PAR Laboratory Assignment Lab 5: Geometric (data) decomposition: heat diffusion equation

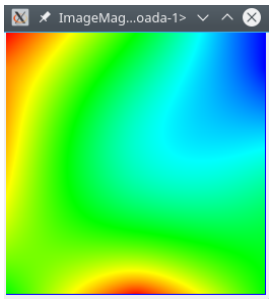
Ismael Douha Prieto, Eloi Cruz Harillo - Group 1305

Q1 2019/20

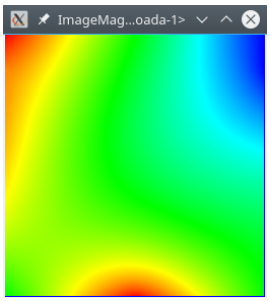
1 Introduction

In this paper we will analyse the implementation of two algorithms that are able to simulate the heat diffusion in a two dimension solid body. They are called Jacobi and Gauss-Seidel and use different strategies to generate the result. Initially we are provided with the sequential implementation of the algorithms so in the next sections we will analyse and parallel them.

The first step has been executing both programs specifying in each one the max number of simulation steps (iterations), the size of the body (resolution) and the heat sources, their position, size and temperature. Doing this we have obtained the images in the Figure 1. Image (a) shows the heat map for the Jacobi program and (b) the heat map for the Gauss algorithm.



(a) Heatmap using Jacobi algorithm.



(b) Heatmap using Gauss algorithm.

Figure 1: Heatmap result of the execution of the heat.c script.

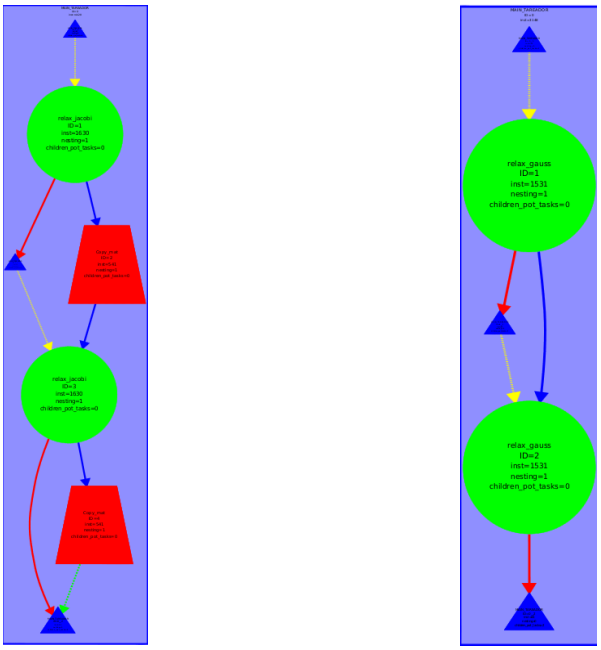
2 Parallelization strategies

To begin with we will explain the two algorithms involved in the assignment. As we explained in the introduction they both simulate the heat diffusion in a two dimension solid body. Nevertheless they use different strategies so the results obtained are similar but not equal. To represent the solid body they use a matrix were different heat sources are defined. The two algorithms define a number of iterations. For each iteration the new value for a cell is defined as the sum of all its surrounding cells(up, bottom, left and right). With this procedure the heat is expanded in every iteration from the initial points to all the matrix. The difference between them is that Gauss only uses one matrix for the whole procedure but Jacobi uses two of them. The main matrix saves the actual values calculated and the auxiliary one saves the temporal values of every cell. At the end

of the iteration the auxiliary matrix is copied into the main matrix.

We are provided with the code for the two algorithms explained above. In the original implementation they use a coarse-grain task decomposition strategy because they don't parallelize the work of the matrix treatment so tasks are really big. Figure 2 shows the graphs obtained with Tareador executing only two iterations. This is not a real scenario because with two iterations we can't obtain an usable result but it is going to be useful to detect the tasks that take most time of the execution and dependencies between them.

