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

Pablo Cebollada Hernández
Iván Díaz Ortega
Par1104
Q1 2019/2020
26/12/19

Index

Introduction

In this assignment we are going to work and analyze the parallelization of a code that simulates the heat diffusion on a solid body using two different solvers for the heat equation, one called Jacobi and another called Gauss-Seidel. First we are going to focus on the dependencies that may exist in both solvers using tareador, so we can get a clear idea of these dependencies and apply them in the parallelized code that we are going to produce. Then the first solver that we are going to get our hands with is the Jacobi solver and following it the Gauss-Seidel one. This parallelization is going to be acquired using, also, a proper data decomposition provided by the code.

1. Analysis of task granularities and dependencies

In order to find the dependencies that may exist with the original program we used the tareador to produce the graph in figure 1.1 which show us the most general dependencies and the amount of work that every task executes. We can see that the main dependencies are between the green task, which represents the execution of the solver for the heat equation, and the copy_mat method, which makes a copy of the matrix produced by the solver. As there are two different approaches to implement a solver for this heat equation we needed to know what kind of dependencies may exist if we parallelized the computation part of the two different solvers.

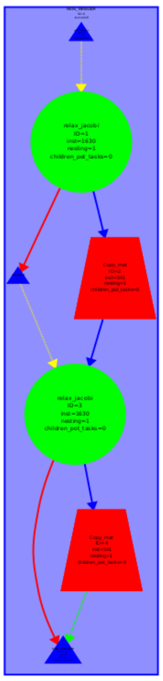
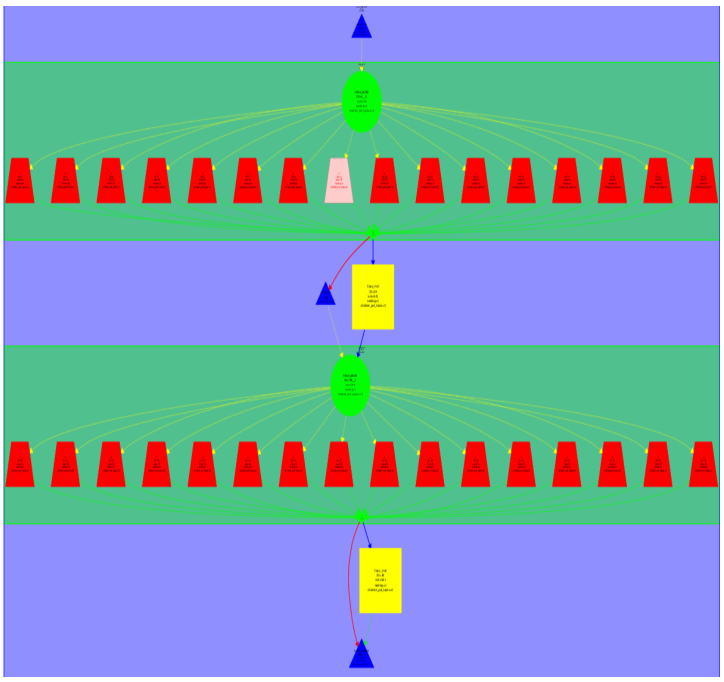
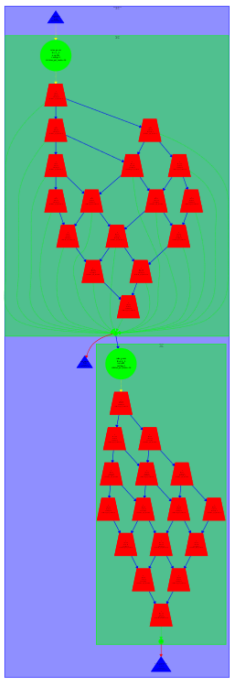
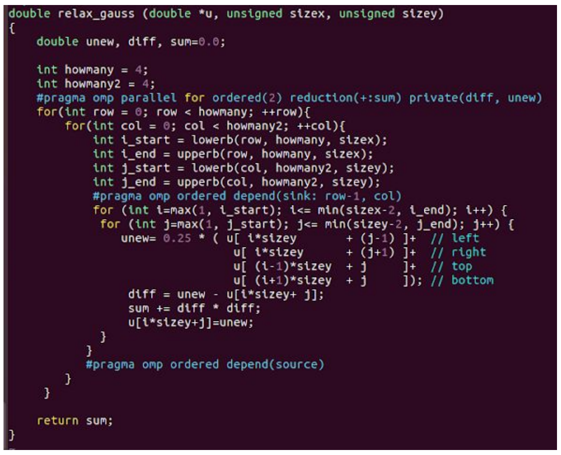
 relax_jacobi
ID=1
inst=1630
nesting=1
children_pot_tasks=0
Copy_mat
ID=2
inst=541
nesting=1
children_pot_tasks=0
relax_jacobi
ID=3
inst=1630
nesting=1
children_pot_tasks=0
Copy_mat
ID=4
inst=541
nesting=1
children_pot_tasks=0
		

