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

Group 2208

Course 2020/2021 Q1

Arnau Lamiel
Oscar Cañabate

arnau.lamiel@estudiantat.upc.edu
oscar.canabate@estudiantat.upc.edu

Index

-

1. Introduction

Since the start of PAR, we have studied two different solvers to parallelize, Jacobi and Gauss-Seidel solver. In this work we will use it to generate heat diffusion images. We will study how they generate these images and the main differences between them, and then we will interpret them.

The main purpose of the work is to study how we can parallelize both solvers and know what is the best strategy. We have to take into consideration that each of these algorithms have different data dependencies, that we will have to take care of in order to parallelize the programs correctly, especially in Gauss-Seidel, which has data dependencies from previous executions.

This work will be divided, as usual, in different parts. We will use *tareador* in order to study both strategies and his graphs. And in the next section, we will use *paraver*, to know how the algorithm is working.

There are differences between this and previouses works for PAR, in these work we will use data decomposition instead of task decomposition, so parallel implementation will be different, but trying to achieve the same outcome with a different strategy.
This way we will implement some different solutions for each solver in order to find the best version and the best way to parallelize them correctly.

2. Parallelization Strategies

Now, we are going to explain the different strategies of this work, and for it we used *tareador* (code is on tareador.c), in order to see better the strategy.

On the one hand we have the **Jacobi solver**, in the Figure 1.

As we can see, the execution of Jacobi solver is practically sequential, that is because there are data dependencies between tasks.

We noticed that the variable that has been shared is *sum*, because all tasks need to update it. That means that there is a big dependency between executions.

In order to improve parallelism, we have to use a reduction omp clause on the variable sum. For this, we declare sum as local variable in each task, and then adding all of them at end of the execution.

The clause is: *pragma omp parallel for reduction(+:sum)*

In tareador we cannot use omp clauses, so we decided to disable the sum variable with this clause: *tareador_disable_object(&sum)*; to simulate in tareador the strategy. We can see it at Figure 2.

Now, graph is notorious better parallelized than before because we solved dependencies, which was the main problem.

In figure 3 we can see how are now the dependencies between tasks.

