PAR Laboratory Assignment

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

PAR2303
Joan Manuel Ramos Refusta
Àlex Aguilera Martínez

1.Analysis of task granularities and dependences

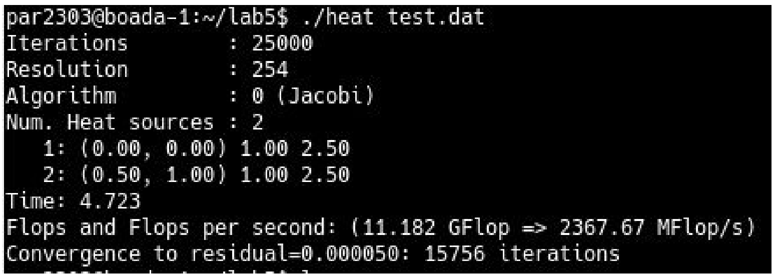
In this session you will work on the parallelization of a sequential code (heat.c) that simulates heat diffusion in a solid body using two different solvers for the heat equation; Jacobi and Gauss-Seidel.

The provided code uses a configuration file (named test.dat) which we have to specify different values like resolution, the algorithm to be used or iterations before the execution.

In this section, we are going to execute the heat program and try to understand what is doing. Then, we will use the tareador tool in order to obtain the task graphs with a finer-grain decomposition to each of the two solvers.

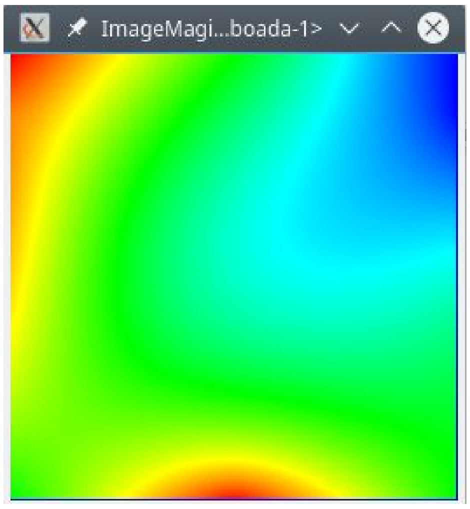
Specifying the algorithm and executing heat.c, we obtain the following results:

Execution of Jacobi:



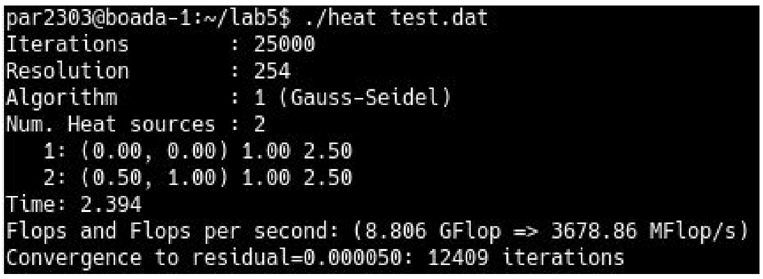
```
par2303@boada-1:~/lab5$ ./heat test.dat
Iterations        : 25000
Resolution        : 254
Algorithm         : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 4.723
Flops and Flops per second: (11.182 GFlop => 2367.67 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Figure 1, execution of heat.c with Jacobi



Execution of Gauss-Seidel:

Figure 2, heat-jacobi.ppm



```
par2303@boada-1:~/lab5$ ./heat 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: 2.394
Flops and Flops per second: (8.806 GFlop => 3678.86 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