Fig 1.1 Initial dependencies.	Fig 1.2 Jacobi version.	Fig 1.3 Gauss-Seidel version.

- Jacobi version: There are not any dependencies between tasks created, then we don’t need to protect the parallel code in any way, since the tasks doesn’t modify the original matrix rather they write the result in an auxiliary matrix and read the original values from the original matrix without modifying it.
- Gauss-Seidel: In this case there are dependencies between tasks generated. These dependencies are produced by the positions of the matrix directly above, below, left and right of the position that is being computed at that moment. Although there are four dependencies we only have to worry about the left and upper ones, since the dependencies shows that these two before have to be already computed but the other two don’t and by granting the first two dependencies we already grant the other

two. So two make sure that this is true we have to execute the iterations in order and making avery one of them wait for the upper and left positions to be already computed at the moment when they start their computation.



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

    int howmany = 4;
    int howmany2 = 4;
    #pragma omp parallel for ordered(2) reduction(+:sum) private(diff, unew)
    for(int row = 0; row < howmany; ++row){
        for(int col = 0; col < howmany2; ++col){
            int i_start = lowerb(row, howmany, sizex);
            int i_end = upperb(row, howmany, sizex);
            int j_start = lowerb(col, howmany2, sizey);
            int j_end = upperb(col, howmany2, sizey);