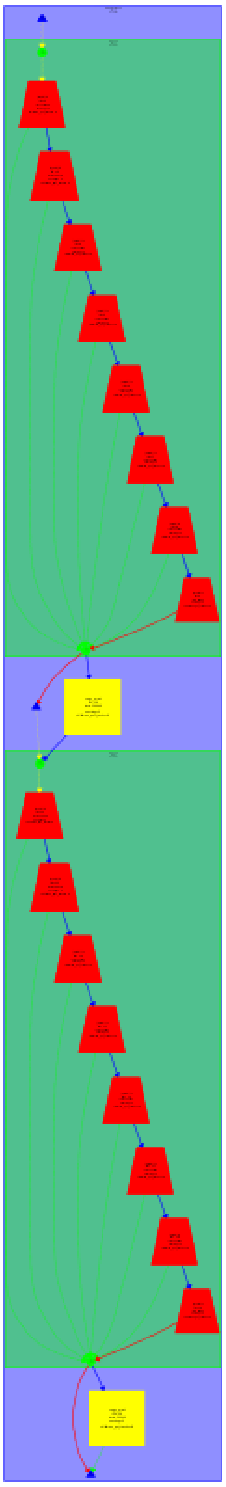


Figure 1: Inner loop tasks in Jacobi solver with Tareador

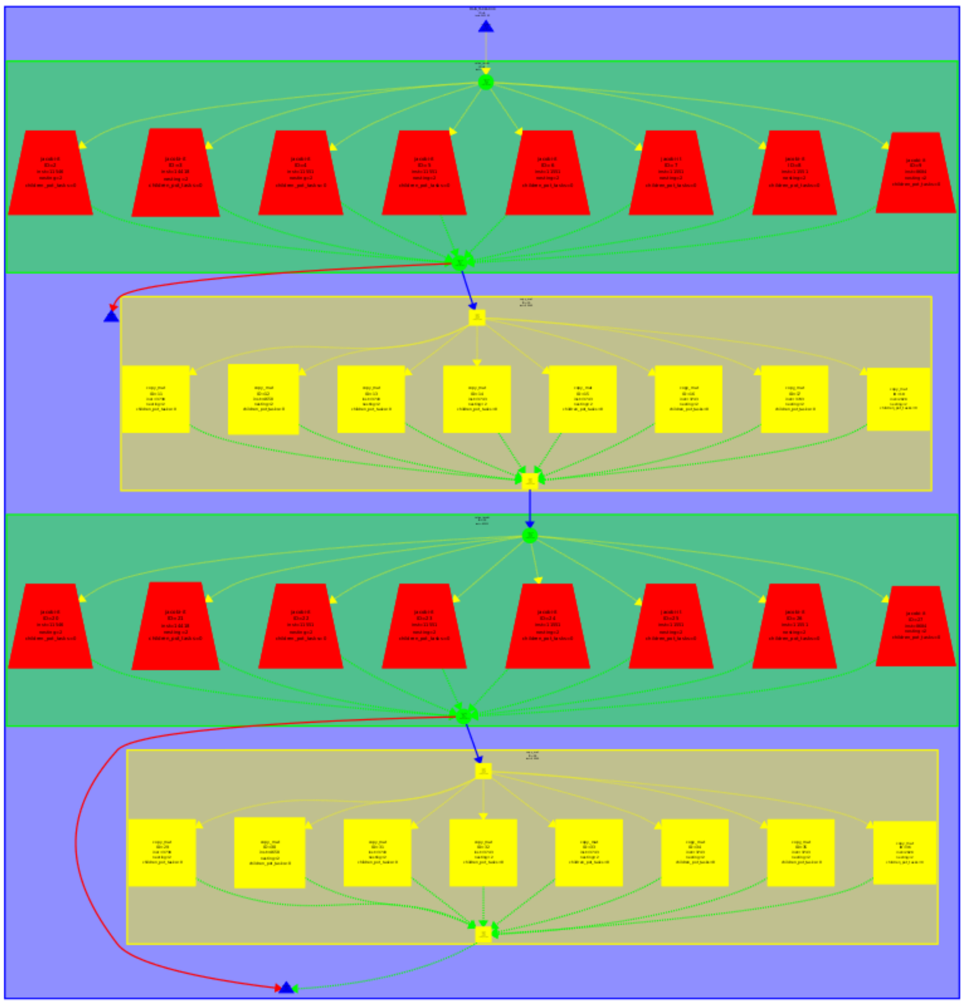


Figure 2: Inner loop tasks in Jacobi solver with Tareador disabling sum variable

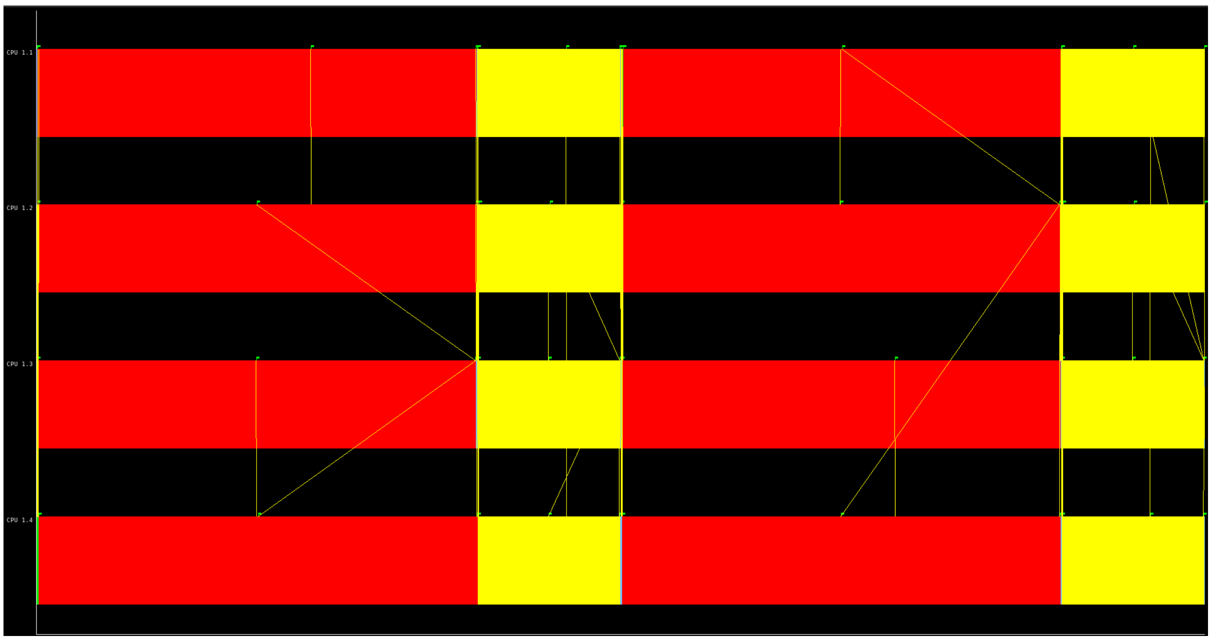


Figure 3: Task dependency on Jacobi solver

On the other hand, we are going to study the **Gauss-Seidel** strategy. In the figure 4, we can see that the main difference between the Gauss and the Jacaobi is that it does not use a function to copy the matrix, as Jacobi. This version calculates the value and then save this auxiliar matrix inside the same matrix.

As in the previous case, we need to do reduction into the sum variable. In addition, in order to increase the parallelism, we noticed that there is a dependency with the upper and the left variables of execution inside the matrix. So we need to fix it too.

In *Tareador* we cannot parallelize that, just the sum variable, we can see it at Figure 5.
There is an increased parallelism, but is not enough, as we said before, because it is still a little sequential.

In a future session, we will fully parallelize it, due to the fact that the innermost loop inside the relax_gauss function can be perfectly parallelized using the pragma order.

To end with this section, we can see at Figure 6 the task dependency of the Gauss-Seidel algorithm. We can see that parallelism still can be improved.

