PARALLELISM

Laboratory 5: Geometric (data) decomposition: heat diffusion equation

PAR1105

Albert Borrellas Maldonado

Víctor Martínez Murillo

27-12-2019

Fall 2019-2020

1. Sequential heat diffusion program

In this final lab assignment, we will study the parallelization of a sequential code named "heat.c" that simulates heat diffusion in a solid body using two different solvers for the heat equation (*Jacobi* and *Gauss-Seidel*).
The assignment is divided into two parts, one for *Jacobi* and one for *Gauss-Seidel*.
The codes generate a picture similar to the one below, representing the aforementioned heat diffusion. Dark blue indicates cold and red indicates hot.

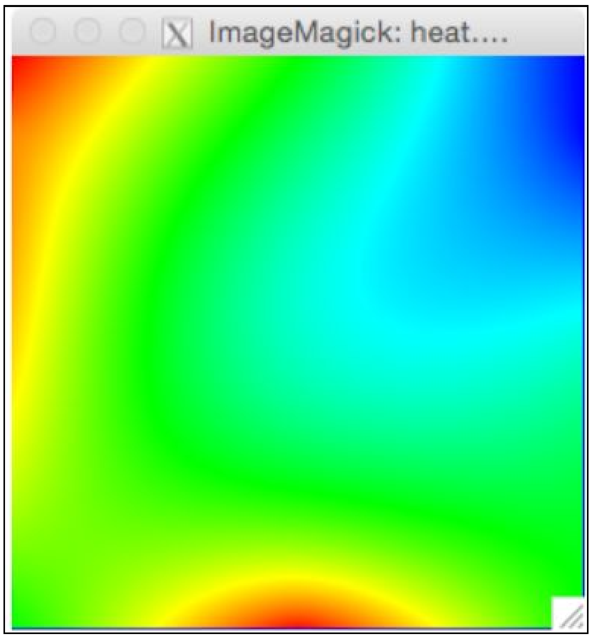
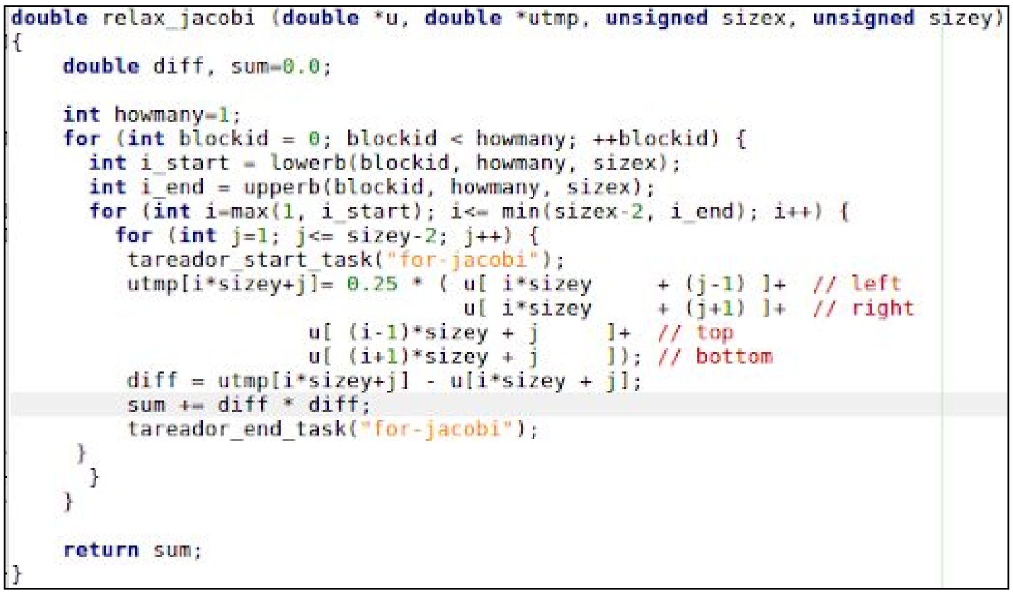


Image 1: Picture generated with the execution of heat.c

2. Analysis with Tareador

We are going to consider the possible parallelism strategies by checking the analysis with *Tareador*. We are going to use it in both *Jacobi* and *Gauss-Seidel* solvers, to simulate the dependences on both of their parallelization.
The strategy will parallelize each iteration in the innermost loop of both solvers generating the finest-grain task decomposition possible.
First of all, images 2 and 3 are the original code of both solvers with the *Tareador* functions included.