Figure 3, execution of heat.c with Gauss-Seidel

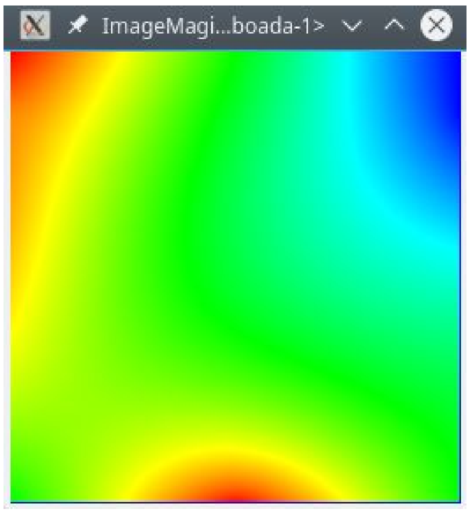
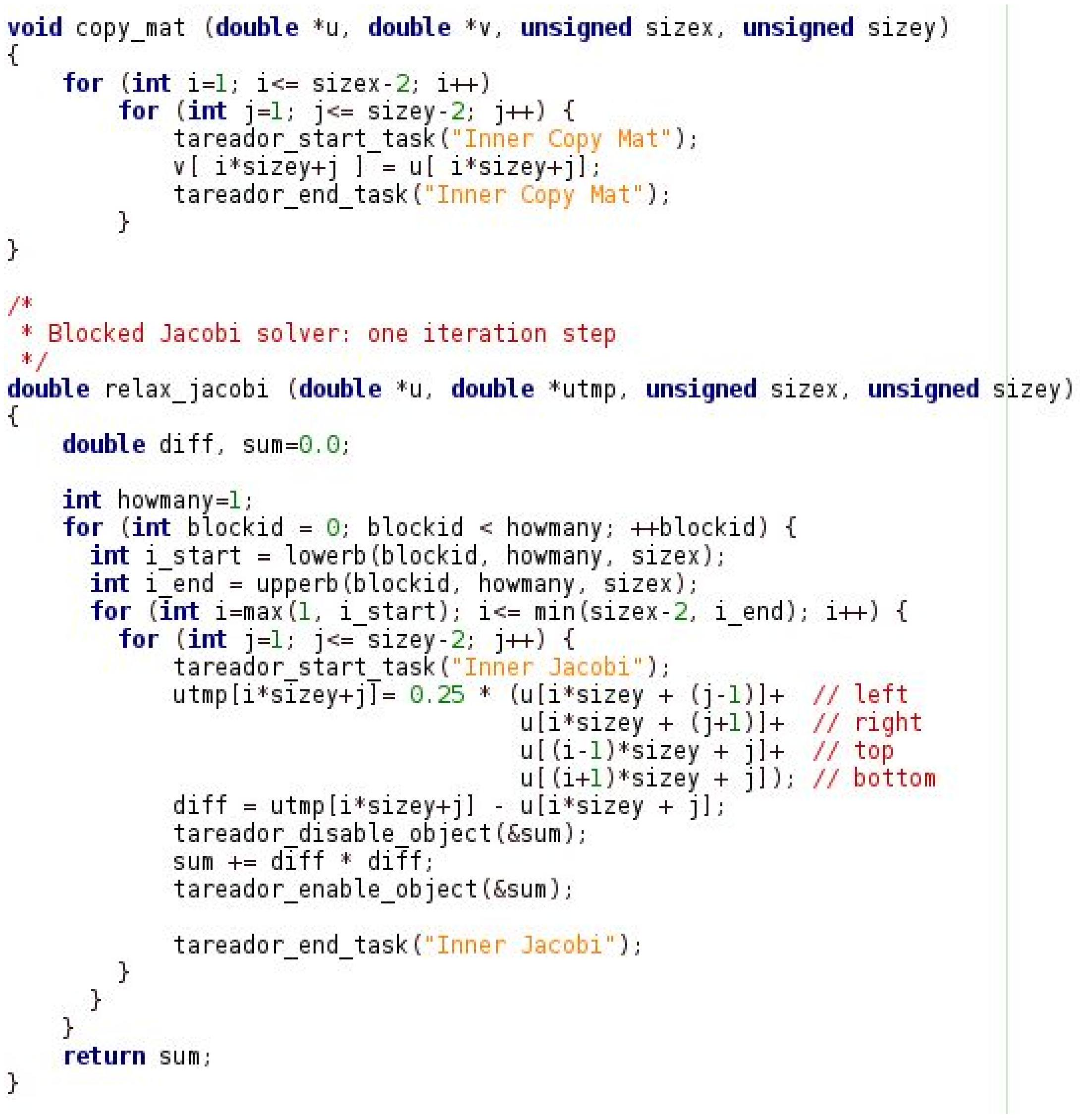


Figure 4, heat-gauss.ppm

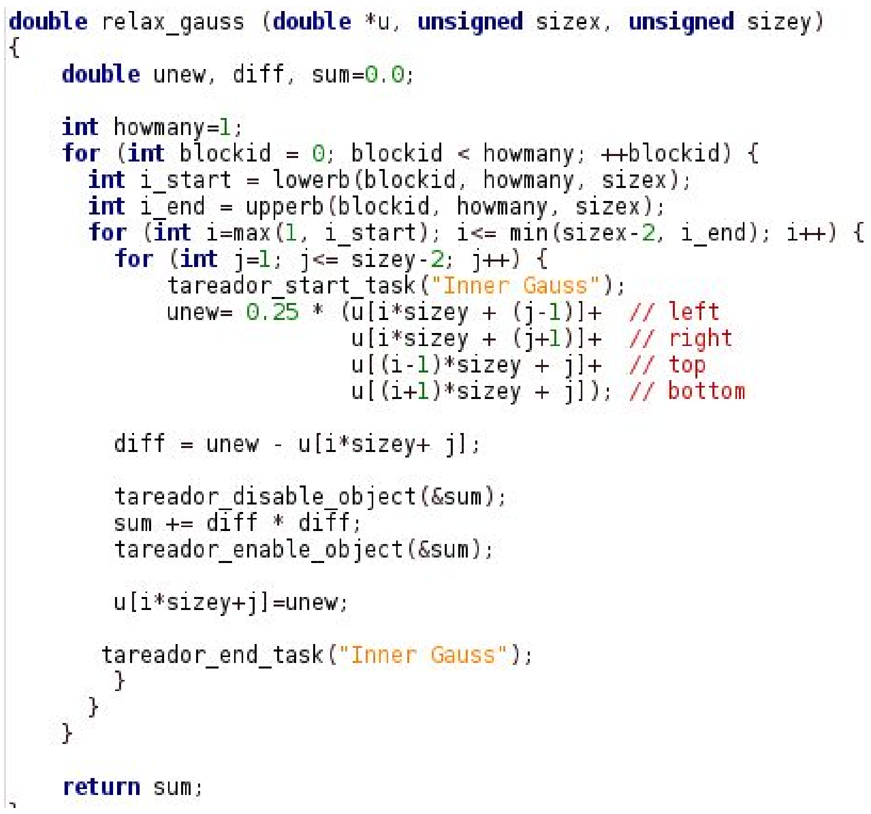
In order to parallelize both solvers, the strategy used has been as finest grain task as possible.