In one hand the Jacobi graph shows two different types of tasks. The ones represented by a green circle are the actual data processing (the calculation of the new value for every cell). The ones represented by a red quadrilateral are the copy of the matrix. We can see that for every iteration we first calculate all the points and after that we copy all the auxiliary matrix the the original matrix. We can see that it is a sequential procedure because to begin an iteration we have to wait to the previous one to finish. At the same time to copy the auxiliary matrix we have to wait until it is completely calculated. In the other hand Gauss graph only shows one type of task represented by a green circle. As we have explained this algorithm does all the procedure over the same matrix so it doesn't need to do any copy.



(a) Initial Jacobi decomposition. (b) Initial Gauss decomposition.

Figure 2: Graphs for a coarse-grain decomposition strategy.

Once we have understood how this algorithms work we will proceed to analyse their actual behaviour. To do this we will change the granularity of the tasks created. To do this we will define every task as an instance of the innermost loop. Figure 3 shows the changes in the code that uses a finer-grain task decomposition strategy. The execution of this two codes produces the graphs in figure 4. The Jacobi one shows a dependence between tasks that calculate all the values (represented by a green circle). We have concluded that these dependencies are produced because of the variable named sum that stores partial results and is shared between threads. The Gauss graph has

a lot of dependencies for two reasons. The first one is the dependence created by the variable sum. The second reason is that neighbour cells produce dependencies. To achieve better parallelism we disabled the sum variable using the sable-object and disable-object commands. The actual result is shown in the figure 5 where tasks are now executed in parallel for the Jacobi algorithm and less dependencies are shown in the Gauss graph (it produces a diagonal execution because of the dependence between the neighbour cells). In the next section we will study how to reduce them.

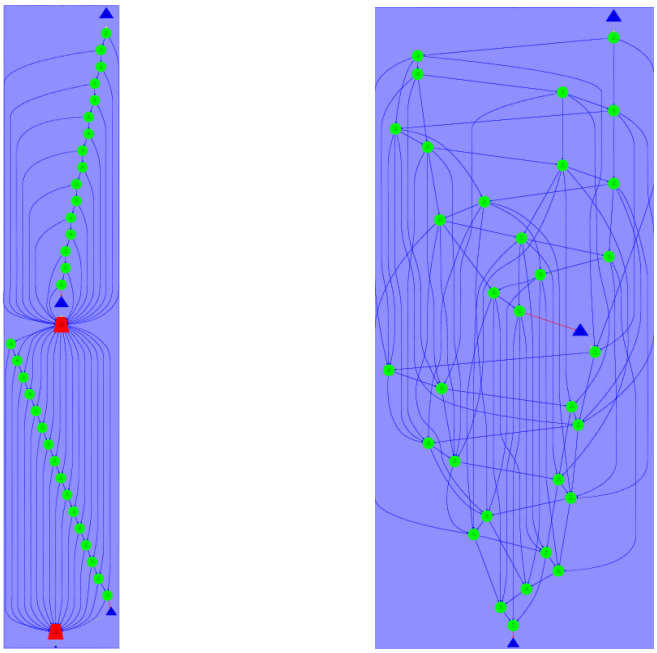
```
for (int blockid = 0; blockid < howmany; ++blockid) {
  int i_start = lowerb(blockid, howmany, sizex);
  int i_end = upperb(blockid, howmany, sizex);
  for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
    for (int j=1; j<= sizey-2; j++) {
         tareador_start_task("relax_jacobi_intern");
         tareador_disable_object(&sum);
         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_enable_object(&sum);
    tareador_end_task("relax_jacobi_intern");
     }
  }
}
```

(a) Jacobi code.

```
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("relax_gauss");
         tareador_disable_object(&sum);
         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_enable_object(&sum);
         tareador_end_task("relax_gauss");
    }
  }
}
```