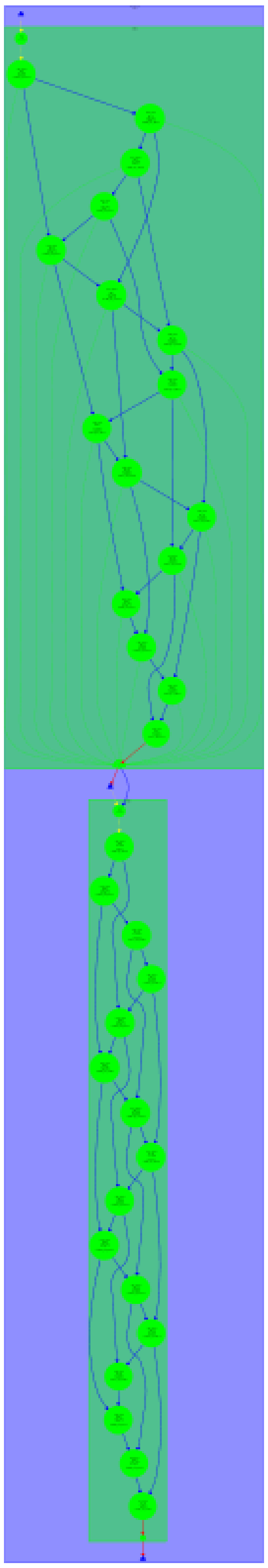


Figure 4. Inner loop tasks in Gauss-Seidel solver with Tareador

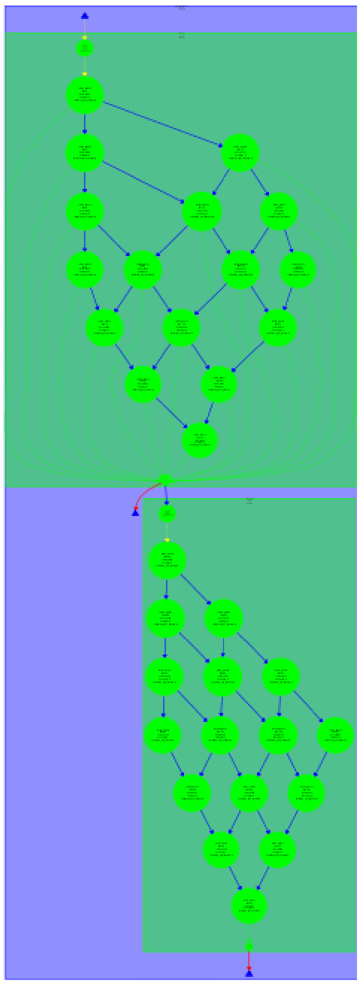


Figure 5: *Inner loop tasks in Gauss-Seidel solver with Tareador disabling sum variable*

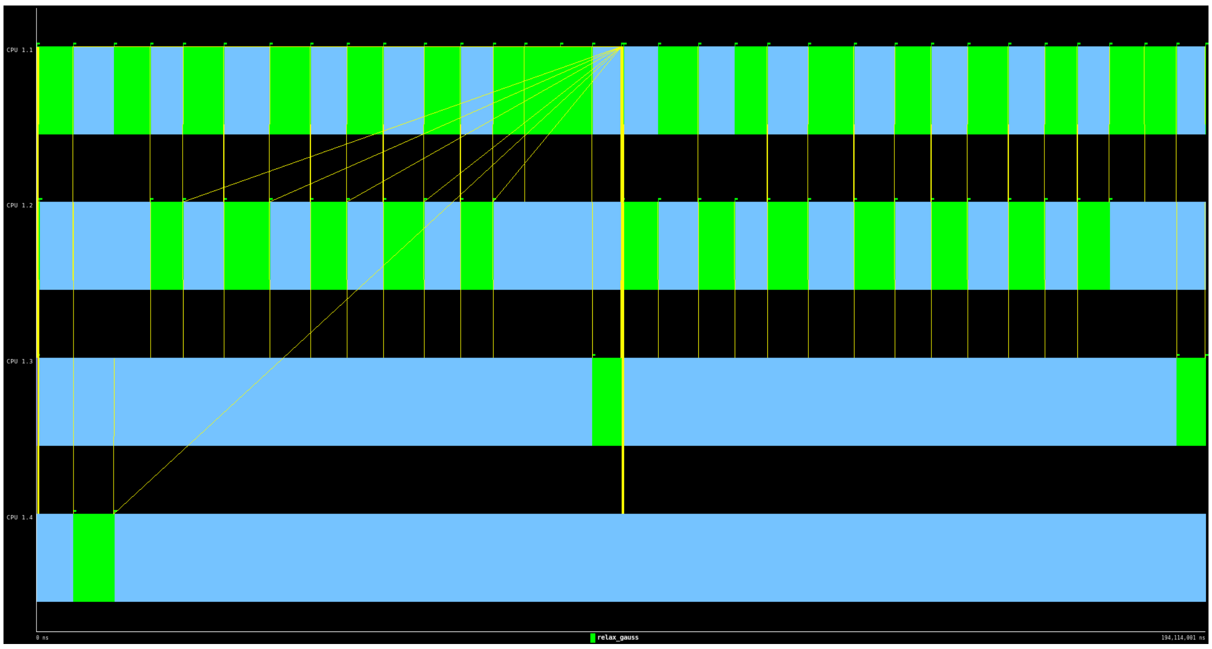


Figure 6: Task dependency on Gauss-Seidel solver

3. Performance analysis

In this section, we will analyze the performance of both strategies, in order to know what is the best parallelization strategy for both algorithms, and which is the best parallelization strategy for the problem.

Jacobi

This parallelization strategy works on two matrix. We used a data decomposition strategy. In Figure 8 we can see the code of this section (all the code is on solver-omp.c) . We executed the same code in all threads but only assigned them to a specific part of the loop.
In Figure 8 we can see the private and reduction clauses to parallelize the code.
In Figure 7, in order to understand the strategy, we have a table of the data decomposition generated when the number of threads is four, where each block has the same size.