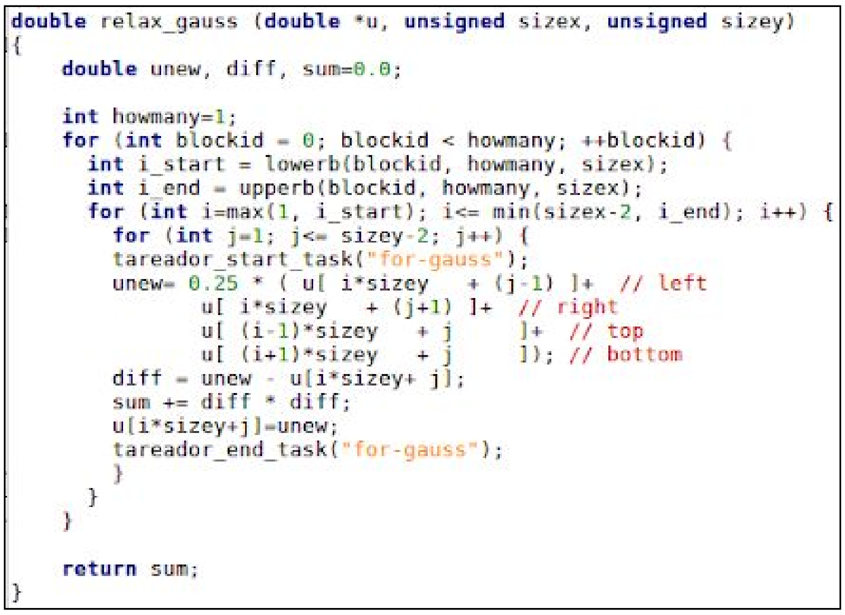


```c
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;

    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("for-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
         diff = utmp[i*sizey+j] - u[i*sizey + j];
         sum += diff * diff;
         tareador_end_task("for-jacobi");
     }
      }
    }

    return sum;
}
```

Image 2: Tareador Jacobi solver code.



```c
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, 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("for-gauss");
        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;
        tareador_end_task("for-gauss");
        }
      }
    }

    return sum;
}
```

Image 3: Tareador Gauss-Seidel solver code.

The above codes produce the task-dependencies graphs shown in image 4, respectively. It is seen that there is little parallelism and that there is a problem in load balancing between tasks. *Jacobi* solver includes more tasks as it uses an auxiliary function called *copy_mat*, used for copying a matrix.

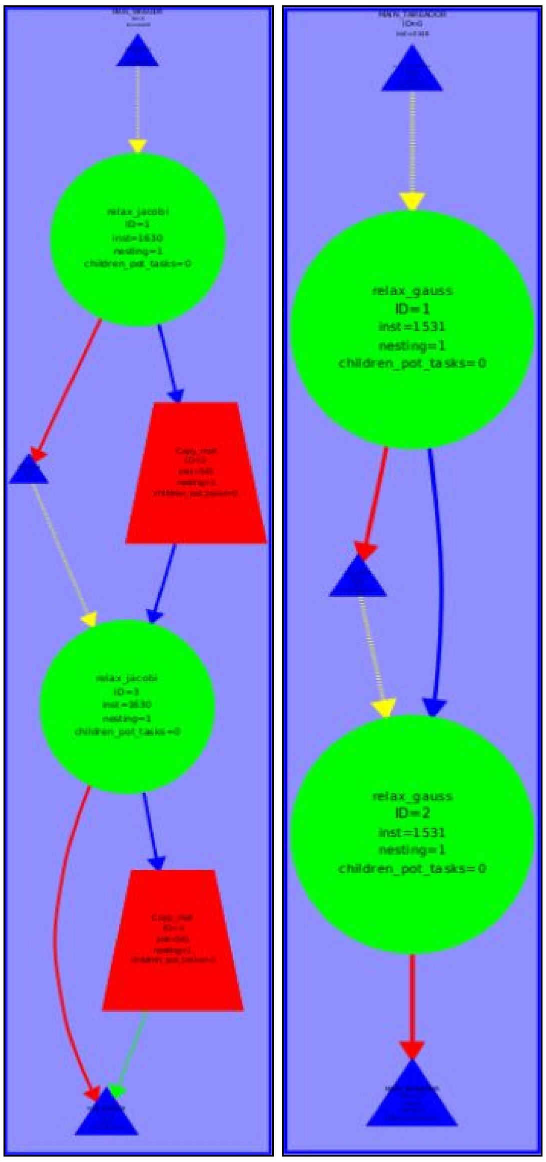
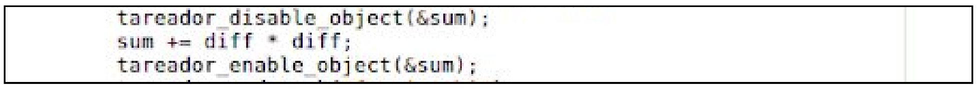