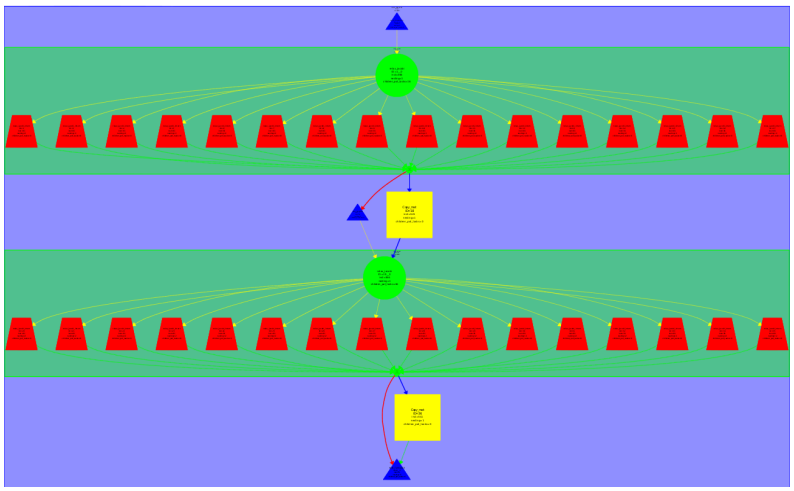
(b) Gauss code.

Figure 3: Code used for the task decomposition with Tareador.

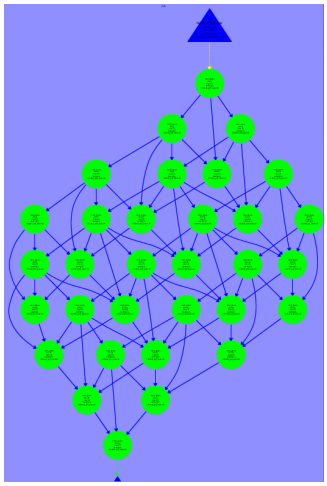


(a) Finer-grain Jacobi decomposition. (b) Finer-grain Gauss decomposition.

Figure 4: Graphs for a finer-grain decomposition strategy with dependencies.



(a) Finer-grain Jacobi decomposition.



(b) Finer-grain Gauss decomposition.

Figure 5: Graphs for a finer-grain decomposition strategy without dependencies.

3 Performance evaluation

3.1 Jacobi OpenMP

In order to parallelize the program we parallelized the relax-jacobi function as shown in the figure 6. In this section we will describe the strategies used to solve the dependencies explained in the previous one. To solve the dependency created by the shared variable sum between all threads the reduction clause is used. Doing this all threads work with a local copy of the variable and when they finish the work they add the result to the global sum. What is more the program divides the data in a geometric way as shown in Figure 7. This figure is an example using only four threads where four blocs of consecutive rows are created and then every group of rows is assigned only to one thread. Doing this we reduce the overhead produced by the dependence between neighbour cells because most of the neighbour cells are treated by the same thread.

```c
/*
 * 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();
    int blockid;
    #pragma omp parallel private(diff) reduction(+:sum)
    {
      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;
}
```

Figure 6: relax-jacobi function.

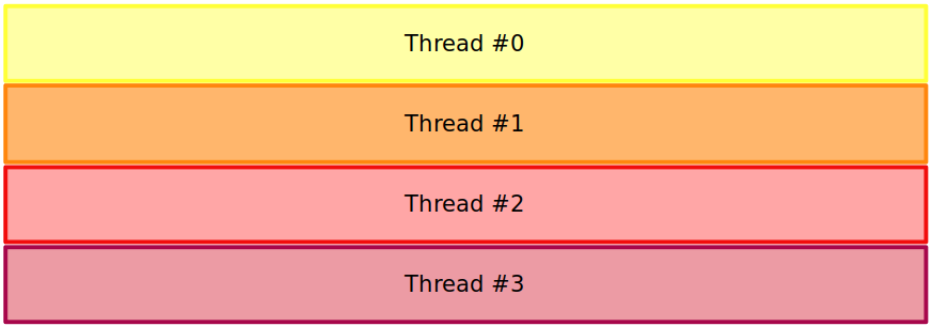


Figure 7: Partition of row blocs between threads.