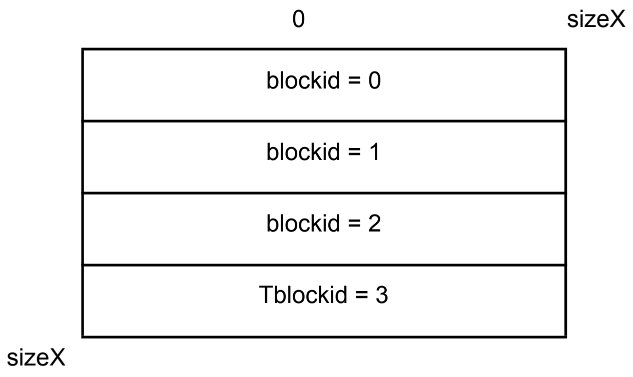
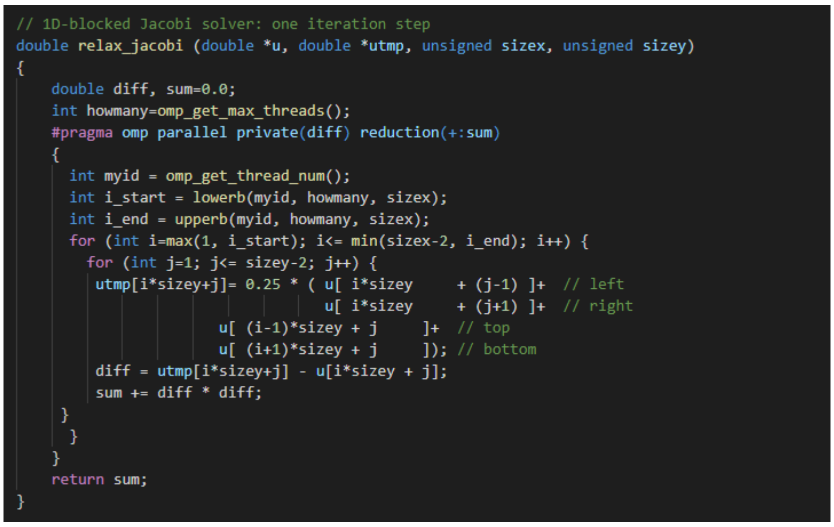