Now, we modify the code including the needed tareador clauses as following:



```c
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++) {
            tareador_start_task("Inner Copy Mat");
            v[ i*sizey+j ] = u[ i*sizey+j];
            tareador_end_task("Inner Copy Mat");
        }
}

/*
 * Blocked Jacobi solver: one iteration step
 */
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("Inner 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];
            tareador_disable_object(&sum);
            sum += diff * diff;
            tareador_enable_object(&sum);

            tareador_end_task("Inner Jacobi");
        }
      }
    }
    return sum;
}
```

Figure 5, code of jacobi’s part of solver-tareador.c



```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("Inner 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];

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

        u[i*sizey+j]=unew;

       tareador_end_task("Inner Gauss");
        }
      }
    }

    return sum;
```

Figure 6, code of gauss's part of solver-tareador.c

With the previous task decomposition, using Tareador is possible to check the dependency graph obtaining by using both algorithms, the following for the Jacobi and copy_mat:

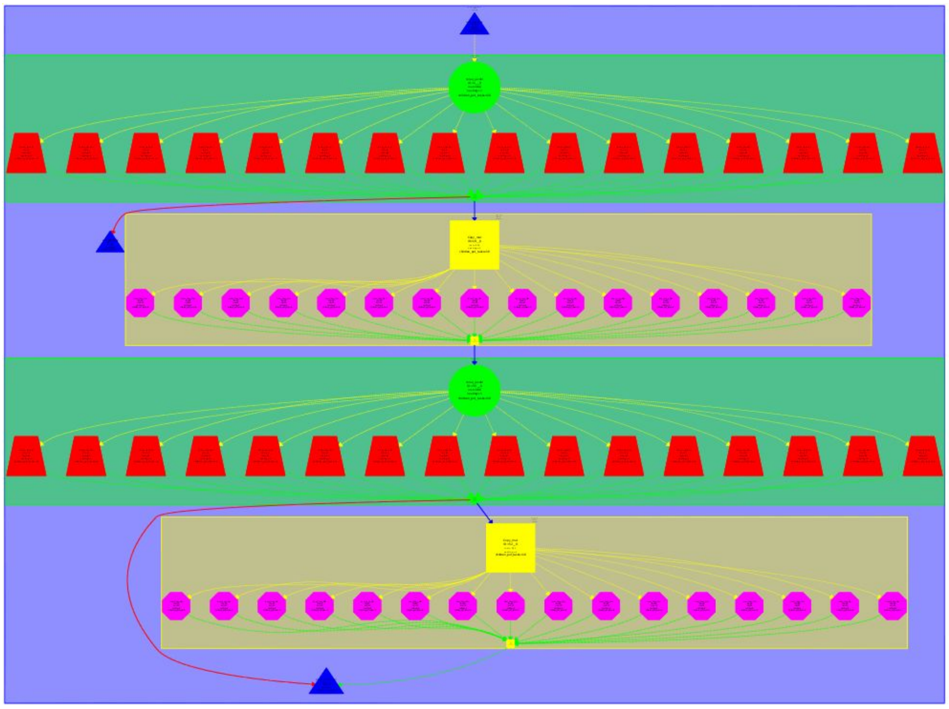
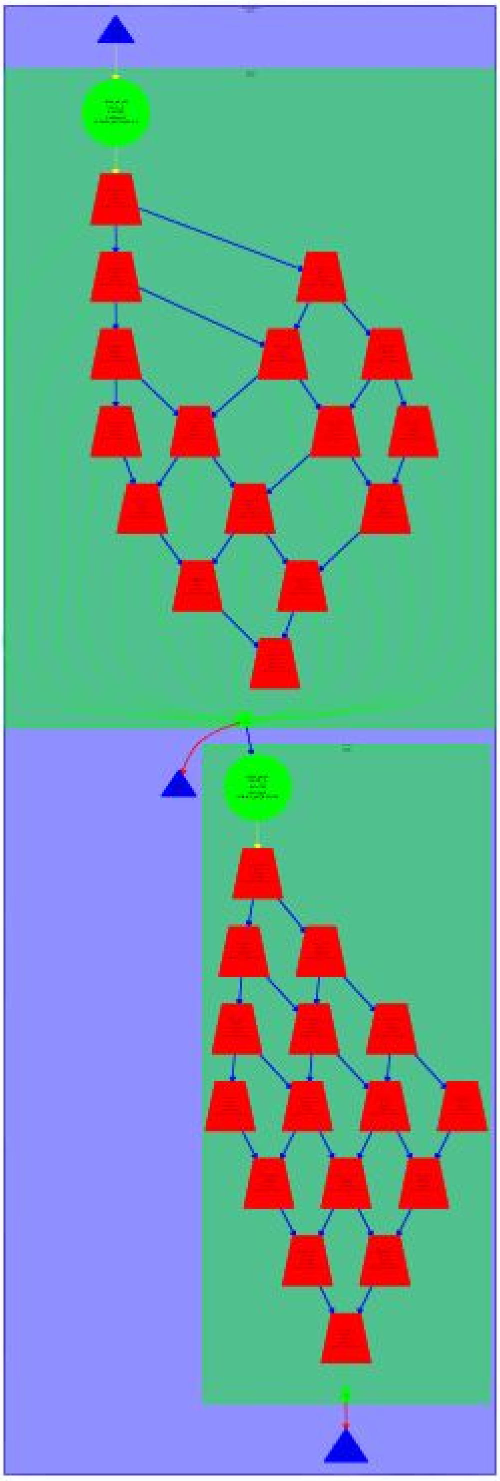


