PAR Laboratory Assignment

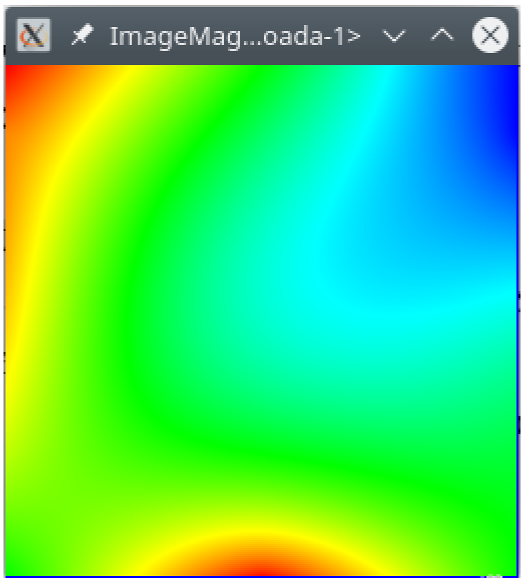
Lab 3: Iterative task decomposition with OpenMP: the computation of the Mandelbrot set

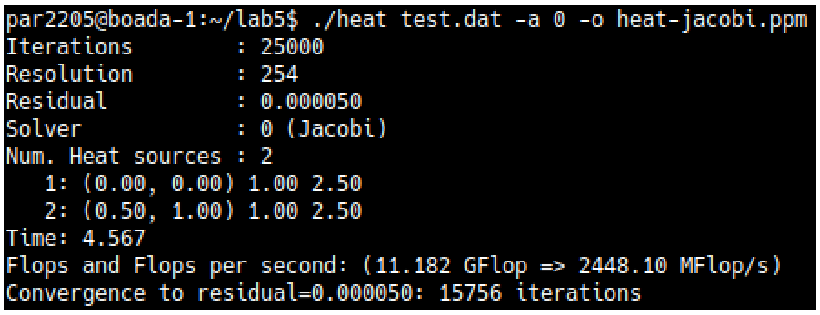
Alvaro Moreno Ribot, PAR 2214
Albert Escoté Alvarez, PAR 2205