The first step after creating the parallel code has been checking if the result is correct. Figure 8 shows that the execution has been correct so we proceed extracting data from paraver. Figure 9 shows the result of the parallel execution. We can see that in the beginning of the execution there is a load balancing problem because threads get different load of work, but after the first merge this load problem is solved. We hasn't detected any bottleneck because we used as many threads as available from the beginning using the function omp-get-max-threads.

```
par1305@boada-1:~/lab5$ OMP_NUM_THREADS=8 ./heat-omp test.dat
Iterations        : 25000
Resolution        : 254
Algorithm         : 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: 0.765
Flops and Flops per second: (11.182 GFlop => 14621.57 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
par1305@boada-1:~/lab5$ diff heat.ppm heat-jacobi.ppm
par1305@boada-1:~/lab5$
```

Figure 8: Execution times.

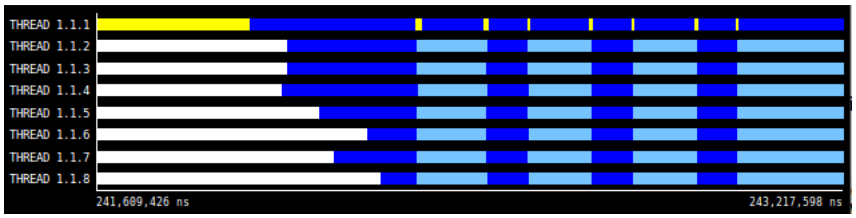


Figure 9: Parallel copy-mat.

In the image extracted from paraver we can see that the relax-jacobi function is executed by all threads and all of them get the same load of work. The big serialization problem appears in the copy-mat function because it is executed only by the first thread. To solve it code shown in the Figure 10 has been used. The load balance at the beginning has not been solved but the serialization problem has disappeared.

```
/*
 * Function to copy one matrix into another
 */

void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey)
{
    #pragma omp parallel for
    for (int i=1; i<=sizex-2; i++)
        for (int j=1; j<=sizey-2; j++)
            v[ i*sizey+j ] = u[ i*sizey+j ];
}
```

Figure 10: copy-mat function.

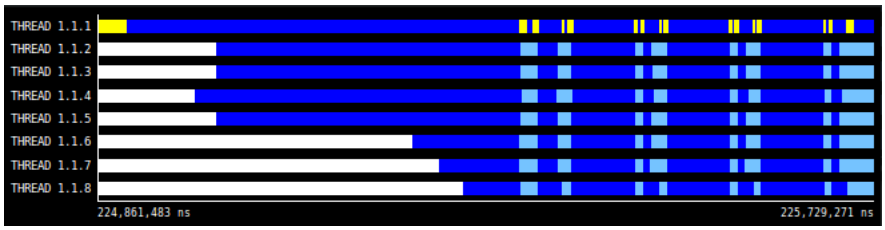


Figure 11: Sequential copy-mat.

Finally we will talk about the strong scalability plot represented in the Figure 12. It shows that the speed-up is increased while increasing the number of threads until we arrive to eight of them. After that the speed-up grows very slowly.

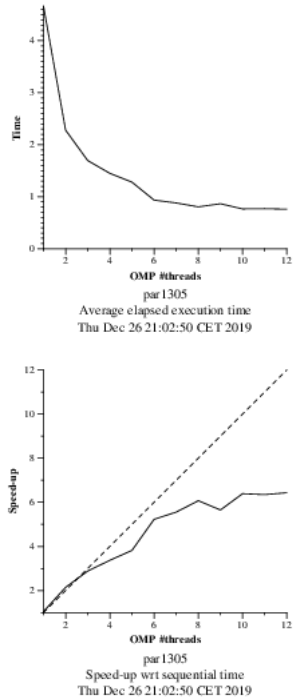


Figure 12: Strong scalability plot.

3.2 Gauss-Seidel OpenMP

In this section we describe our strategy to parallelize the Gauss-Seidel solver. In this case, we used a parallel for to parallelize the relax-Gauss function, with private and reduction clauses as we did with Jacobi function. This time, we also used a ordered clause, which guarantees the order between iterations. For the synchronisation, we used the sink clause, which set the dependencies between iterations. Finally, for divided the computation, we decided to add a new for to divide every row in different columns, making easier the synchronisation.