            #pragma omp ordered depend(sink: row-1, col)
            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;
}
```

Fig 1.4 Gauss-Siedel with pragma omp depend.

2. Parallelization of Jacobi with openMP ordered

To achieve our goal, we have opted for a block decomposition, in the Fig 2.1 it can be seen how every block is assigned to every threads.

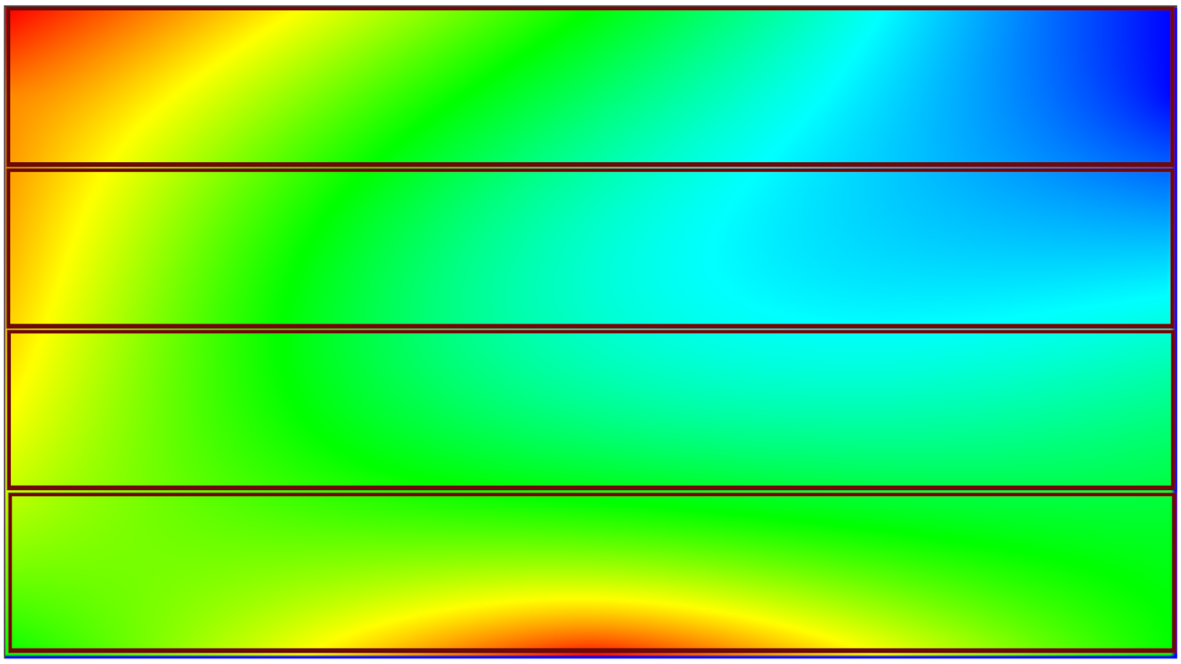
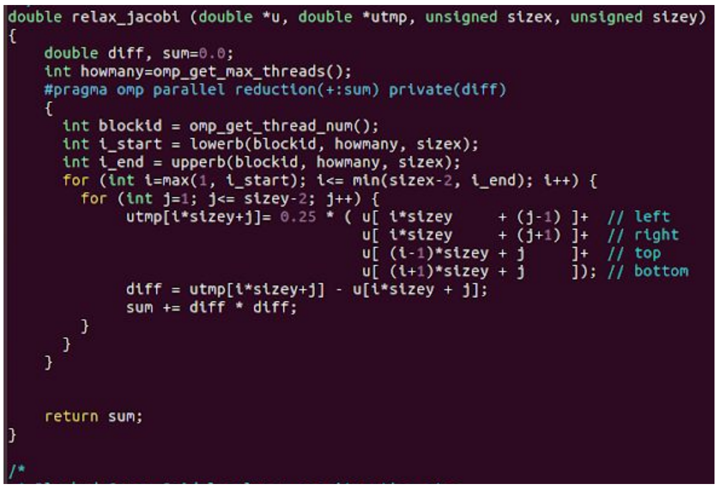


Fig 2.1 Block decomposition, one block for every thread.

We change some code to create the block decomposition as shown in Fig 2.2, for example, we needed to obtain how many threads exist so we call, outside the parallel zone, omp_get_max_threads() which return the max number of threads that we have, also inside the parallel code the function omp_get_thread_num() is called to obtain the id of the thread executing this part, we used the function lowerb to calculate where the thread must start, this function divides equally the matrix in number of threads parts and assigns each part to one thread, and upperb for the opposite, this function marks where the thread must finish. When we have all of this every thread executes the zone between his start and end.



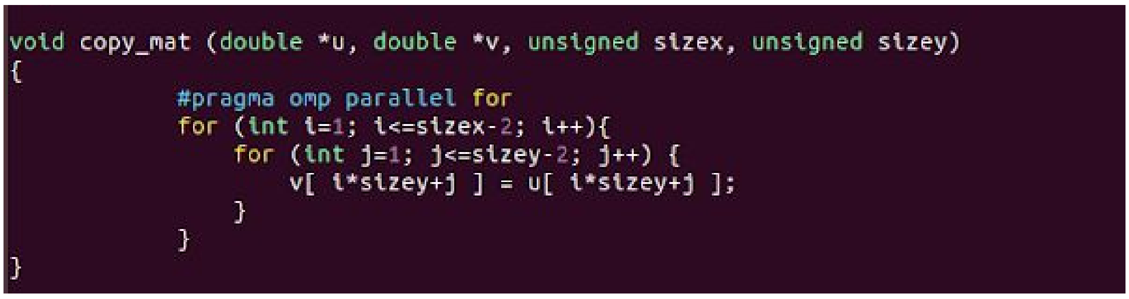
```c
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;
    int howmany=omp_get_max_threads();
    #pragma omp parallel reduction(+:sum) private(diff)
    {
      int blockid = omp_get_thread_num();
      int i_start = lowerb(blockid, howmany, sizex);
      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;
}