Image 4: Task-dependencies graphs, Jacobi (left) and Gauss-Seidel (right).

Due to dependencies problems, there is a need of ignoring variable sum (as it stores partial results from threads during execution) when analyzing with *Tareador*. Using two functions we achieve easily the aim of this part of the session:



```
tareador_disable_object(&sum);
sum += diff * diff;
tareador_enable_object(&sum);
```

Image 5: Portion of code added to ignore variable sum in *Tareador.*

After running *Tareador* with the new code, we obtain two different task-dependencies graphs, one for each solver.

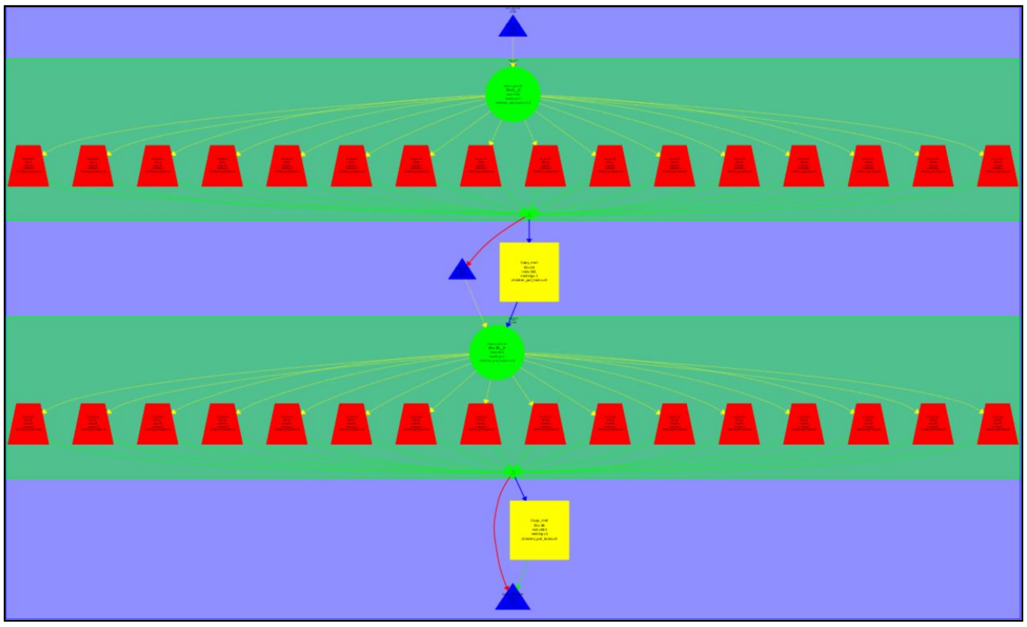


Image 6: Task-dependencies graph for Jacobi solver (without sum).

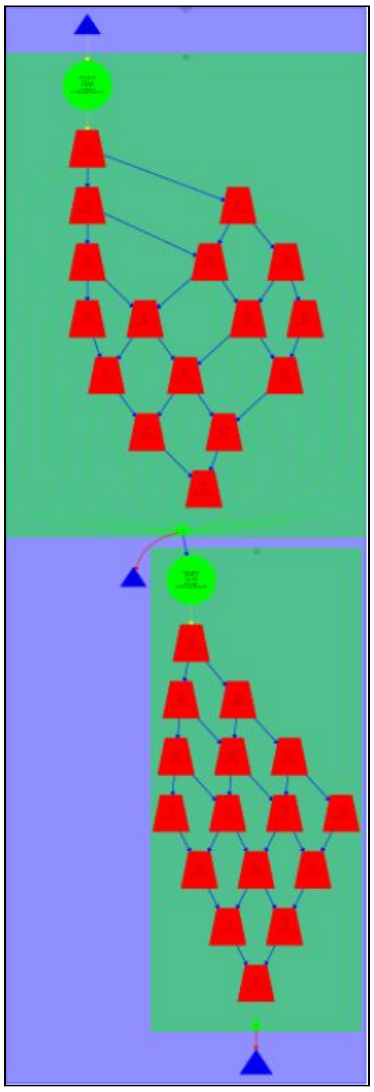


Image 7: Task-dependencies graph for Gauss-Seidel solver (without sum).