Figure 7, task graph of Jacobi's part

And the following for the Gauss-Seidel decomposition:



We can observe that both codes are sharing the variable sum between threads to store partial results, which has forced us to protect this variable in order to prevent data races. In OpenMP, it's needed to use a critical pragma combined with a shared variable sum, the pragma reduction, the atomic pragma instead of the critical, etc.
However, in tareador, we choose the pragma reduction which stores the partial result in a vector after the parallel region makes the sum of the partial result of each thread.

In addition, each iteration has a dependency with the neighbourhood cells.

Figure 8, task graph of Gauss-Seidel's part.

2.OpenMP parallelization and execution analysis: Jacobi

In this section of the deliverable, we are going to parallelize the sequential code of Jacobi following a geometric data decomposition. In this part, we can't make use of #pragma omp for or the combined #pragma omp parallel for.

Understanding the C macros defined in heat.h, we can conclude that we have to use the block data decomposition which we saw in class. The picture of this decomposition is the following one: (each processor executes a block of memory which means several elements of vector u)

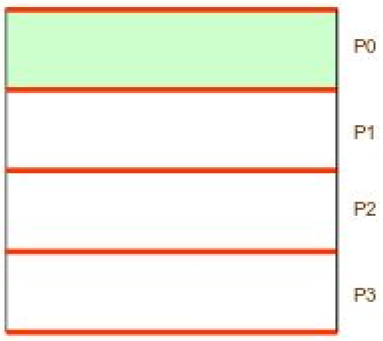
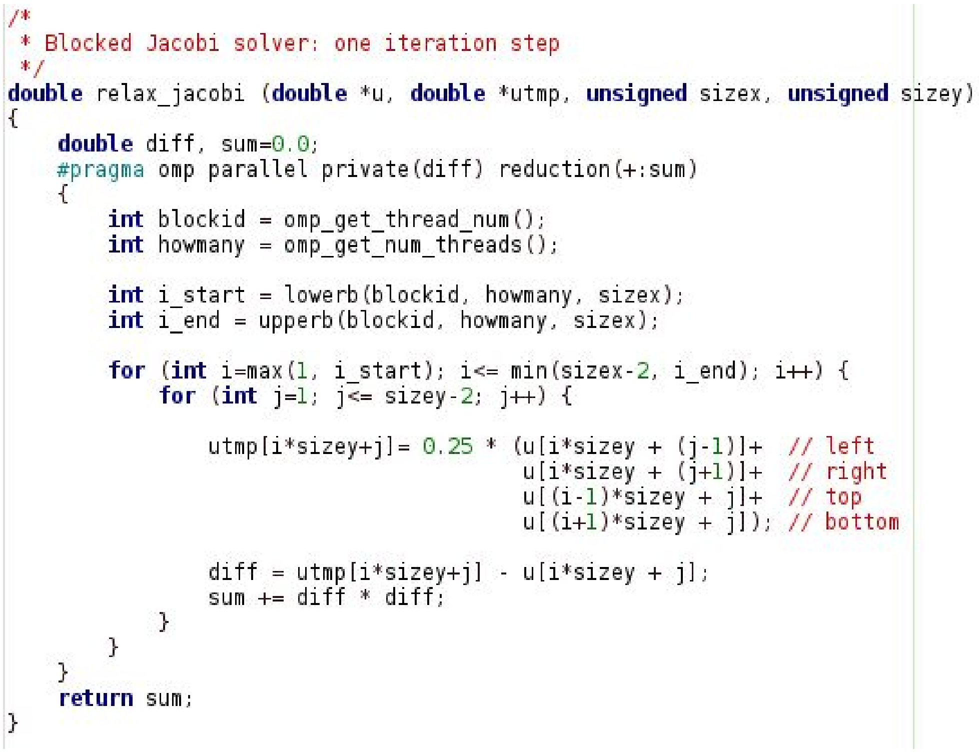