Table of contents

Introduction

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

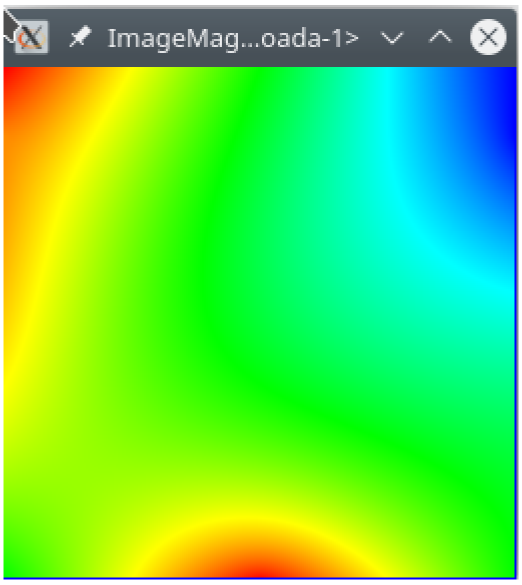


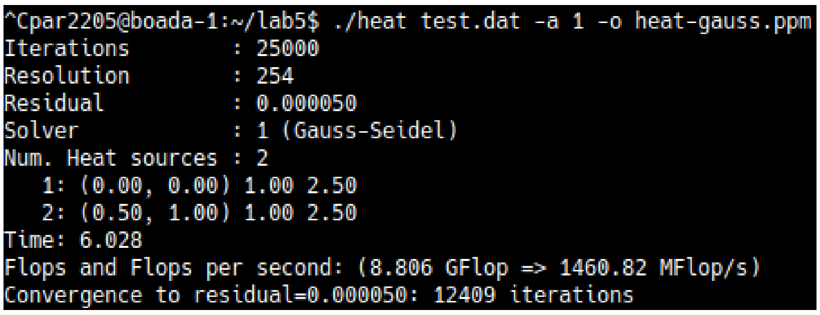


```
par2205@boada-1:~/lab5$ ./heat test.dat -a 0 -o heat-jacobi.ppm
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 4.567
Flops and Flops per second: (11.182 GFlop => 2448.10 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

First of all, we compile the **heat.c** program and execute the binary file generated using the *Jacobi* solver. We obtain the following image file and results:

Now we switch the solver to the *Gauss-Seidel* solver and obtain the following results:





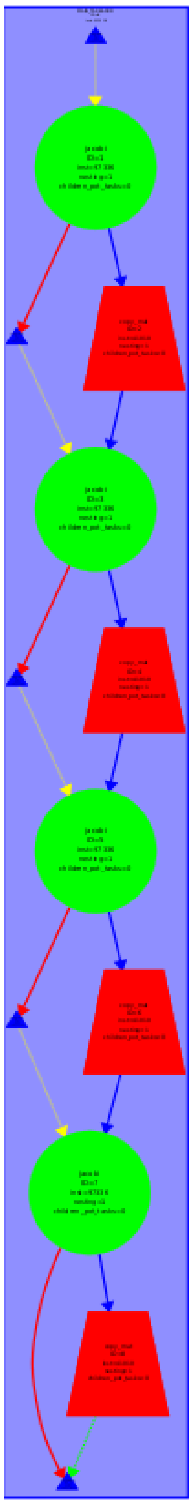
```
^Cpar2205@boada-1:~/lab5$ ./heat test.dat -a 1 -o heat-gauss.ppm
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 1 (Gauss-Seidel)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 6.028
Flops and Flops per second: (8.806 GFlop => 1460.82 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

Sequential heat diffusion program and analysis with Tareador

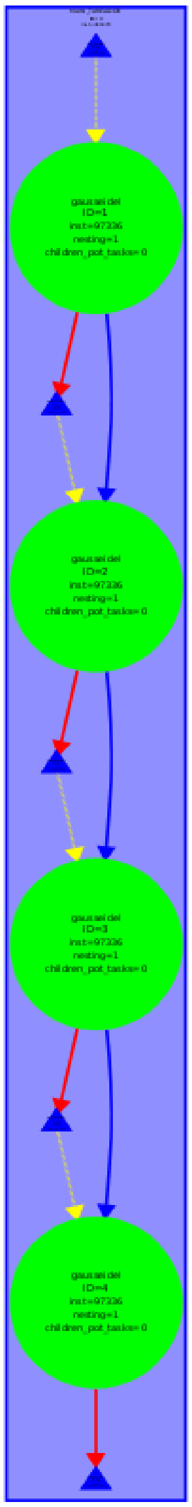
In this section we will be using Tareador to analyse the TDGs generated when using the different solvers. The TDGs will be really useful in order to decide and implement the best parallelization strategy.

Once we have understood the code of **heat.c** and **solver.c** now we can proceed to analyse the task graphs generated when using the two different solvers. To do this we take a look at **heat-tareador.c** and **solver-tareador.c** and compile and execute them with the **run-tareador.sh** scrpit. After executing it, for both solvers, we obtain the following task graphs:

Jacobi



Gauss-Seidel



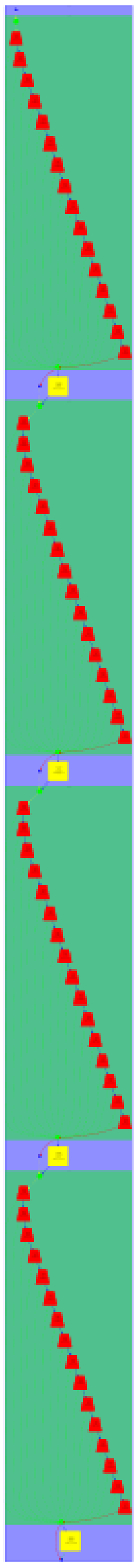
For the *Jacobi* solver we have two tasks defined: a task called '*jacobi*' (green task) and a task called '*copy_mat'* (red task). For the *Gauss-Seidel* solver we only have one task defined: a task called '*gausseidel*' (green task). As we can see, there is not much parallelisation to be exploited at this granularity level.