Figure 7. Table of data decomposition



```
// 1D-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 myid = omp_get_thread_num();
      int i_start = lowerb(myid, howmany, sizex);
      int i_end = upperb(myid, 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 8: Code for jacobi solver

Now, we will analyse the *paraver* plots. In Figures 9 and 10, we can see the paraver execution of the Jacobi Solver.

The data input is very small and the initialization of the vectors and variables among others is larger than the parallelizable part of the code, making us seem like using 8 threads is not as benefitial as we would initially think. In figure 9, we can see how the sequential part of the code is much bigger than the parallelizable part. In figure 10, we have zoomed in the parallelizable part, and we can see that it is working correctly, with actual parallelization, which is making the execution faster.

The sequential part of the execution is caused

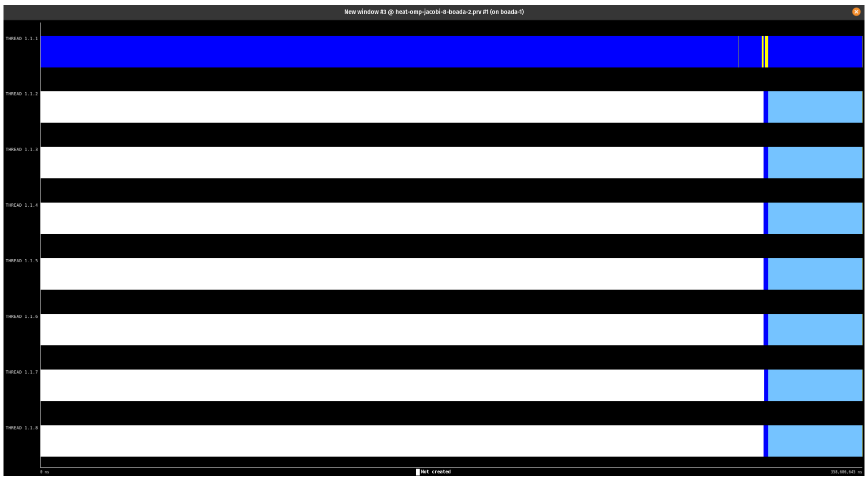


Figure 9. Paraver execution of the Jacobi Solver with 8 threads

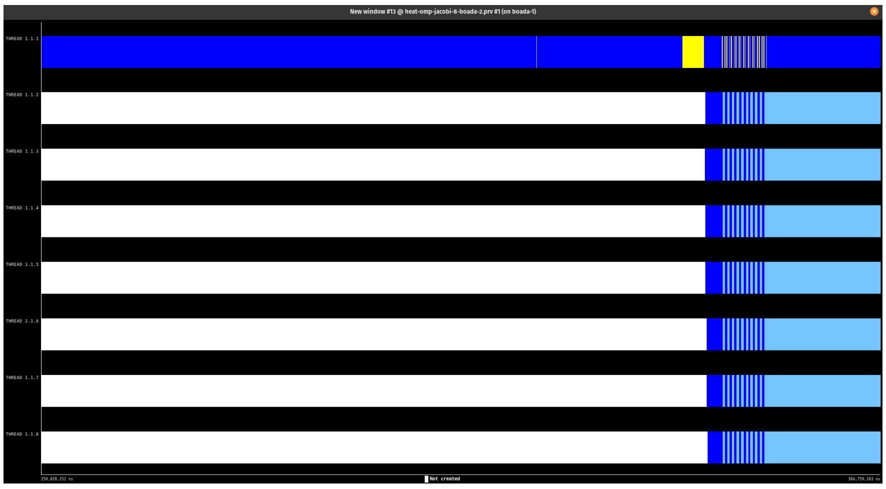
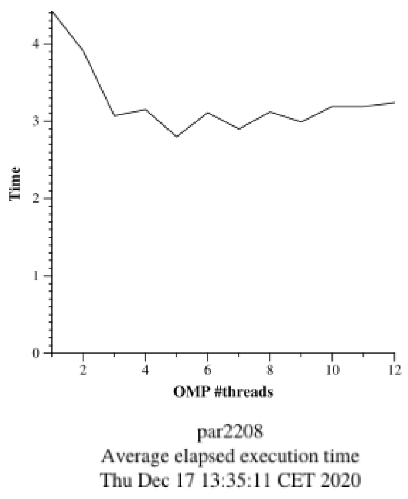


Figure 10. Zoom in paraver execution of the Jacobi Solver with 8 threads

Finally, we will analyze the speed-up plots of the Jacobi Solver execution. As we can see in Figure 11, there is a problem with our plots, we tried several times to generate them but they always generate bad. In theory, the plots should be better, because the speedup should be positive and almost perfect, because when we use two matrix, is easier to parallelize the code.
In our plots, speedup is decreasing, that has no sense. Time execution is not so bad because it decreases using more processors, but is not good because the decreation is not notorious.



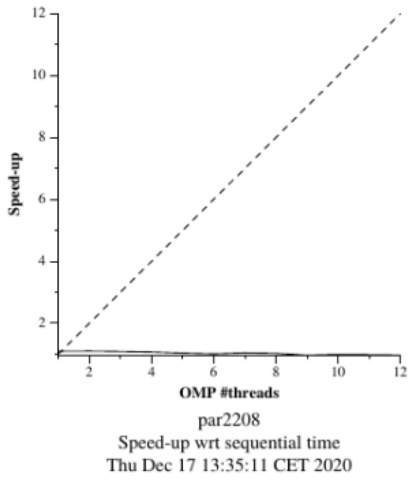
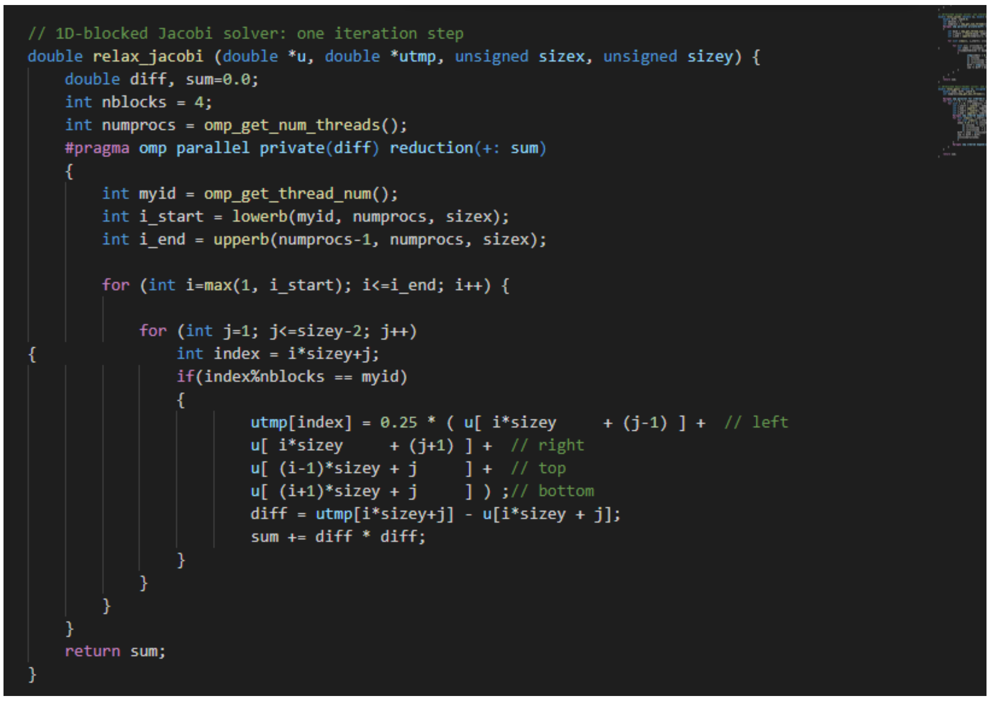


Figure 11. Speed up and execution time plots for Jacobi solver

For the Optional 1, we decided to implement the next code (Figure O1) in order to implement an alternative block-cyclic by column data decomposition.



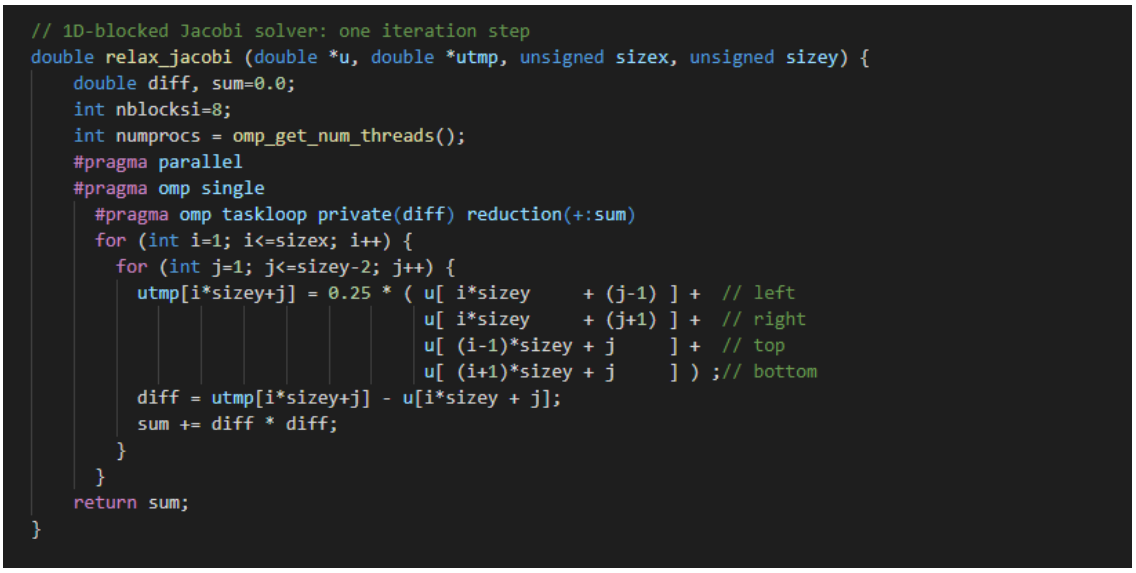
```c
// 1D-blocked Jacobi solver: one iteration step
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey) {
    double diff, sum=0.0;
    int nblocks = 4;
    int numprocs = omp_get_num_threads();
    #pragma omp parallel private(diff) reduction(+: sum)
    {
        int myid = omp_get_thread_num();
        int i_start = lowerb(myid, numprocs, sizex);
        int i_end = upperb(numprocs-1, numprocs, sizex);