Figure 9, picture of block data decomposition

Using the macros provided, we parallelize the code to achieve the data decomposition in the following way: (verified that the generated image is the same as the sequential version)



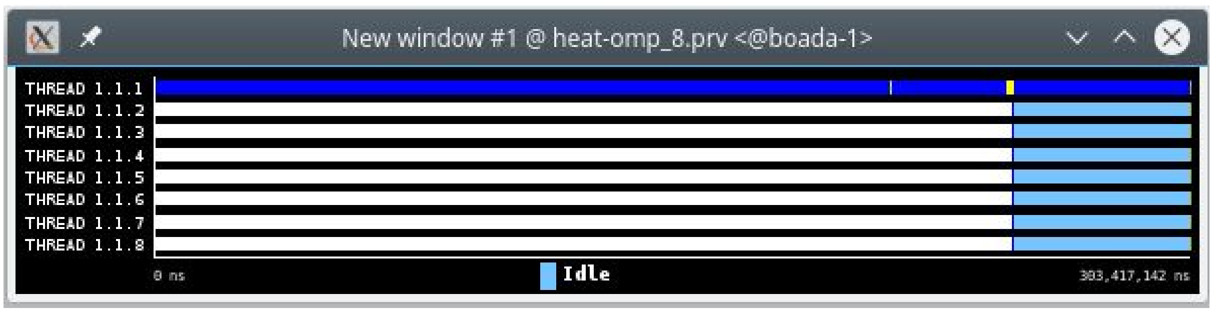
```
/*
 * Blocked Jacobi solver: one iteration step
 */
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;
    #pragma omp parallel private(diff) reduction(+:sum)
    {
        int blockid = omp_get_thread_num();
        int howmany = omp_get_num_threads();

        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 10, relax_jacobi function parallelized

The code of the relax_jacbi function needs a private clause because it's value is stored at every iteration, the reduction clause is needed because for every iteration we are incrementing that value, so we need to do reduction (+:sum) in order to protect the value of sum and to ensure that the variable is properly updated.

Moreover, we take the value of block id from omp_get_thread_num() and the howmany value from omp_get_num_thread() in order to do the data decomposition in outer loop between the available threads.

Using the wxparaver tool, we obtain the following timelines: (emphasize that we have to specify the Jacobi algorithm in .dat file instead of Gauss)



