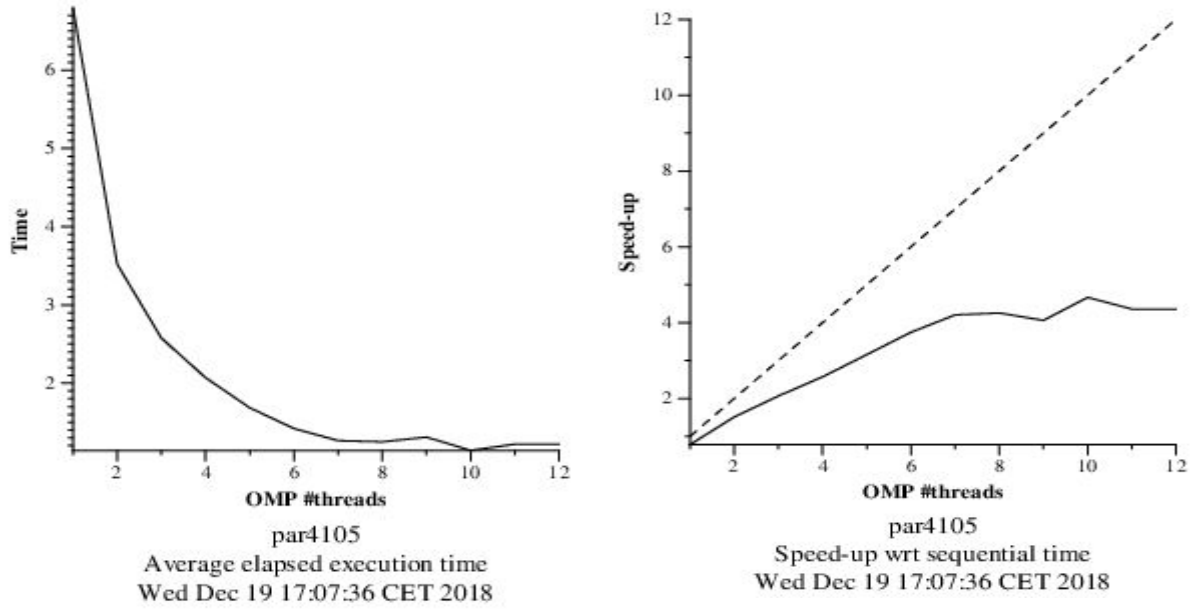

Figure 5.18 Average elapsed execution time and Speed up plots obtained after submitting our data decomposition with copy improvement.

It is clear that our speed up had increased by a factor of four, just by changing a small piece of code. This shows us that we, as a programmers, need to take into an account that every decision we take while programming can affect either positively or negatively to our performance.

Before finishing this section we provide the final paraver results, now with all the improvements together, in order to see the big picture of our program.