        for (int i=max(1, i_start); i<=i_end; i++) {

            for (int j=1; j<=sizey-2; j++)
{               int index = i*sizey+j;
                if(index%nblocks == myid)
                {
                        utmp[index] = 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 O1. Code for the Optional 1

For the Optional 2, we decided to implement the next code (Figure O2) in order to use explicit tasks following an iterative task decomposition. There is not a better performance, so we do not show *paraver* execution.



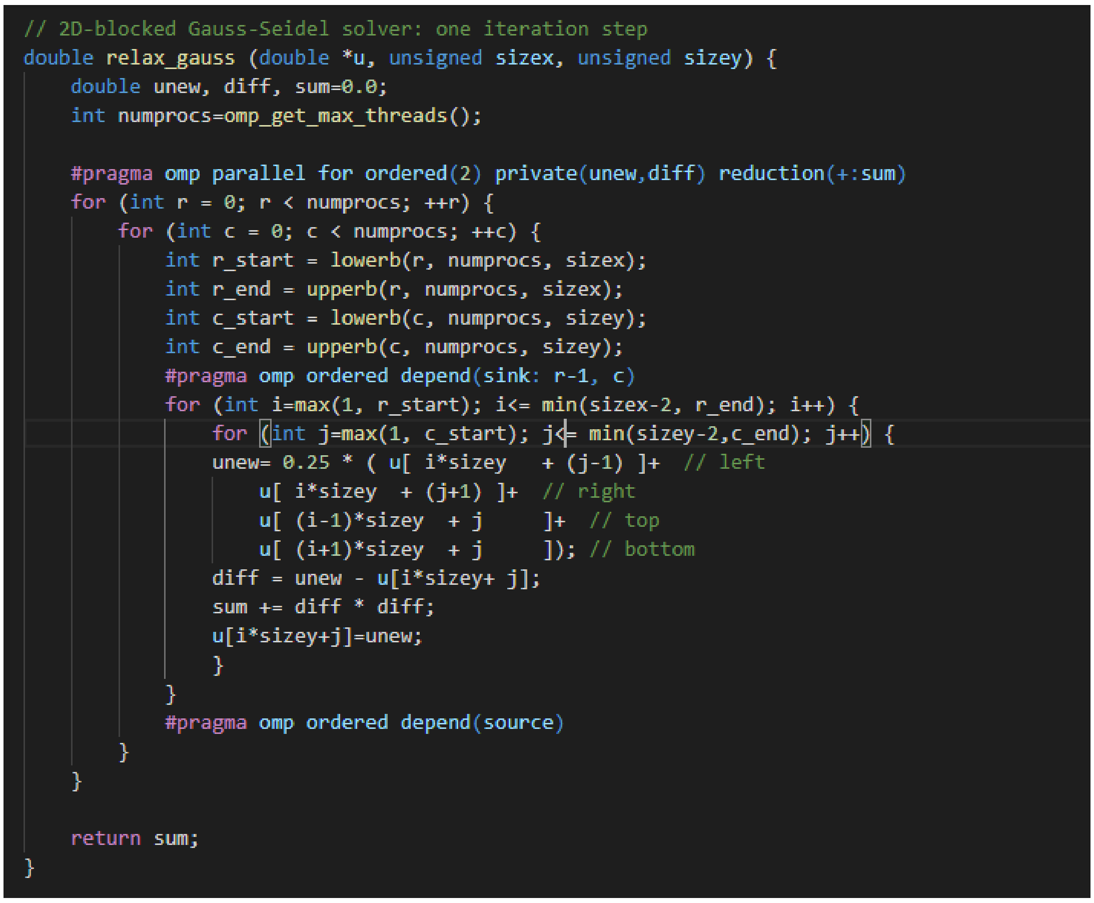
```c
// 1D-blocked Jacobi solver: one iteration step
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey) {
    double diff, sum=0.0;
    int nblocksi=8;
    int numprocs = omp_get_num_threads();
    #pragma parallel
    #pragma omp single
      #pragma omp taskloop private(diff) reduction(+:sum)
      for (int i=1; i<=sizex; 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 O2. Code for the Optional 2

Gauss-Seidel

To do the parallelization of this solver, we used the *for ordered* clause.

As we can see in the Figure 12, we used *#pragma omp parallel for ordered(2) private(unew,diff) reduction(+:sum)* clause to parallelize the code. All the code is in solver-omp.c.