/*
```

Fig 2.2 Jacobi parallel code. To make it we make the variable diff private and make a reduction into sum.

After the parallelization the load balancing is still a problem. To have a better load balancing we focused in the function copy_mat in the solve-omp.c code. Our way of solving the problem was parallelize the loop in copy_mat introducing a pragma omp for.



```c
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 ];
                }
            }
}
```

Fig 2.3 Copy_mat code

When we compare the two versions (Fig 2.3 and Fig 2.4l), we can observe the usual. One is using only one thread while the other uses all of them, although it is true that the parallel version uses all the threads for a very short period of time, thus the parallelization of this part might be worthless, since the synchronization overhead may surpass the gain of parallelizing the code. (sequential time 200.000.000 ns approx and parallel time 300.000.000 approx).

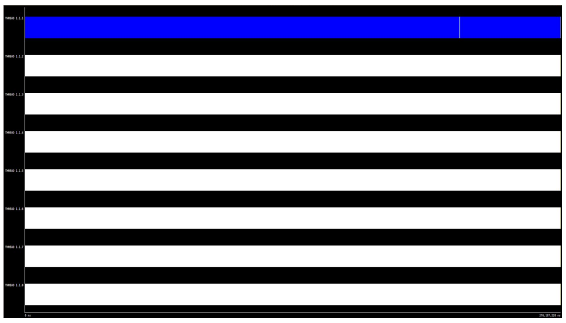
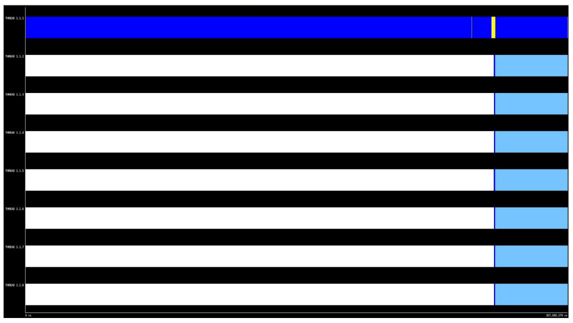
	

Fig 2.4 Sequential Jacobi	Fig 2.5 Parallel Jacobi

Finally when we analyze the speed-up plot (Fig 2.6), we can see how approximately at 4 threads the speed-up plot stops following the ideal plot, and from there it shows a similar performance at different thread count. We can see a little loss of performance for a few thread number around 9 or 10 threads, that may be due to a little misbehaving of the processor, since the next core counts the performance returns to the previous value. But what we can certainly say is that the synchronization overhead is pretty considerable at high thread counts.