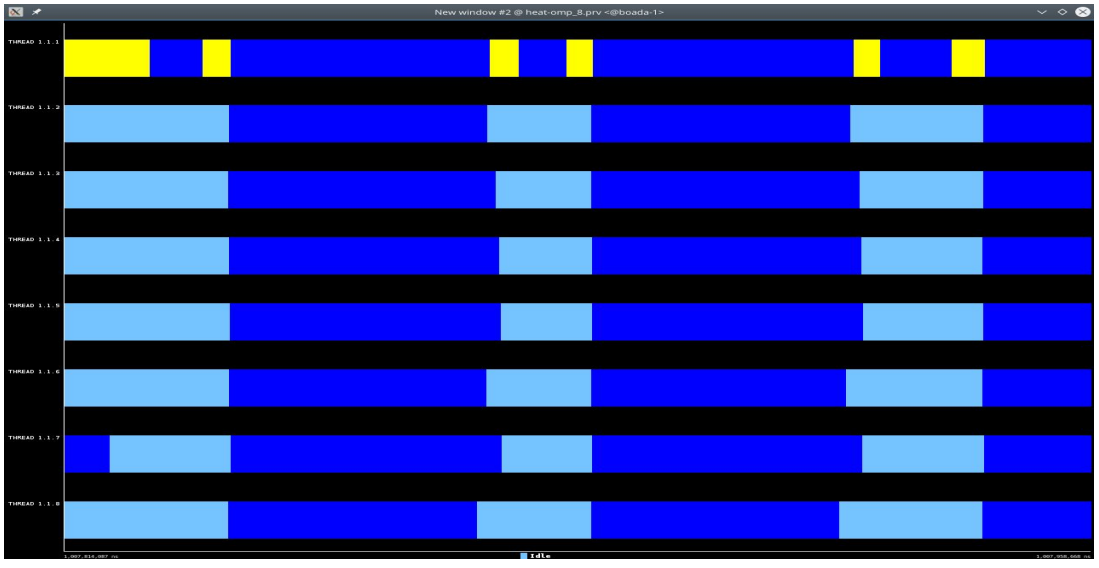

Figure 5.19 Paraver results with all the optimizations implemented. This image is a zoom in on the middle region of the screen.

Parallelization of Gauss-Seidel with OpenMP ordered

For this part of the report we have parallelized Gauss algorithm applying a different strategy that we have previously done. For this particular algorithm we need to take into an account that a dependency has to be respected, hence, we can't program Gauss algorithm following the same strategy of dividing the matrix by blocks of rows and assigning to different threads. Also Gauss does not use the *copy_mat* function because a full copy of the matrix is not needed.
The code we have done for this part can be found in Figure 5.20.

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
        double unew, diff, sum=0.0;
        #pragma omp parallel for ordered(2) private(unew,diff) reduction(+:sum)
        for(int blockid = 0; blockid < omp_get_num_threads(); ++blockid) {
                for (int blockid2 = 0; blockid2 <omp_get_num_threads(); ++blockid2) {
                        int howmany = omp_get_num_threads();
                        int i_start = lowerb(blockid, howmany, sizex);
                        int i_end = upperb(blockid, howmany, sizex);
                        int i_start2 = lowerb(blockid2, howmany, sizey);
                        int i_end2 = upperb(blockid2, howmany, sizey);
                        #pragma omp ordered depend(sink: blockid-1,blockid2)
                        for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
                                for (int j= max(1,i_start2); j<= min(sizey-2,i_end2); j++) {
                                        unew= 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 = unew - u[i*sizey+ j];
                                        sum += diff * diff;
                                        u[i*sizey+j]=unew;
                                }
                        }
                        #pragma omp ordered depend(source)
                }
        }
        return sum;
}
```

Figure 5.20 Gauss data decomposition applying ordered clause.

As it can be appreciated, now instead of parallelizing the code by rows, we are dividing the code by columns in order to avoid dependencies between different threads. What ordered clause does is marking the dependencies that our code has. The number inside the ordered means that, in the following two loops, there will be a dependency.
Before the innermost for loop we found another ordered depend clause, that what indicates is there is a dependency in the *blockid - 1* iteration. Finally the last depend clause indicates until this dependency will be valid.

After verifying that our code executes correctly, by executing diff command after the execution of our program, we submitted our binary, along with *submit-omp-strong.sh* script, in order to get the plots needed for studying the code that we have programmed.