```c
// 2D-blocked Gauss-Seidel solver: one iteration step
double relax_gauss (double *u, unsigned sizex, unsigned sizey) {
    double unew, diff, sum=0.0;
    int numprocs=omp_get_max_threads();

    #pragma omp parallel for ordered(2) private(unew,diff) reduction(+:sum)
    for (int r = 0; r < numprocs; ++r) {
        for (int c = 0; c < numprocs; ++c) {
            int r_start = lowerb(r, numprocs, sizex);
            int r_end = upperb(r, numprocs, sizex);
            int c_start = lowerb(c, numprocs, sizey);
            int c_end = upperb(c, numprocs, sizey);
            #pragma omp ordered depend(sink: r-1, c)
            for (int i=max(1, r_start); i<= min(sizex-2, r_end); i++) {
                for (int j=max(1, c_start); j<= min(sizey-2,c_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 12: Code for Gauss-Seidel solver

Now, we will analyse the execution on *paraver*. We can see the execution diagram in Figure 13 and 14. As in the previous part, we did a zoom to see better the parallelization.

We executed the code in 8 threads(Figure 13) and we had the same problem as in the previous part, data input is very small and the parallel part is not significant in the general execution.

If we zoom (Figure 14), we can see that all threads are balanced, and the parallel execution is good enough.

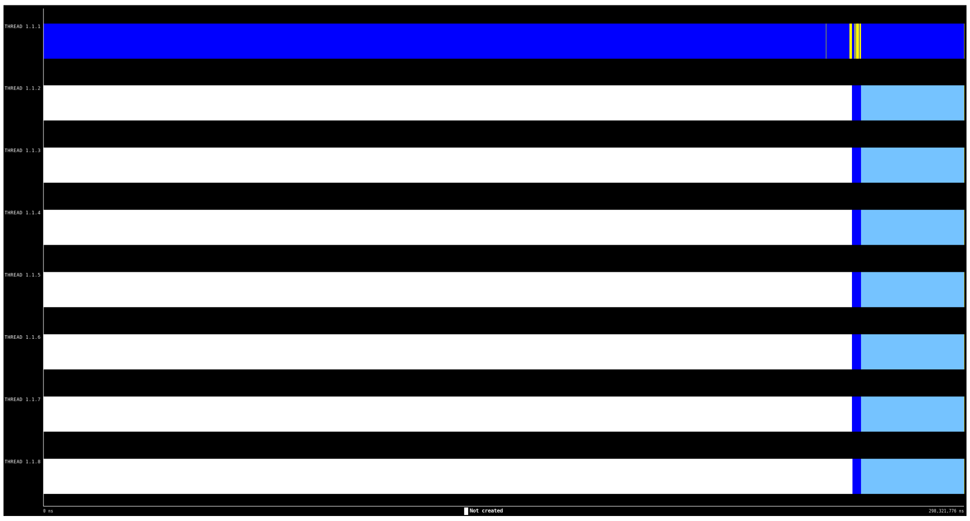


Figure 13: Paraver execution of the Gauss-Seidel solver with 8 threads

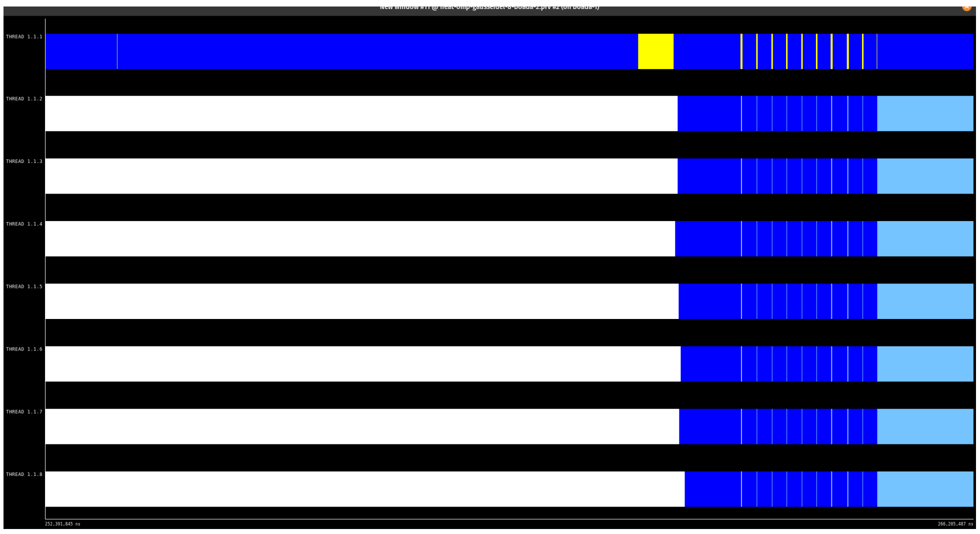


Figure 14: Zoom of Paraver execution of the Gauss-Seidel solver with 8 threads

Next, we will analyze the speed-up and execution time plots. We can see it at Figure 15.