Compared to the previous *Tareador* snapshot, there is an improvement in image 6 and 7, not only in parallelism (in which *Jacobi* is higher) also in load balancing, which both solvers have improved it.

3. Parallelization of Jacobi with OpenMP parallel

In this laboratory session we will parallelize the sequential code for *Jacobi* using **#pragma omp parallel** (neither the clause *for* nor the combination parallel + for can be used)**.** With this restrictions we will select a block data decomposition where we divide all the iterations in the inner loop, where each block-id corresponds to a thread. Each block of iterations is (almost) of the same size (last iterations may vary according to the remainder of number of iterations divided by number of threads).

block-id = 0
1
2
3

Figure 1: Block data decomposition.

After executing the code in boada 1-4 and checking that the code works, we use *Paraver* to see the timeline window where the color yellow indicates the creation of new tasks and the color dark blue where the threads are running tasks.

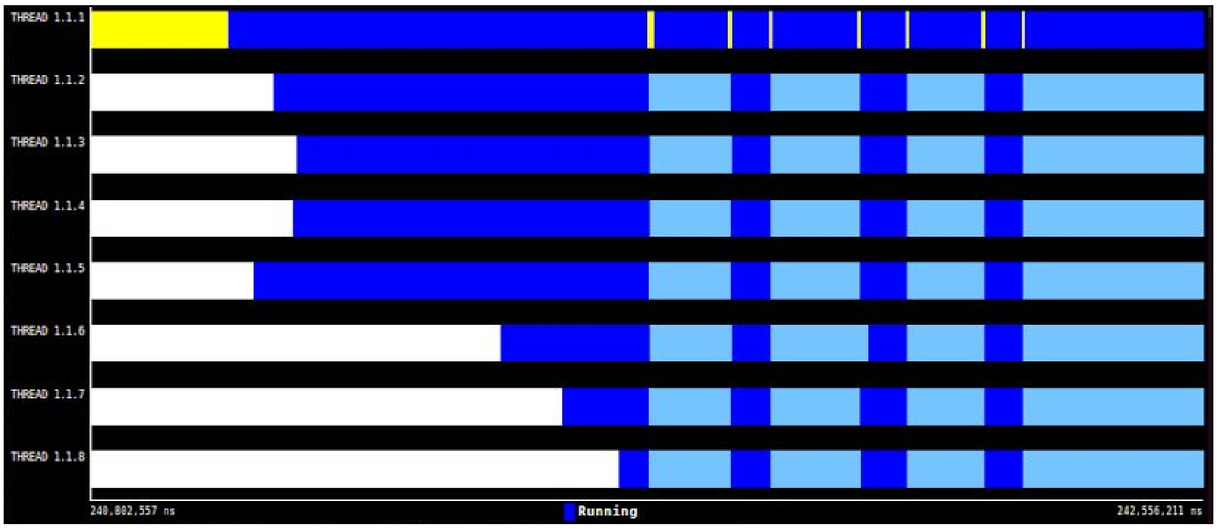
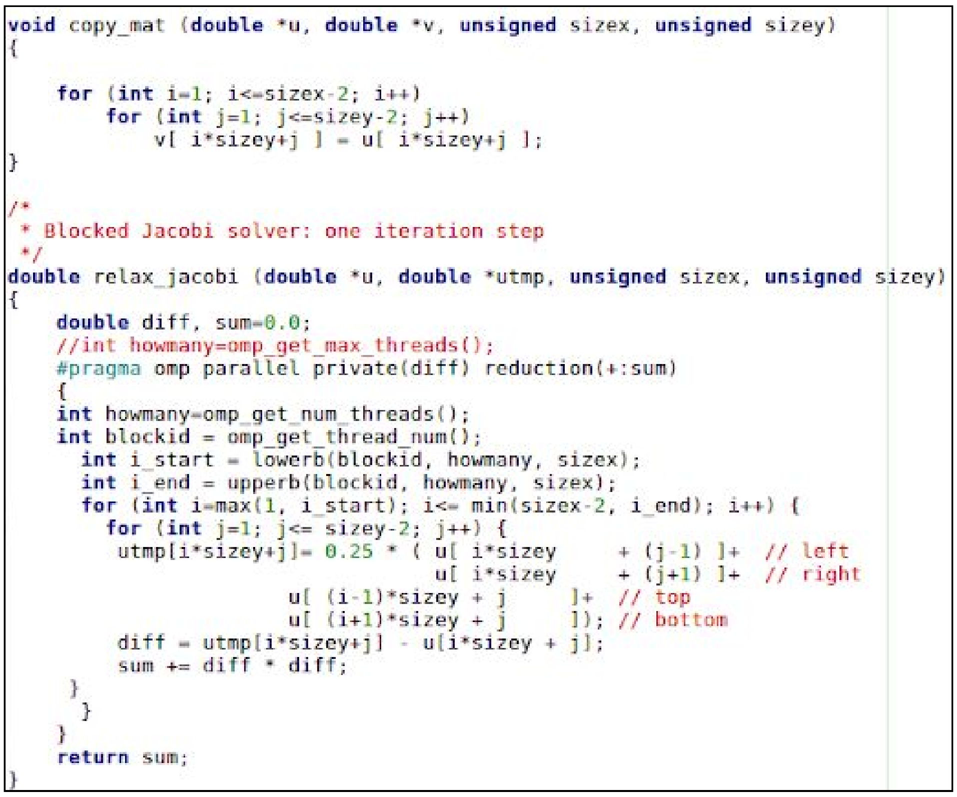