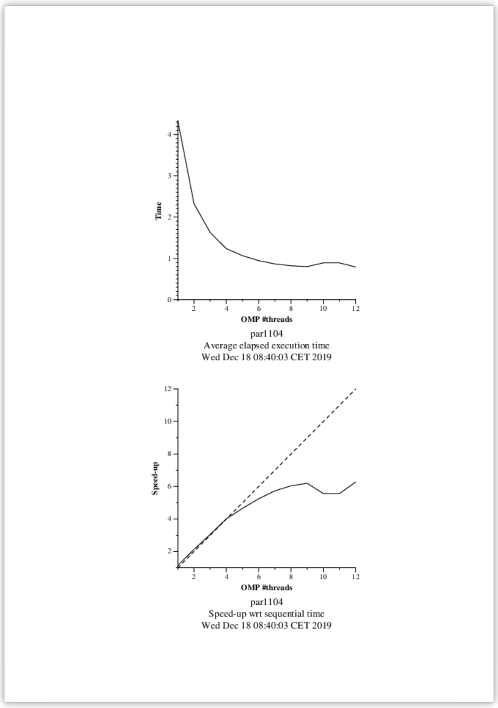
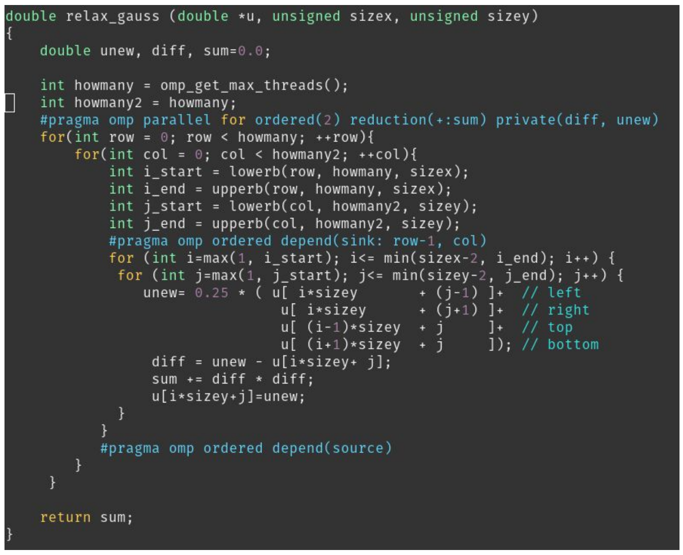


Fig 2.6 Strong scalability Jacobi

3. Parallelization of Gauss-Seidel with openMP ordered

To parallelize the code of the Gauss-Seidel code we had to maintain the dependencies that were shown in the tareador instrumentalization, and also we had to make it using the ordered clause, so we used the for clause to parallelize de two nested for loops with the ordered clause saying that the total of nested loops that are in our doacross nest are the two first loops. As in the Jacobi version the reduction clause for the sum variable is necessary as well as the privatization of the diff and unew variables.



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

    int howmany = omp_get_max_threads();
    int howmany2 = howmany;
    #pragma omp parallel for ordered(2) reduction(+:sum) private(diff, unew)
    for(int row = 0; row < howmany; ++row){
        for(int col = 0; col < howmany2; ++col){
            int i_start = lowerb(row, howmany, sizex);
            int i_end = upperb(row, howmany, sizex);
            int j_start = lowerb(col, howmany2, sizey);
            int j_end = upperb(col, howmany2, sizey);
            #pragma omp ordered depend(sink: row-1, col)
            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;
}
```