In Figure 11 you can see our parallelized code.

```c
/*
 * Blocked Gauss-Seidel solver: one iteration step
 */
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;

    int howmany=omp_get_max_threads();
    #pragma omp parallel for ordered(2) private(unew, diff) reduction(+:sum)
    for (int blockid_i = 0; blockid_i < howmany; ++blockid_i) {
        for(int blockid_j = 0; blockid_j < howmany; ++blockid_j){
              int i_start = lowerb(blockid_i, howmany, sizex);
              int i_end = upperb(blockid_i, howmany, sizex);
              int j_start = lowerb(blockid_j, howmany, sizey);
              int j_end = upperb(blockid_j, howmany, sizey);
              #pragma omp ordered depend(sink: blockid_i-1, blockid_j)
              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++) {
                    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 13: relax-gauss function.

Once the code was done, we executed it and we saw that performs better than the serial code.

```
par1305@boada-1:~/lab5$ OMP_NUM_THREADS=8 ./heat-omp test.dat
Iterations        : 25000
Resolution        : 254
Algorithm         : 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: 2.004
Flops and Flops per second: (8.806 GFlop => 4394.86 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
par1305@boada-1:~/lab5$ diff heat.ppm heat-gauss.ppm
par1305@boada-1:~/lab5$
```

Figure 14: Execution time.

With this paraver trace we saw that every thread does not have the same load but in general, each one has the same execution time approximately.

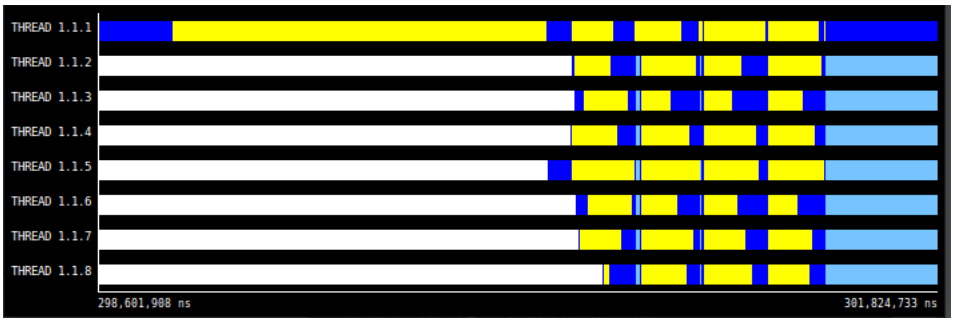


Figure 15: Execution data extracted with paraver.

In addition, with the strong scalability plot, we see how as more threads we have, more parallelism reach it.

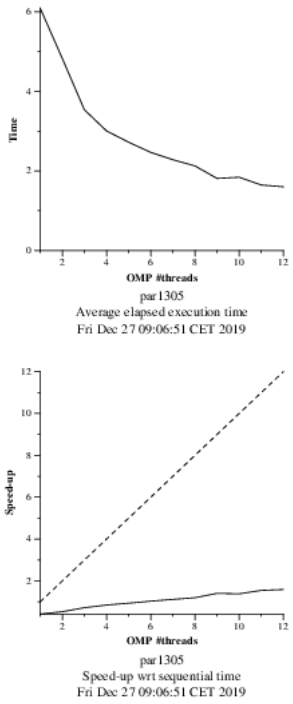
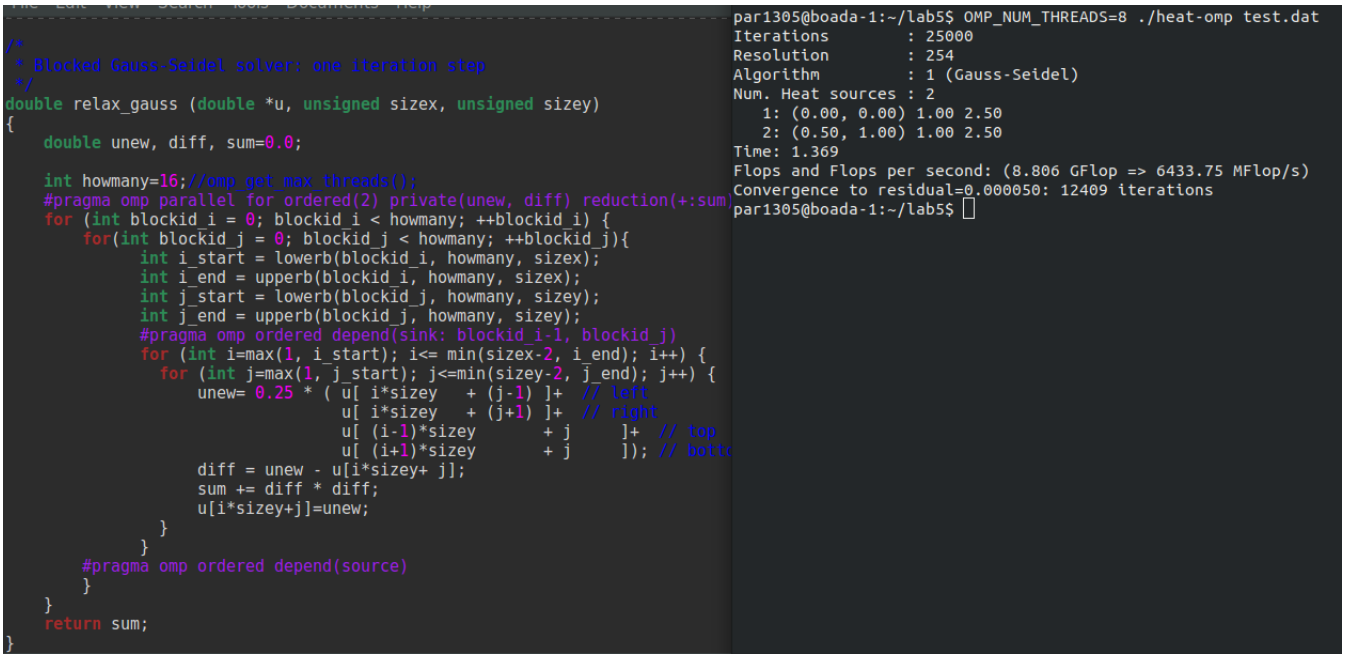


Figure 16: Strong scalability plot.

In order to control better the trade-off synchronisation-computation, you can make bigger or smaller the load of the thread, what directly affects synchronisation time. In our main version, we decided to use the function **omp_get_max_threads()**, but to test which granularity is better, we change it and we hardcoded many values. After some test, we found that 16 is the better value, which is the double of the used threads.