Image 8: Timeline plot of the parallelized version for Jacobi solver (bottlenecked).

As seen in Image 10, only thread 1 is creating new tasks, this is due to a bottleneck caused by the *howmany* variable, initially equal to 1.

The bottleneck is solved by using the fucking *omp_get_num_threads* from OpenMP.



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

    for (int i=1; i<=sizex-2; i++)
        for (int j=1; j<=sizey-2; j++)
            v[ i*sizey+j ] = u[ i*sizey+j ];
}

/*
 * Blocked Jacobi solver: one iteration step
 */
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;
    //int howmany=omp_get_max_threads();
    #pragma omp parallel private(diff) reduction(+:sum)
    {
    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];
         sum += diff * diff;
     }
      }
    }
    return sum;
}
```

Image 9: Parallelization of the Jacobi code.

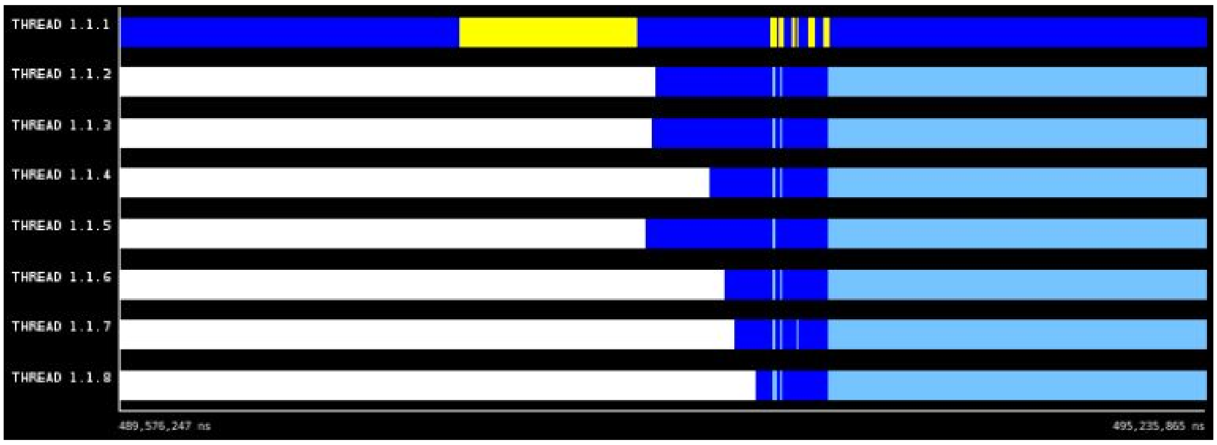


Image 10: Timeline plot of the parallelized version for Jacobi solver (no-bottlenecked).