```
par4105@boada-1:~/lab5$
par4105@boada-1:~/lab5$ diff heat-gauss.ppm heat.ppm
par4105@boada-1:~/lab5$
```

Figure 5.21 diff command result, indicating that any difference has been found.

The plots showing the results obtained can be found in the following Figure.

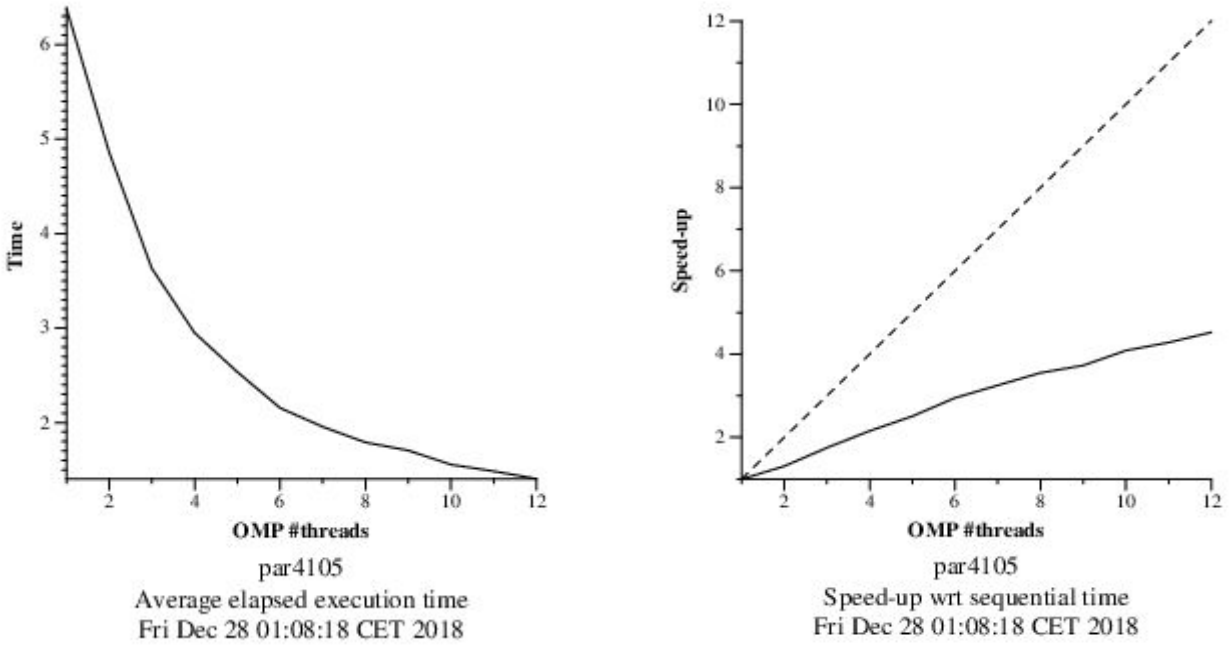

Figure 5.22 Execution time and speedup plots.

For twelve threads we get a tiny increase for our speed up plot. It seems that de speed up is more consistent during the increment of threads than it was in the Jacobi's parallelization. The average elapsed time, by contrast, needs more threads to reduce the program below two units of time that what Jacobi's algorithm needed.

Figure 5.23 Paraver results.

Before finishing with this section we got the trace, in order to analyse in detail the execution of our algorithm, using Paraver tools. As it can be seen in the previous Figure, Paraver software, shows us that a lot of Scheduling and Forking are needed.
With this screenshot it's not easy to see under the hood of every yellow line shown in that window. We have got another image, in Figure 5.24, to show in detail what threads are doing in the middle of the execution of the program.

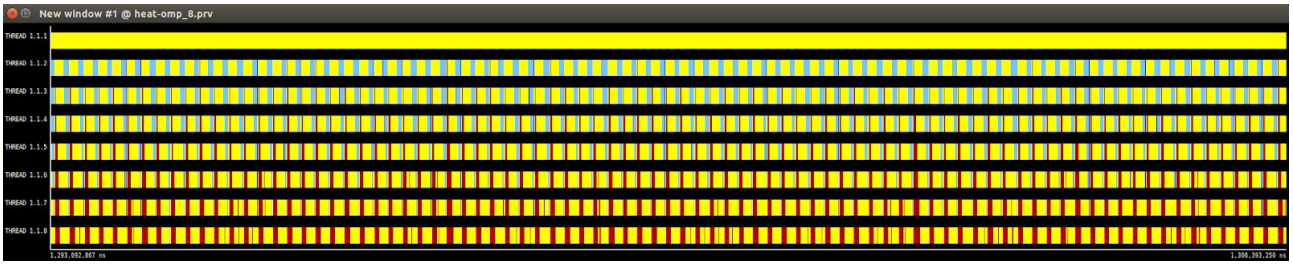

Figure 5.24 Paraver results with zoom, in the middle of the window.

Opcional

In this part we must try to implement an alternative parallel version for Gauss-Seidel using #pragma omp task and task dependences. Then we must compare the solution with the solution that we have got before. The code we have done for this part can be found in Figure 5.25. We are conscious that the execution time could be better but it is the best solution that we find. However we think the first version with ordered clause is better because the execution time and the speedup plots, which we can see in the Figure 5.26, are better than this alternative version with task clauses.

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey) {
   double unew, diff, sum=0.0;
   int howmany = omp_get_max_threads();
   char matrix[howmany][howmany];
   #pragma omp parallel
   #pragma omp single
   for (int blockid = 0; blockid < howmany; ++blockid) {
            int i_start = lowerb(blockid, howmany, sizex);
            int i_end = upperb(blockid, howmany, sizex);
            for (int z = 0; z < howmany; z++) {
                     int j_start = lowerb(z, howmany, sizey);
                     int j_end = upperb(z,howmany, sizey);
                          #pragma omp task firstprivate (j_start,j_end, i_start, i_end) depend(in:
matrix[max(blockid-1,0)][z], matrix[blockid][max(0,z-1)]) depend (out: matrix[blockid][z]) private(diff,unew)
                     {
                     double sum2 = 0.0;
                              for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
                              for (int j = max(1, j_start); j<= min(j_end, sizey-2); j++) {
                                       unew= 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 = unew - u[i*sizey+ j];
                                       sum2 += diff * diff;
                                       u[i*sizey+j]=unew;
                                  }
                              }
                              #pragma omp atomic
                              sum += sum2;
                              }
                     }
   }
   return sum;
}
```