```
/*
 * Blocked Gauss-Seidel solver: one iteration step
 */
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;

    int howmany=16;//omp_get_max_threads();
    #pragma omp parallel for ordered(2) private(unew, diff) reduction(+:sum)
    for (int blockid_i = 0; blockid_i < howmany; ++blockid_i) {
        for(int blockid_j = 0; blockid_j < howmany; ++blockid_j){
              int i_start = lowerb(blockid_i, howmany, sizex);
              int i_end = upperb(blockid_i, howmany, sizex);
              int j_start = lowerb(blockid_j, howmany, sizey);
              int j_end = upperb(blockid_j, howmany, sizey);
              #pragma omp ordered depend(sink: blockid_i-1, blockid_j)
              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++) {
                    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     ]); // bott
                    diff = unew - u[i*sizey+ j];
                    sum += diff * diff;
                    u[i*sizey+j]=unew;
                }
              }
        #pragma omp ordered depend(source)
        }
    }
    return sum;
}
```

```
par1305@boada-1:~/lab5$ OMP_NUM_THREADS=8 ./heat-omp test.dat
Iterations        : 25000
Resolution        : 254
Algorithm         : 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: 1.369
Flops and Flops per second: (8.806 GFlop => 6433.75 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
par1305@boada-1:~/lab5$
```

Figure 17: Performance results for the optim **howmany** value and its code

4 Conclusions

In this last laboratory session we have learned two algorithms that simulate the heat diffusion in a solid body. The main challenge has been to parallelize them. In both cases the solid body is represented by a matrix so we have learned different ways to iterate over all the cells of a matrix in a way that multiple threads can work at the same time. The two algorithms use different strategies to solve the problem so in each case we have found different problems and we have developed different solutions during the assignment.