Fig 3.1 Gauss-Seidel solver parallelized code

In order to procure that the dependencies are achieved correctly we used the depend clause. As the data decomposition grants that every thread would be execution a portion of the original matrix with row rows and col columns and we know that every chunk depends on the chunk above and on its left we pot an ordered depend clause claiming that to execute this piece of code we have to wait for these chunks to be done being calculated, which means that the iteration with row as row-1 of the actual iteration, and the iteration of the same column have to be over.

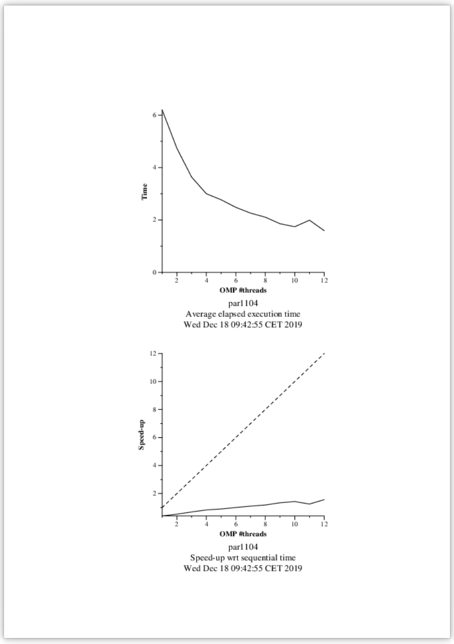
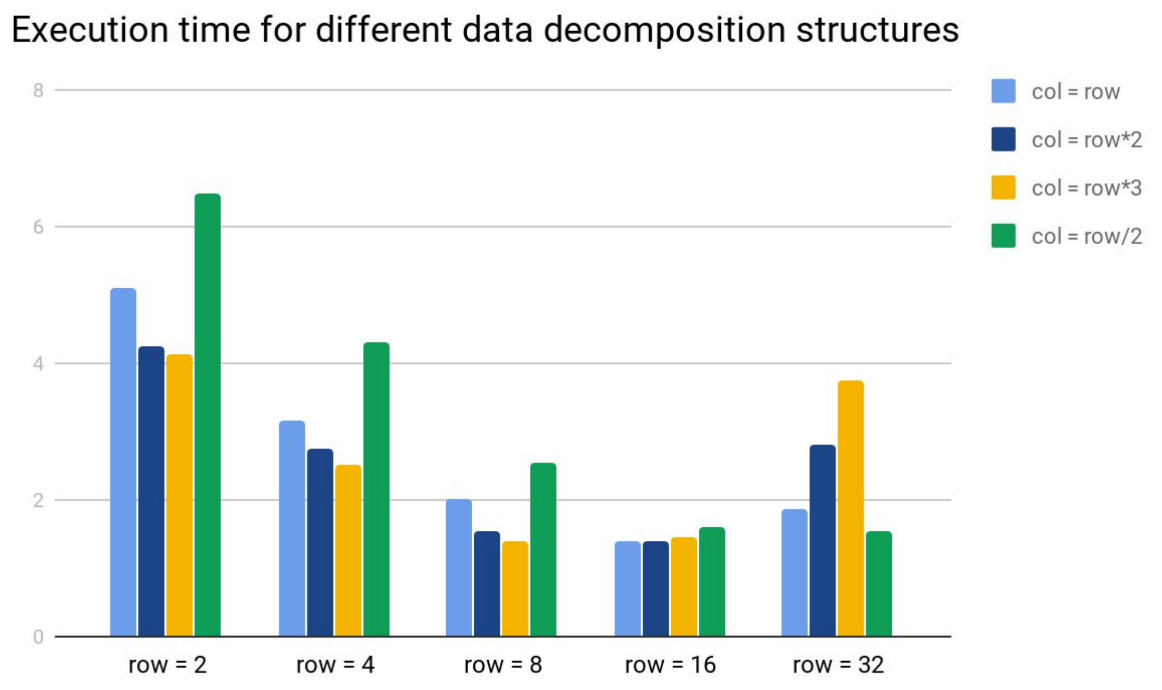


Fig 3.2 Strong scalability plot for the Gauss-Seidel parallelization

As we can see in the strong scalability plot for the parallelized code of the Gauss-Seidel solver, the spee-up plot is not very promising. It is quite clear that the speed-up barely arrives to 2 with 12 cores. This is due to the dependencies that must be achieved in order to properly execute the code, that pulls us to this poor improvement over the seqüential code.
However this code uses a data decomposition that produces a matrix of (number of threads)^2, so we can try to find an optimal data decomposition to maximize this performance.



This plot above shows the different execution time for different structures for the data decomposition that we used, so we could find the optimal value to have a good ratio synchronization/computation. This means that we have to balance the granularity of the data decomposition that we are doing so that there is not much computation per thread nor much synchronization time that might waste time. Then, just looking at the graph we can see that there are a couple of good values for the amount of rows and the amount of columns that the decomposition might have, these are between 8 and 16 rows and between 2 to 3 times more columns.

Conclusion

To sum up, we could see which of the two solvers for the heat equation turned out to be better to parallelized, which was the Jacobi version, mainly due to the dependencies that had to be maintained to properly execute the code, since the Jacobi solver had not a single dependency between iterations, while the Gauss-Seidel solver had several dependencies between iterations that limited our scalability.