Figure 5.25 Gauss data decomposition applying #pragma omp task and task dependences.

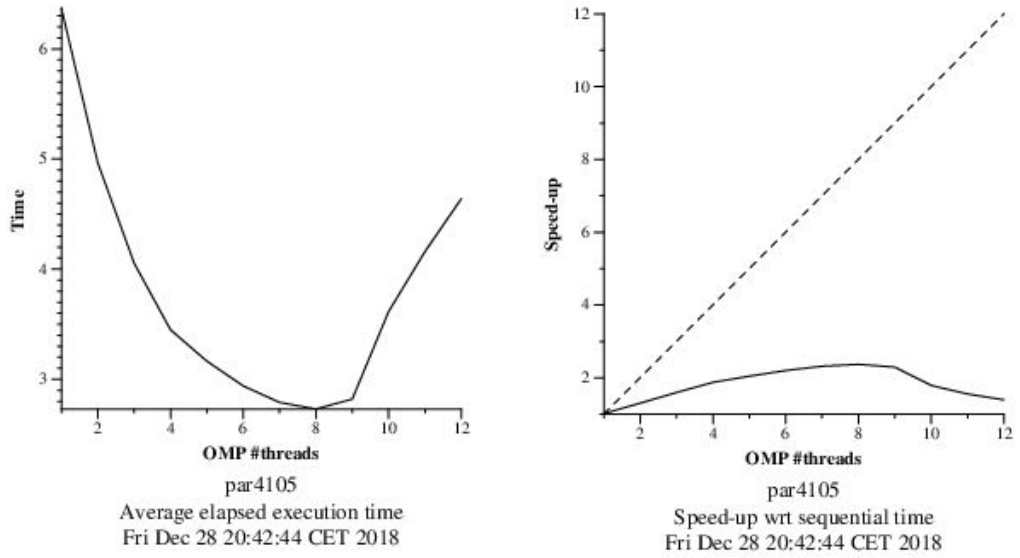

Figure 5.26 Execution time and speedup plots.

Conclusions

This laboratory session has taught us that not always a simple data decomposition can be applied in every algorithm. We need to consider each situation individually and, for doing so, a good comprehension of the code is a must.
Apart from that, we need to know that, by parallelizing only one portion of the code, we are not getting the most of it. Instead we need to focus on other parts to identify which of them can be optimized, in order to reduce execution times.