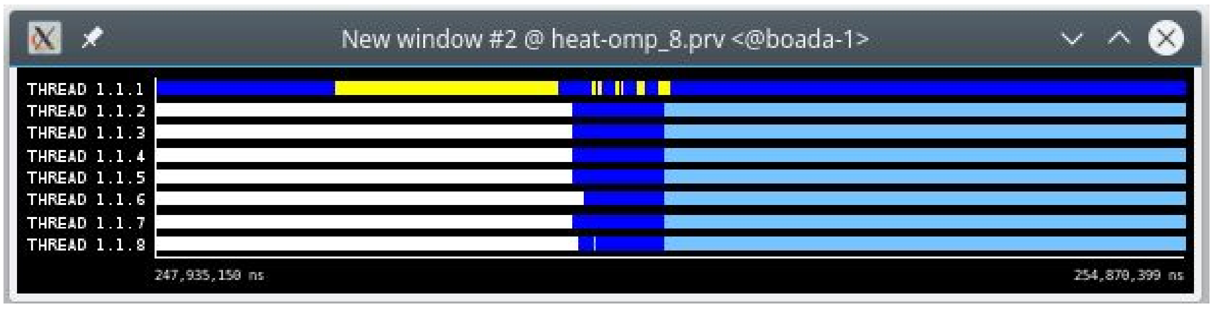
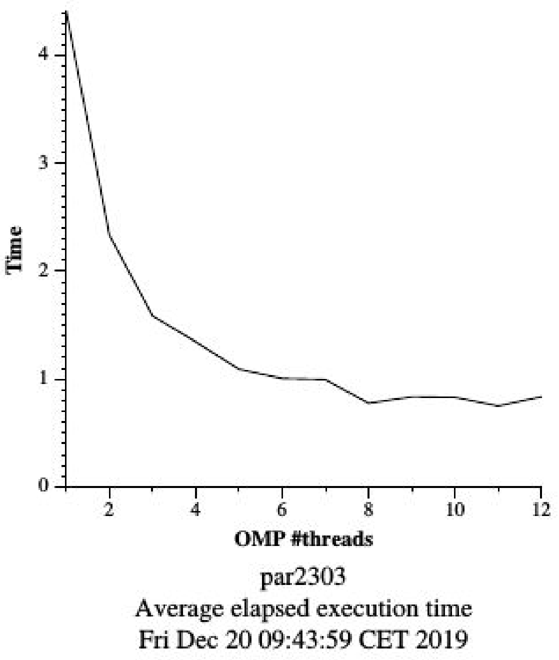


Figure 11, timeline of the heat-omp.c with the previous Jacobi's parallelization.

Figure 12, timeline of the heat-omp.c with the previous Jacobi's parallelization with a zoom at the parallelized zone.

In figure 12, we can observe that the parallelization part is only done at that zone where relax_jacobi() is executed and the available threads are dividing the work. (blue zone)

Once we are satisfied with the parallel behaviour observed, we decide to analyze the scalability of the parallelization. Therefore, we use the submit-strong-omp.sh script in order to obtain the following scalability plots:



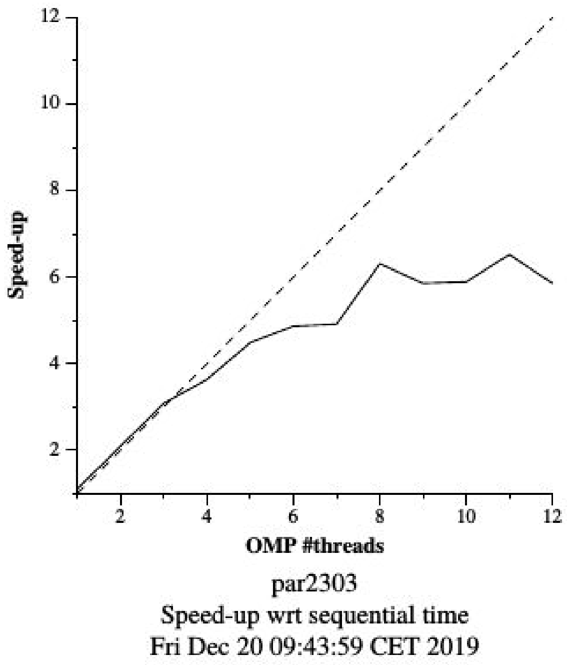
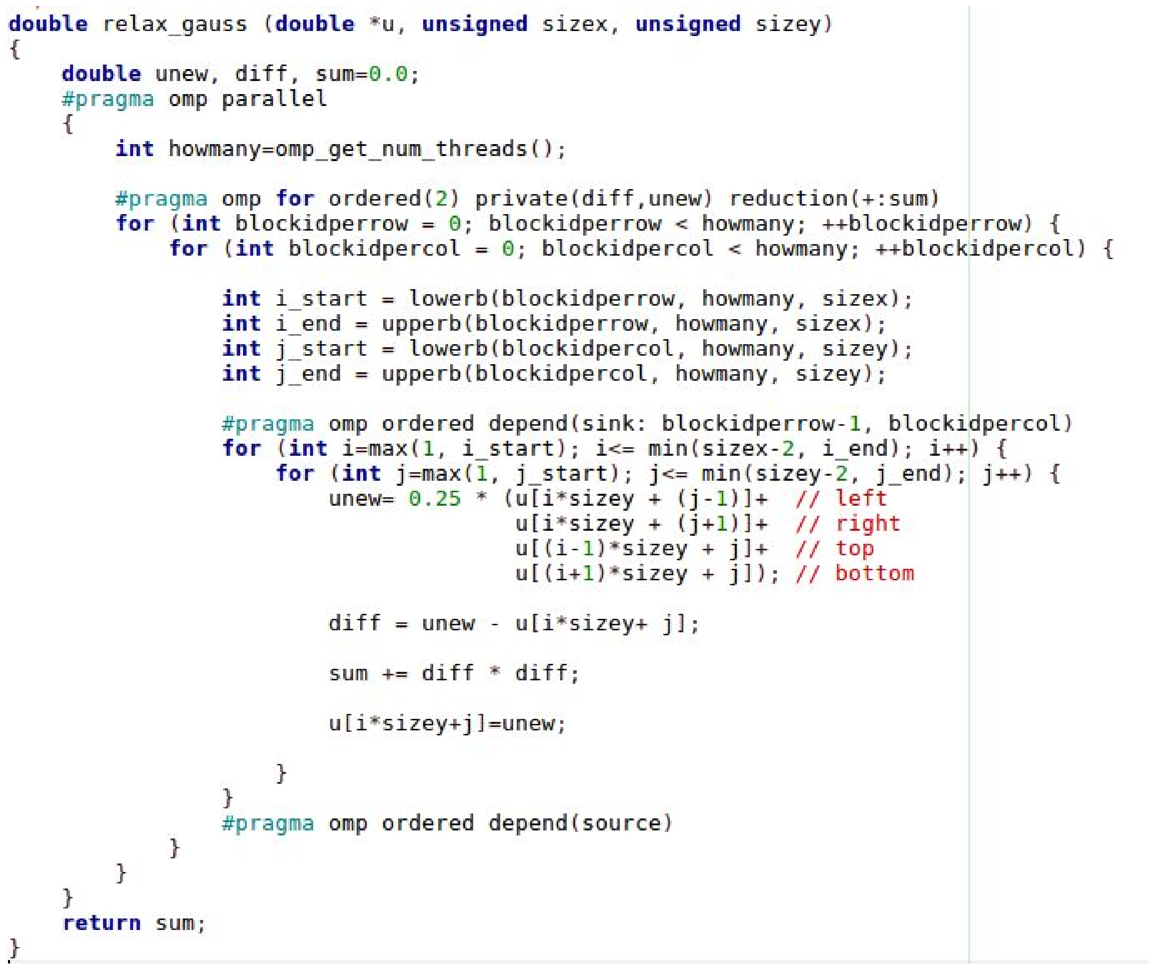


Figure 13, scalability plots of jacobi version

3.OpenMP parallelization and execution analysis: Gauss-Seidel

In this section of the deliverable, we are going to parallelize the Gauss-Seidel solver using #pragma omp for and its ordered clause. In this part, it is important to decide how we will synchronize the parallel execution of the rows assigned to each processor in order to guarantee the dependences which we detected with Tareador in the first section.

Then, the code would be the following one:



```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;
    #pragma omp parallel
    {
        int howmany=omp_get_num_threads();

        #pragma omp for ordered(2) private(diff,unew) reduction(+:sum)
        for (int blockidperrow = 0; blockidperrow < howmany; ++blockidperrow) {
            for (int blockidpercol = 0; blockidpercol < howmany; ++blockidpercol) {

                int i_start = lowerb(blockidperrow, howmany, sizex);
                int i_end = upperb(blockidperrow, howmany, sizex);
                int j_start = lowerb(blockidpercol, howmany, sizey);
                int j_end = upperb(blockidpercol, howmany, sizey);

                #pragma omp ordered depend(sink: blockidperrow-1, blockidpercol)
                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 14, parallelization of the relax_gauss function

As we can observe here, the code of the parallelization of the relax_gauss function we need to add the ordered(2) because of the two nested loops, and the same as the previous jacobi's function (the private and reduction clause).

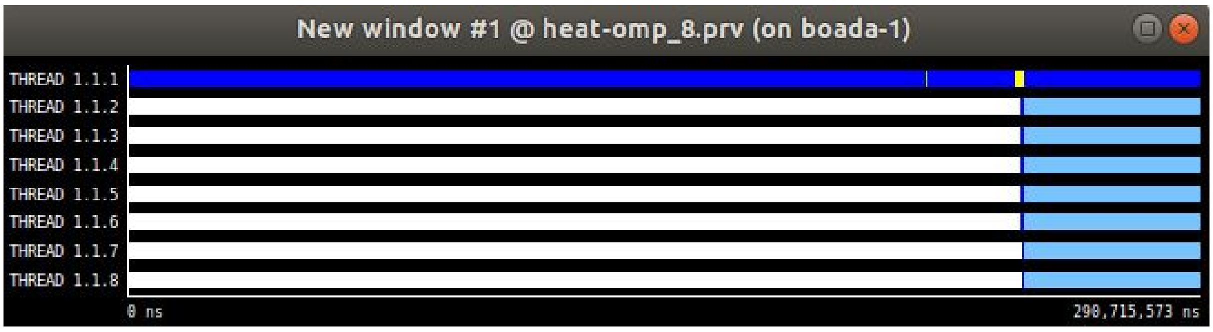
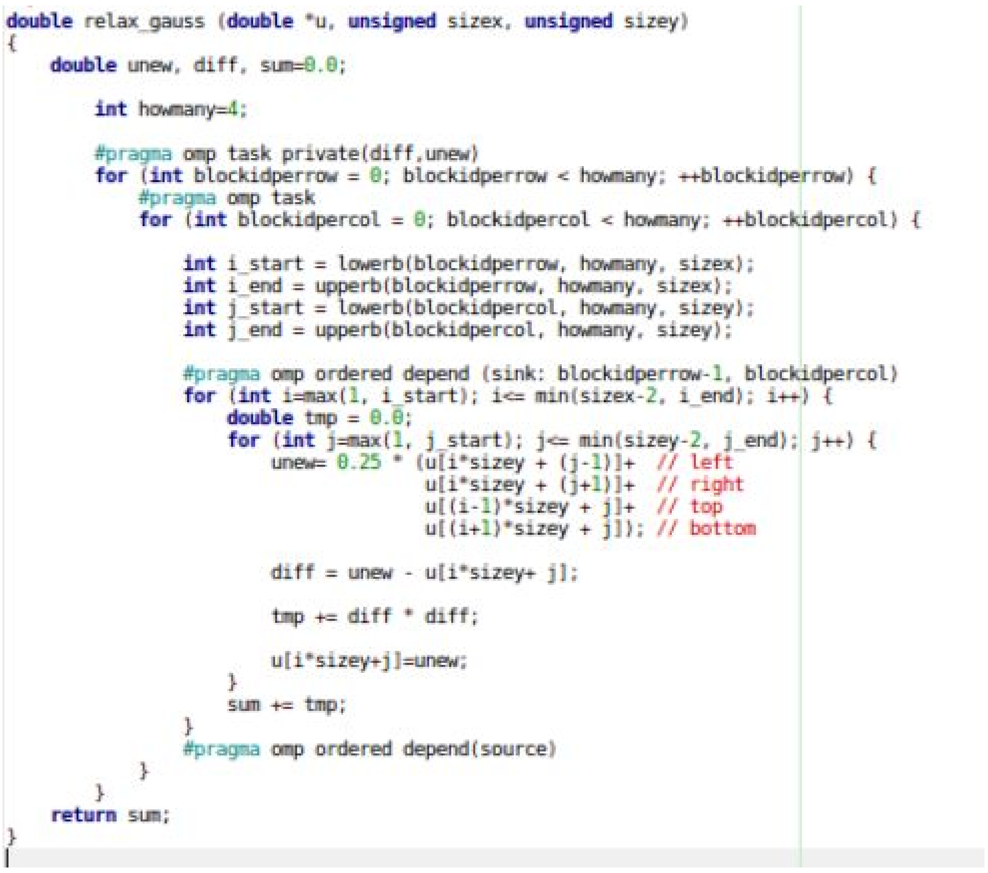


Figure 15, timeline of the code with the relaxed_gauss function parallelized

4.Optional

In this final section of the deliverable, we are going to parallelize the Gauss-Seidel solver using #pragma omp task instead of #pragma omp parallel. The code would be the following one:



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

        int howmany=4;

        #pragma omp task private(diff,unew)
        for (int blockidperrow = 0; blockidperrow < howmany; ++blockidperrow) {
            #pragma omp task
            for (int blockidpercol = 0; blockidpercol < howmany; ++blockidpercol) {

                int i_start = lowerb(blockidperrow, howmany, sizex);
                int i_end = upperb(blockidperrow, howmany, sizex);
                int j_start = lowerb(blockidpercol, howmany, sizey);
                int j_end = upperb(blockidpercol, howmany, sizey);

                #pragma omp ordered depend (sink: blockidperrow-1, blockidpercol)
                for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
                    double tmp = 0.0;
                    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];