Using the aforementioned function, the bottleneck is solved and now every threads creates new tasks improving the efficiency of the program. Another improvement is parallelizing the function *copy_mat*, using a #pragma omp parallel for.

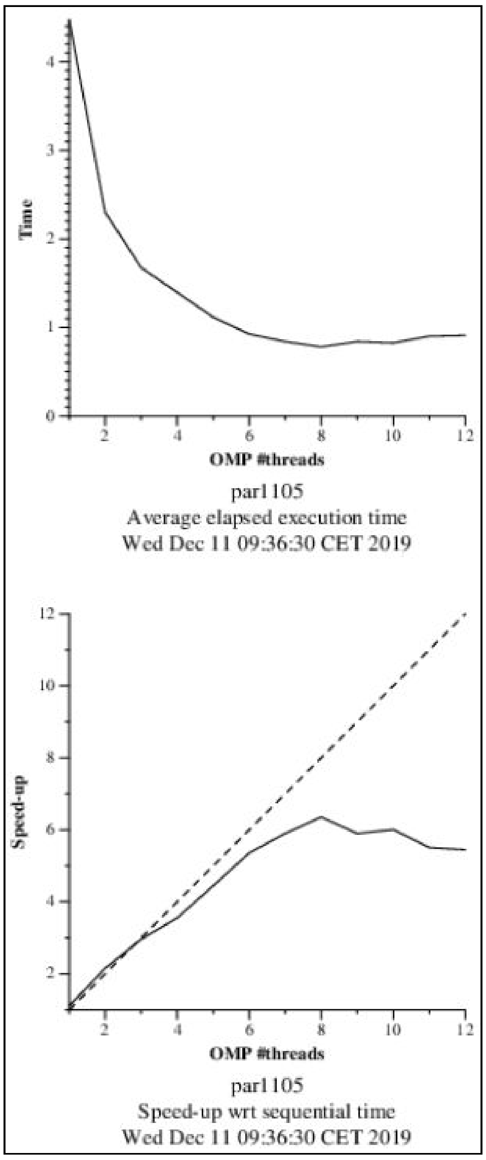
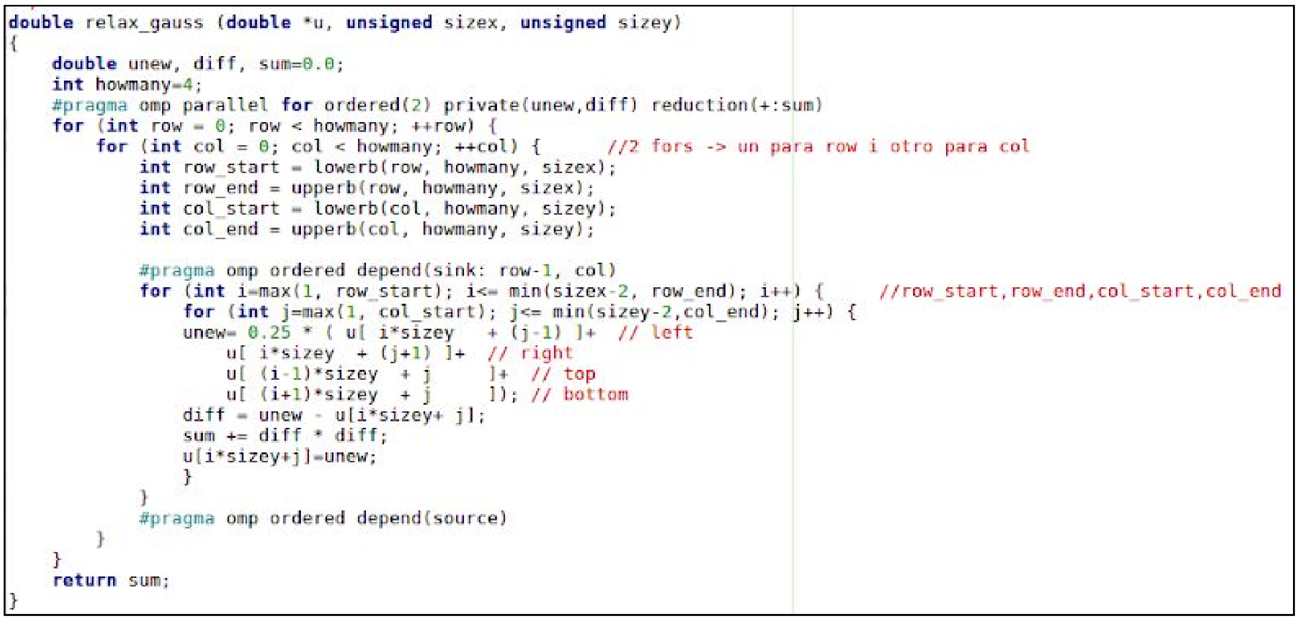


Image 11: Strong scalability plot of the parallelized version for Jacobi solver.

Finally, analyzing the scalability plot it is seen that there is a good speed-up (nearly the ideal case) until the number of threads is equal to 8. This may occur due to the block size not being enough to distribute between so many threads.

4. Parallelization of Gauss-Seidel with OpenMP ordered

In this final laboratory session we have the task to parallelize the Gauss-*Seidel* solver using #pragma omp for with the ordered clause in it. We will have to find a way to synchronize the parallel execution of the rows assigned to each processor in order to guarantee the dependences that we detected with *Tareador*.