As we can see, the execution time is reducing for a bigger number of threads, what is important and how we can expect initially. If we observe the speed-up plot, we can see that it is far from a perfect plot, and that is because of the difference between the synchronization and computation for each thread. But at least, the speed-up is increasing for a bigger number of threads, so that means that it is not so terrible.

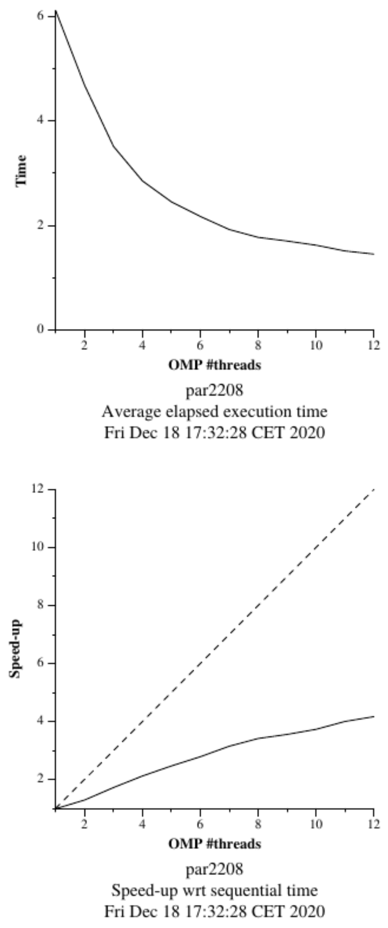
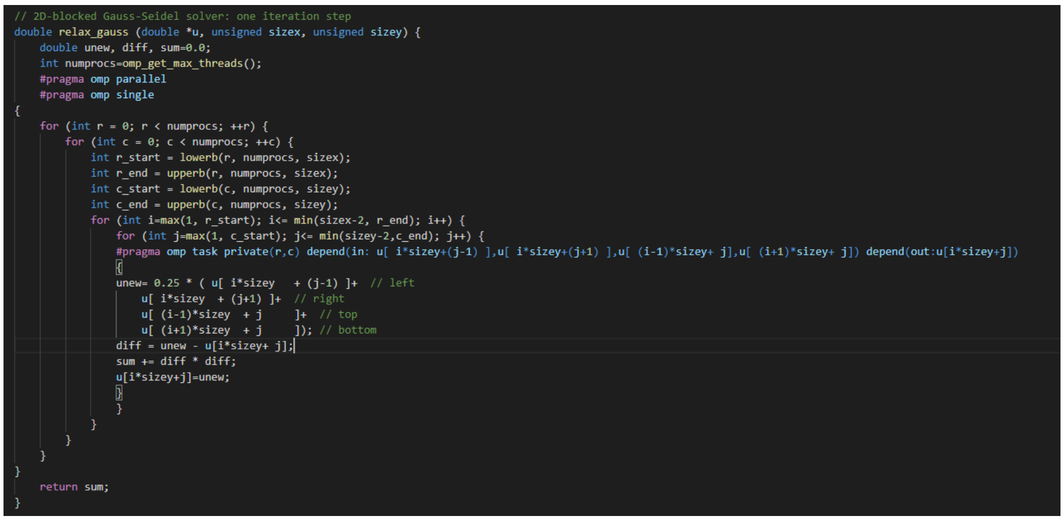


Figure 15. Speed up and execution time plots for Gauss-Seidel

For the Optional 3 we implemented the Gauss-Seidel solver using explicit tasks and following an iterative task decomposition. Code is on Figure O3



```
// 2D-blocked Gauss-Seidel solver: one iteration step
double relax_gauss (double *u, unsigned sizex, unsigned sizey) {
    double unew, diff, sum=0.0;
    int numprocs=omp_get_max_threads();
    #pragma omp parallel
    #pragma omp single
{
    for (int r = 0; r < numprocs; ++r) {
        for (int c = 0; c < numprocs; ++c) {
            int r_start = lowerb(r, numprocs, sizex);
            int r_end = upperb(r, numprocs, sizex);
            int c_start = lowerb(c, numprocs, sizey);
            int c_end = upperb(c, numprocs, sizey);
            for (int i=max(1, r_start); i<= min(sizex-2, r_end); i++) {
                for (int j=max(1, c_start); j<= min(sizey-2,c_end); j++) {
                #pragma omp task private(r,c) depend(in: u[ i*sizey+(j-1) ],u[ i*sizey+(j+1) ],u[ (i-1)*sizey+ j],u[ (i+1)*sizey+ j]) depend(out:u[i*sizey+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;
                }
                }
            }
        }
    }
}
    return sum;
}
```

Figure O3: Code for Optional 3

4. Conclusions

We studied the different versions of the heat diffusion algorithm and now we can conclude different facts.

First, we started analyzing the *tareador* graphs, to see if there is any dependency. This exercise makes us see that there are different dependencies and thinks how we can parallelize the code properly.

For the Jacobi solver, we used a data decomposition strategy. We can observe that we used a reduction on the sum variable and we privatized diff variable. We also distributed work between threads using howmany as a *omp_get_max_threads()* clause and id of thread as omp_get_thread_num() clause.
This version was easier to parallelize than the Gauss version, due to the fact that we didn’t have to parallelize the bounders of the Matrix.

For the Gauss-Seidel version, we cannot affirm that it is a better solution than the Jacobi, and it seems worse in a lot of cases. That is because the Gauss-Seidel algorithm has more data dependencies than the Jacobi one. It was expected because it uses just one Matrix, which is more difficult to parallelize than if we use two, like in the Jacobi solver.