                        tmp += diff * diff;

                        u[i*sizey+j]=unew;
                    }
                    sum += tmp;
                }
                #pragma omp ordered depend(source)
            }
        }
    return sum;
}
```

Figure 16, relaxed_gauss with task and depend

In order to maintain the previous pragma reduction which stores the partial result in a vector after the parallel region makes the sum of the partial result of each thread, we made a tmp variable where it does the same as the reduction manually.
We also put the pragma omp task at the beginning and for every iteration a new task is created, which makes the same as the pragma omp parallel clause.
The depend clauses are maintained.

But the solution with the tmp variable produces false sharing. One method to avoid this is padding.

5.Conclusions

The aim of this laboratory deliverable has been to make use of the different data decomposition strategies, which we have seen in theory class, in a practical example using Jacobi and Gauss-Seidel algorithms.

Firstly, we have used the tareador tool in order to see a finer-grain decomposition and their dependences with the task graphs obtained for each solver. For it, we included the creation of tasks inside the inner loop and we protected the variable sum to prevent data race.

Secondly, to parallelize our code, we had to use the task graphs obtained in the first part of the deliverable in order to parallelize our parallel code accurately. In Jacobi, we are forced to not use parallel for, so we used a block data decomposition which we saw in theory class. We obtain the same image as the sequentially version, but improving the performance of the code.

In Gauss-Seidel, we could use parallel for and ordered to parallelize the code using a data decomposition. Here, we have had some problems when we had to obtain the timelines and the scalability graphs.

To sum up, we think that we have learned and have put it into practice interesting things about data decomposition which we hadn't used it in a real situation, only in theory class. Moreover, we can conclude that in this subject we have learned the basic concepts of OpenMP and parallelism thanks for paraver and tareador tools. These laboratory sessions have been a good experience and have been worth it.