But now we are going to explore a finer granularity for both solvers. We are asked to obtain a granularity level of one task per block, so we add some Tareador instrumentation in the **solver-tareador.c** program:

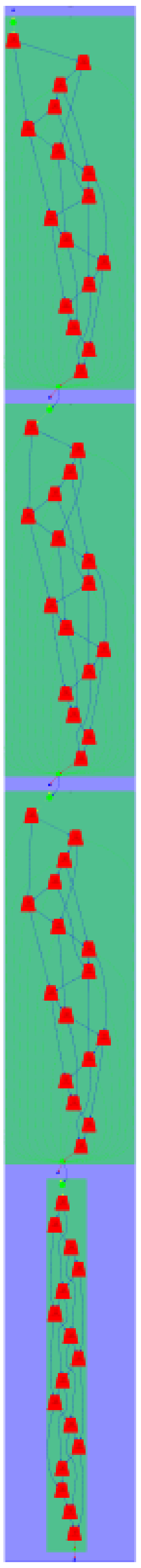
```
// 2D-blocked solver: one iteration step
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
      double tmp, diff, sum=0.0;

      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);
      for (int blockj=0; blockj<nblocksj; ++blockj) {
      tareador_start_task("Inner task");
      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("Inner task");
      }
      }
      //tareador_enable_object(&sum);

      return sum;
}
```

As we can see, we created a task in every iteration of the inner for loop for each block. The results regarding the dependency graph can be seen in the following graphs dependence generated by Tareador:

Jacobi



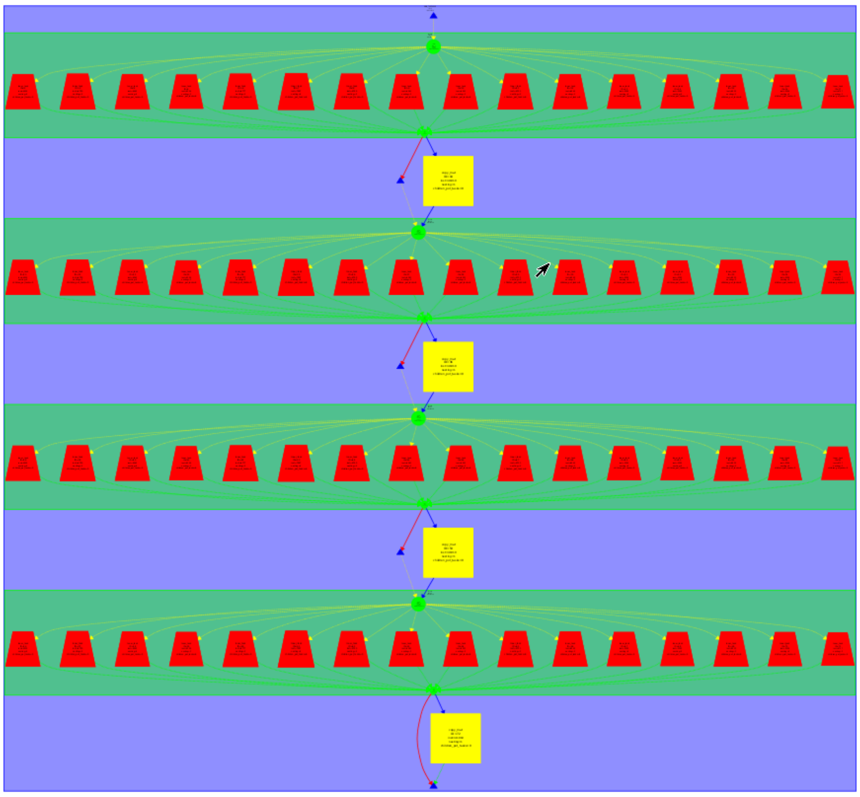
Gauss-Seidel

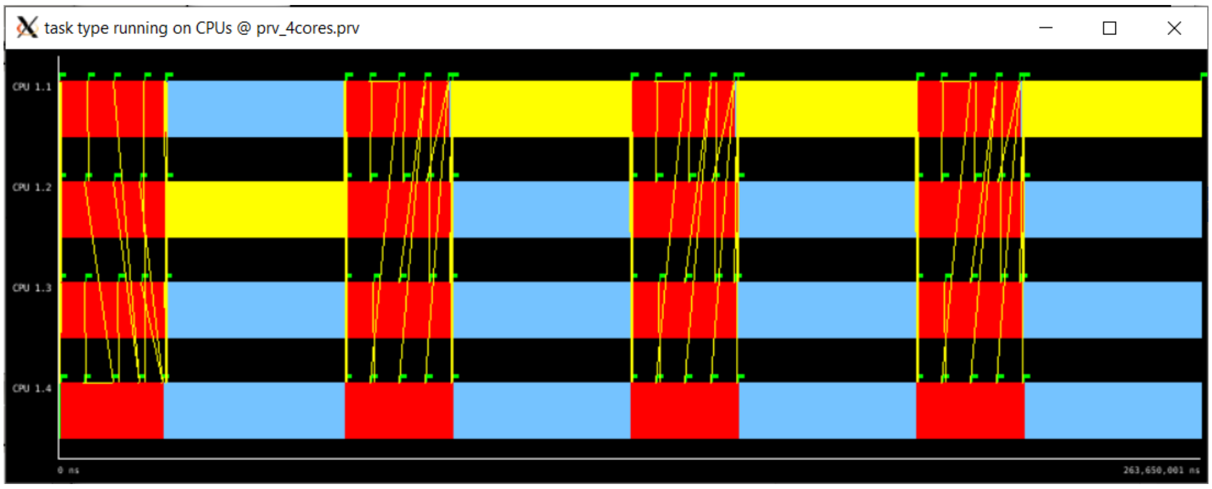


As we can see, there is no parallelisation in both versions. That is due to the fact that both codes share the variable “sum” between threads to store partial results. So what we need is to protect the variable to avoid data races.

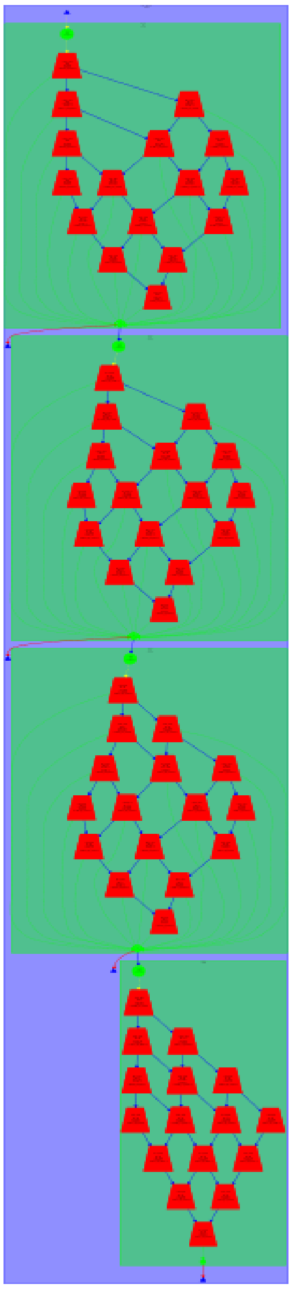
To protect it, we used the Tareador functions **tareador_disable_object(&var)** and **tareador_enable_object(&var)**, that simulates a reduction of the variable. So we uncomment these functions that were already in the code and recompile and execute the program. To protect this variable in our OpenMP implementation we could add the **reduction(+: sum)** directive on the parallel section creation.

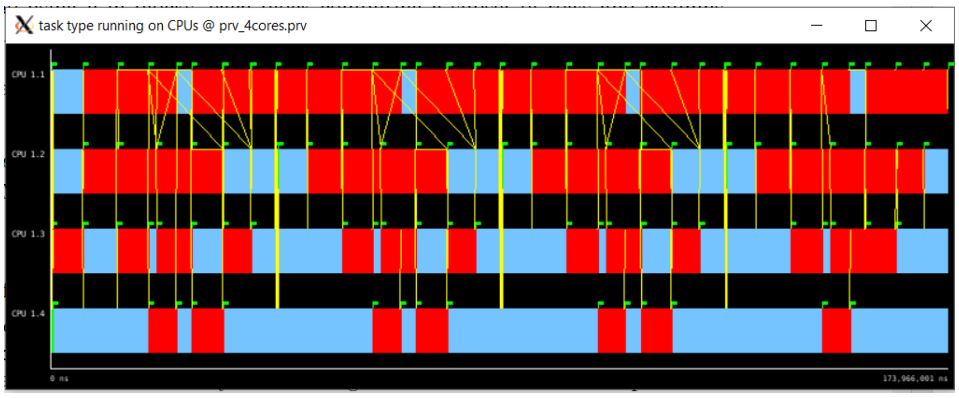
So we first try the new task dependency for the *Jacobi* solver and observe a significant difference in the parallelisation, now we have more parallelism to be exploited. We also simulate the execution when using 4 processors.