```c
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;
    int howmany=4;
    #pragma omp parallel for ordered(2) private(unew,diff) reduction(+:sum)
    for (int row = 0; row < howmany; ++row) {
        for (int col = 0; col < howmany; ++col) {      //2 fors -> un para row i otro para col
            int row_start = lowerb(row, howmany, sizex);
            int row_end = upperb(row, howmany, sizex);
            int col_start = lowerb(col, howmany, sizey);
            int col_end = upperb(col, howmany, sizey);

            #pragma omp ordered depend(sink: row-1, col)
            for (int i=max(1, row_start); i<= min(sizex-2, row_end); i++) {      //row_start,row_end,col_start,col_end
                for (int j=max(1, col_start); j<= min(sizey-2,col_end); 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;
}
```

Image 12: Parallelized code for Gauss-Seidel solver.

A block decomposition will also be used in this code to avoid the dependencies between iterations. The main difference compared to the *Jacobi* code is that two *for* loops (x-axis and y-axis) are needed to distribute the iterations with this decomposition.

We did not achieve a desired strong scalability plot (as seen in Image 13), but it is supposed to be better in performance than *Jacobi*, as Gauss-Seidel does not need an auxiliary matrix and, consequently, using the function *copy_mat*. Therefore, the execution time would be better.

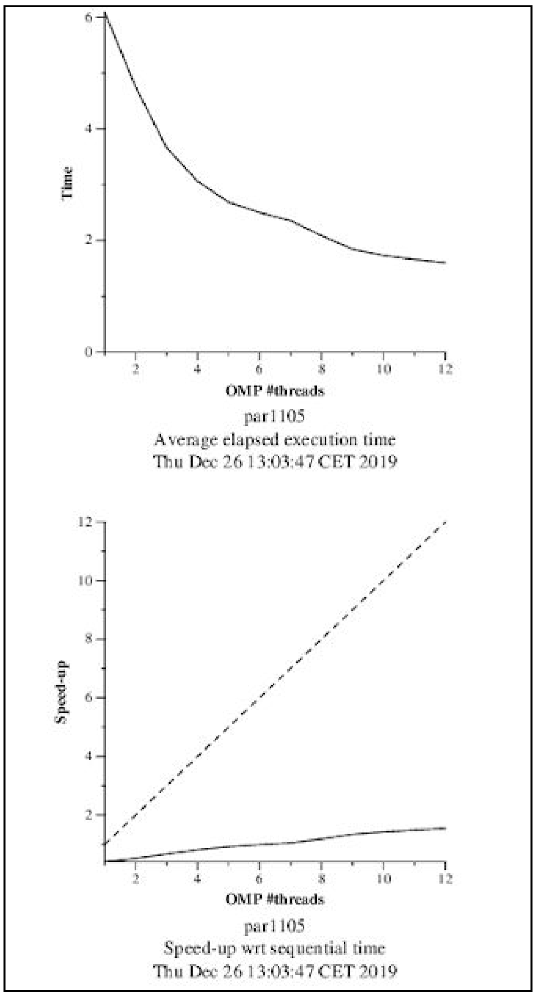


Image 13: Strong scalability plot of parallelized Gauss-Seidel solver.