We did the same for the *Gauss-Seidel*.



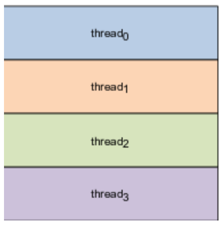


As we can see, we still obtained some dependencies, but with the Tareador instrumentation we can not obtain a better parallelism.

OpenMP parallelization and execution analysis: Jacobi

From what we learnt from the previous analysis, we will be parallelizing using the #pragma omp constructions we learnt along the course. We will not be implementing the maximum granularity as we are not working with an ideal environment and the overhead would be too high for it to be efficient. To solve this we will apply a data decomposition. First we will be choosing between the input and output data structures that need to be decomposed. Then we will be dividing it across tasks following the known data distributions we know (cyclic, block and block-cyclic).

Here we need to group consecutive pixels, so we will be using blocks. The block size will be $n\,rows \div n\,threads$, this way a block will be done by a thread (a block per thread). As we are looking at the Jacobi method, we know we will have a table of size N. We will divide N by the number of threads. We can visualize it as the small image on the right.



The code block below is the implementation for the Jacobi using the **#pragma omp parallel for** in the first loop. To solve the **sum** dependency, as we have already mentioned before, we will be using a reduction (same for the variable **diff**).

The results from the sequential execution of the Jacobi heat generation we got are the following:

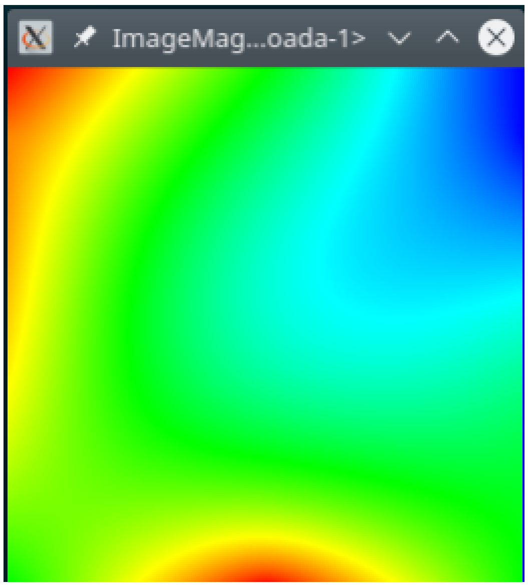
```
par2214@boada-1:~/lab5$ ./heat test.dat -a 0 -o heat-jacobi-seq.ppm
Iterations     : 25000
Resolution     : 254
Residual       : 0.000050
Solver         : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 9.718
Flops and Flops per second: (11.182 GFlop => 1150.56 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

We can see the time highlighted in a pink color. The following block shows the same information for the parallelized version using the **#pragma omp parallel for** directive:

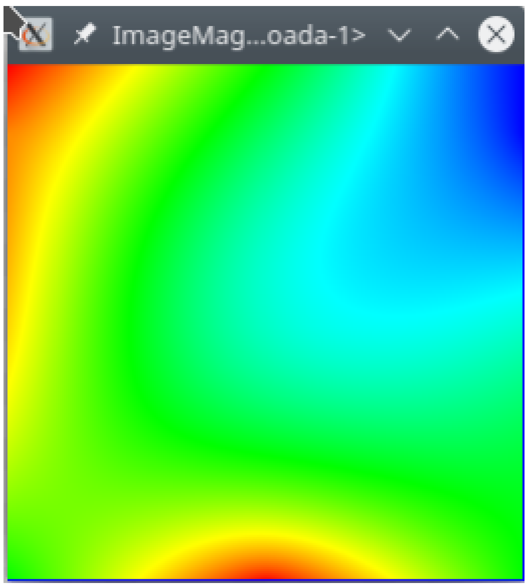
```
par2214@boada-1:~/lab5$ cat heat-omp-jacobi-8-boada-4.txt
Iterations     : 25000
Resolution     : 254
Residual       : 0.000050
Solver         : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 4.376
Flops and Flops per second: (11.182 GFlop => 2555.23 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Note that with the parallelization the time has halved, so this is a big improvement. The following code is the one we used in order to parallelize the Jacobi version:

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

       int nblocksi=8;
       int nblocksj=1;
       for (int blocki=0; blocki<nblocksi; ++blocki) {
       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);
        #pragma omp parallel for private(diff) reduction(+:sum)
        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;
          }
        }
       }
       }

       return sum;
}
```



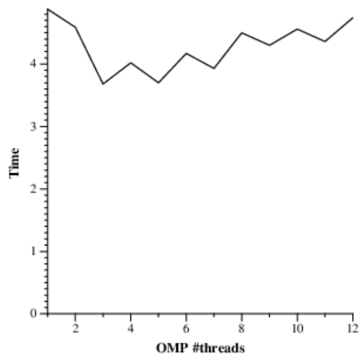
Parallel generation



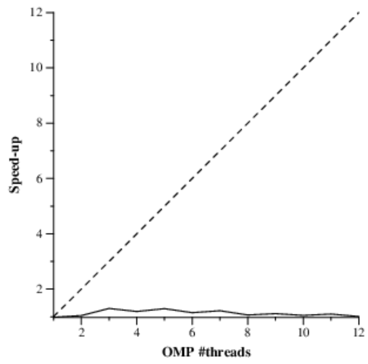
Sequential generation

We have checked the differences between the two generated files to make sure that there are no errors and they are identical, so there are no errors for the parallel version of this code.

```
par2214@boada-1:~/lab5$ diff -s heat-jacobi.ppm heat-jacobi-seq.ppm
Files heat-jacobi.ppm and heat-jacobi-seq.ppm are identical
```



par2214
Average elapsed execution time
Thu May 27 13:33:42 CEST 2021



par2214
Speed-up wrt sequential time
Thu May 27 13:33:42 CEST 2021

The plot on the left shows the scalability for the Jacobi execution. The speedup we obtain is practically insignificant. We don't really know why this is happening. Also the execution time is more or less regular all along the plot. So we are not seeing any benefits from the parallelization that we just made.

We think that there might be synchronization problems.

We had a lot of problems wiht the remote execution of some programs like **wxparaver** and **tareador**. We constantly go a message saying "Instance is too busy" and also ***Failed to load module "canberra-gtk-module"***. We were not able to get all the information we were asked.

OpenMP parallelization and execution analysis: Gauss-Seidel

In this section we will make a parallelization of the function **solve**. We realized that each access to the matrix has a dependency with the positions (i - 1, j) and (i, j - 1) of the same matrix. We approached this problem by applying a similar geometric data decomposition as we did on section 2.

We can distribute the rows in blocks among the threads as we did before by just parallelizing the outer for loop, and on the second loop we also divide the iterations by column blocks. This ends with a block decomposition strategy. We do this so each thread can depend only on the block that has on top of him on position (row - 1, column) as the dependency of (row, column - 1) is already covered by the fact that the execution of a row is done by the same thread on the same task, which means that is doing it ordered, what takes care of the dependency.

We tried to reach the solution we explained before but we have some troubles at implementing it. So for this section we are not going to obtain the results expected. But we thought that for *Gauss-Seidel* solver, the parallelism exploited would make, as a consequence of the amount of task synchronization, to not reduce the execution time as *Jacobi* does.

Conclusions

In this laboratory we have been able to parallelize a complex code of two equation solvers for a real world problem. We also experimented what can happen with performance whenever there are dependencies and we learned to solve it.

We conclude that in this subject we learned from basic to more advanced concepts about parallelization using good and real-world material, like boada or paraver. It has been a good experience and we have learnt a lot in these laboratory sessions.