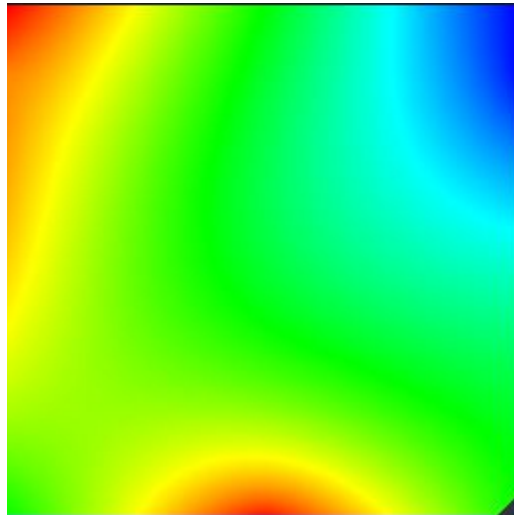


# PAR Laboratory Assignment

---

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



Edgar Perez Blanco  
Bartomeu Perelló Comas  
Group: **PAR2301**

---

PAR - GEI FIB  
Spring 2019-2020

# Contents

<b>Introduction</b>	<b>3</b>
<b>Sequential heat diffusion program</b>	<b>4</b>
<b>Analysis of task granularities and dependencies</b>	<b>5</b>
<b>Parallelization and execution analysis: Jacobi</b>	<b>9</b>
<b>Parallelization and execution analysis: Gauss-Seidel</b>	<b>14</b>
<b>Optional: Using explicit tasks: Gauss-Seidel</b>	<b>18</b>
<b>Conclusion</b>	<b>21</b>

# Introduction

In this last report we will be dealing with different data decomposition strategies. The heat diffusion simulation program can be reduced to solving the heat equation, which in this laboratory will be solved with two different iterative methods, Jacobi and Gauss-Seidel.

We are going to parallelize methods which write on auxiliar structures, so they do not have dependencies between iterations and also methods with methods with dependencies between iterations.

We will use taredor to detect data dependencies, implicit and explicit tasks and extrae to evaluate the performance.

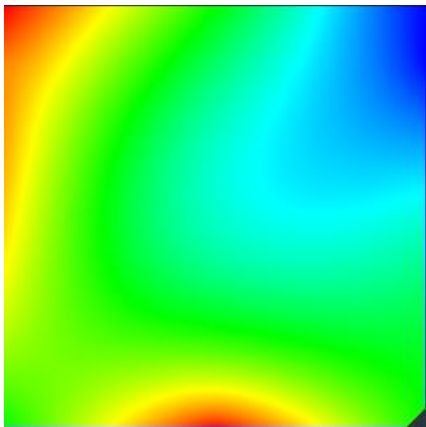
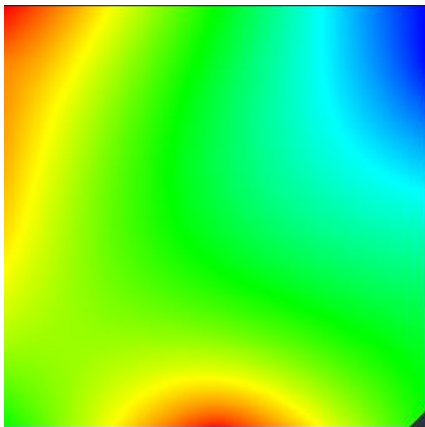
# Sequential heat diffusion program

In order to simulate the heat behaviour of a solid, we must solve the heat equation. Among all the mathematical equation algorithms, we are going to test two of them: Jacobi and Gauss-Seidel.

Both methods are iterative, but leaving the mathematical part aside, from an informatic point of view, the big difference between them is that Jacobi's algorithm writes its output on an external matrix whereas Gauss-Seidel's one writes on the original matrix. This situation provokes that Jacobi's method is easier to parallelize than the other one, due to data dependencies.

To represent real values, the program uses double precision floating point numbers, which we already know that, due to discretization, could not be exactly. So, in order to compare the results, we would assume that the correctness would not be the equality but a fixed number of correct decimals (tolerance).

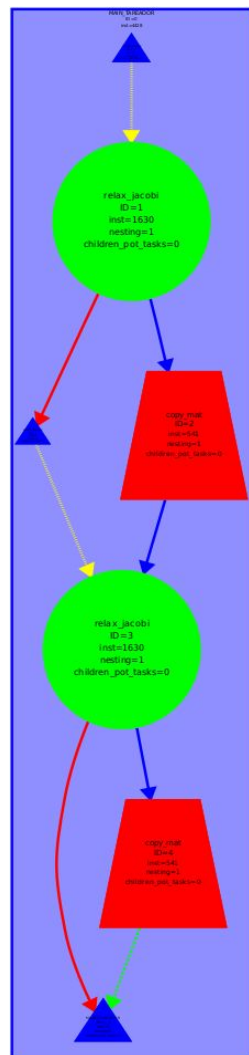
The following captures show for each algorithm the configuration and the obtained results of sequential execution:

Jacobi	Gauss-Seidel
<pre>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: 4.742 Flops and Flops per second: (11.182 GFlop =&gt; 2357.87 MFlop/s) Convergence to residual=0.000050: 15756 iterations</pre>	<pre>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: 2.387 Flops and Flops per second: (8.806 GFlop =&gt; 3688.81 MFlop/s) Convergence to residual=0.000050: 12409 iterations</pre>
	

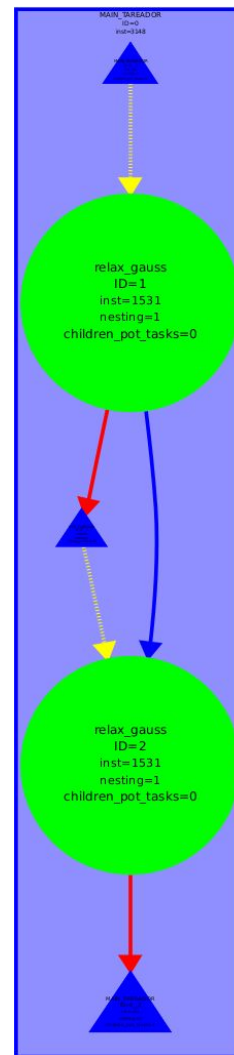
We can see that the images are slightly different (soft blue zone) due to limited accuracy (maximum tolerance = 0.00005). In this Gauss-Seidel do less Flops than Jacobi and lasted less too, therefore has better Flops/s ratio.

# Analysis of task granularities and dependencies

In order to analyse the data dependencies we will be using tareador. First of all, we have inspected the dependencies between different method iterations. As expected, the iterations can not be parallelized, because these mathematical methods' iterations require of the previous iteration to be finished after executing themselves. The following TDGs illustrate this concept for 2 iterations:

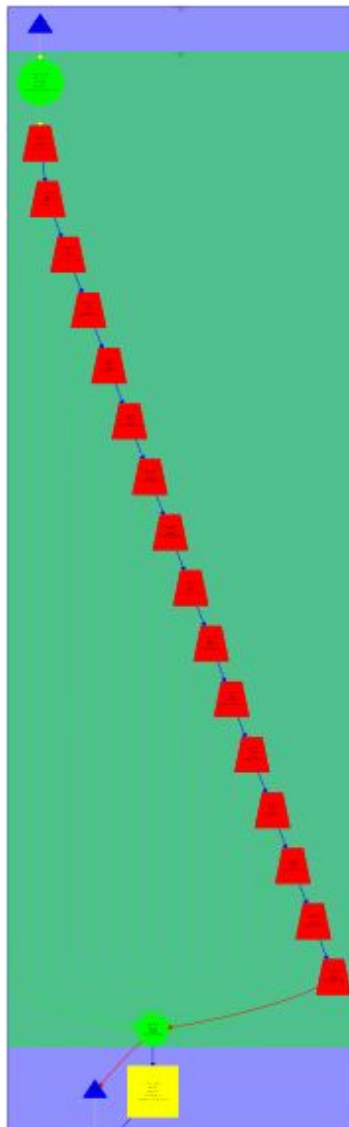


**Jacobi TDG(v0)**

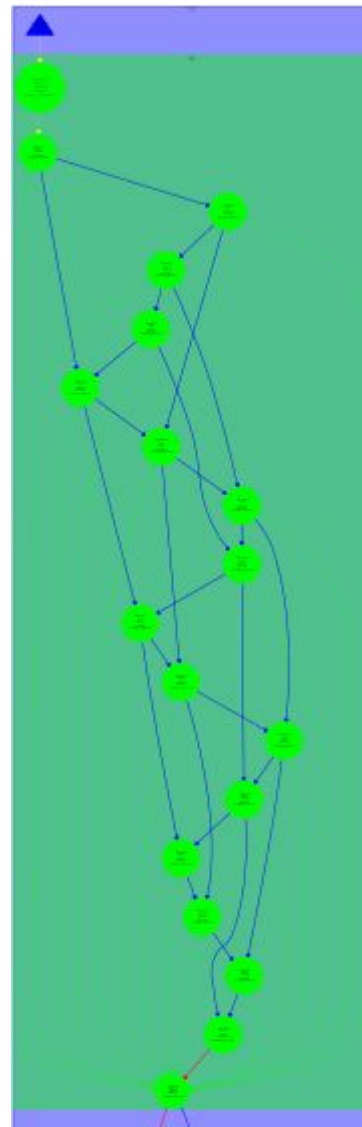


**Gauss-Seidel TDG(v0)**

At this point, as different iterations are not parallelizable and both methods consist in walk through a matrix and compute a new value for each cell, we must look for dependencies among different cell computations. To do this we have created a task for each innermost loop iteration, and we obtained the following TDGs:

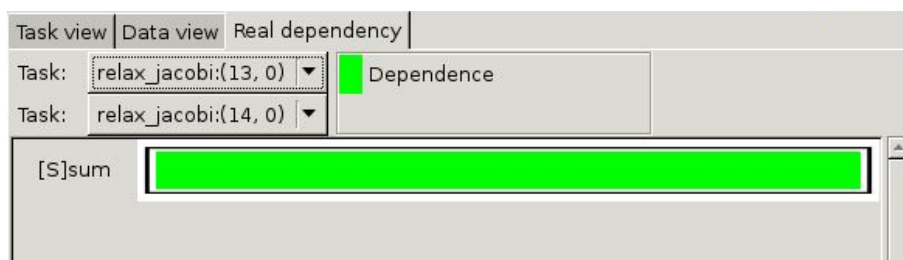


**Jacobi TDG(v1)**



**Gauss-Seidel TDG(v1)**

The TDGs above, show a suspicious behaviour, because they appear to be sequential. This is happening because there is a variable `sum`, which stores the sum of all computations. The following taredor's capture shows the dependence:



This data dependency limits the program in terms of parallelization, but it is an easy problem to solve. If we have used a `#pragma omp for`, we would have added the clause `reduction(+: sum)` else if we have used a `#pragma omp task`, we would have added the clause `#pragma omp atomic` in order to ensure the atomicity of the sum and avoid dependencies. But, as we are working with `tareador` we decided to use the `tareador_disable|enable_object(&var)` construct which makes `tareador` ignore the dependencies that the variable specified is involved.

Here we have the new code version of both algorithms so we can appreciate the new task dependence graphs:

```
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, sizey);
        for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
            for (int j=1; j<= sizey-2; j++) {
                // One task for each innermost loop iteration
                tareador_start_task("relax_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

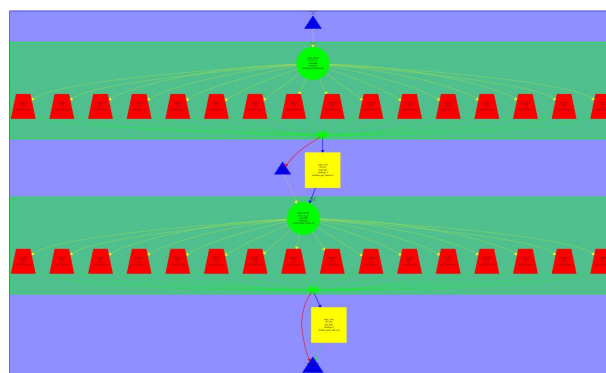
                diff = utmp[i*sizey+j] - u[i*sizey + j];

                // Protect sum variable
                tareador_disable_object(&sum);
                sum += diff * diff;
                tareador_enable_object(&sum);

                tareador_end_task("relax_jacobi");
            }
        }
    }

    return sum;
}
```

***Jacobi modified code***



***Jacobi TDG***

```

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++) {
                // One task for each innermost loop iteration
                tareador_start_task("relax_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];

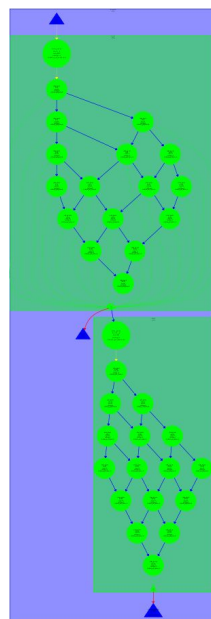
                // Protect sum variable
                tareador_disable_object(&sum);
                sum += diff * diff;
                tareador_enable_object(&sum);

                u[i*sizey+j]=unew;
                tareador_end_task("relax_gauss");
            }
        }
    }

    return sum;
}

```

***Gauss modified code***



***Gauss-Seidel TDG***



The basic difference between both codes is that the jacobi algorithm uses an auxiliary matrix which makes that all access can be made in parallel, on the other side, as gauss-seidel uses the same matrix to access and store values, there are dependencies between iterations making the execution in a wavefront pattern. A wavefront pattern is made by the dependencies being on the block before and the upper one so the advance of the execution is in the inverse diagonals of the matrix.

# Parallelization and execution analysis: Jacobi

As we told before, compute each cell of jacobi's matrix is an embarrassingly parallel problem so we decided to apply an static decomposition, giving each thread a portion of the iterations.

For example, if using 4 threads, the portion of the matrix assigned to each thread would be like the following diagram:

Blockid = 0
Blockid = 1
Blockid = 2
Blockid = 3

To do this, we used the provided macros to generate the first and the last rows' indexes for each thread, depending on its ID. We changed the howmany value to the number of threads in order to divide the matrix in the N portions, where N is the total number of available threads.

```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;
    int blockid, howmany;
    #pragma omp parallel private(diff,blockid,howmany) reduction(+:sum)
    {
        howmany = omp_get_num_threads();
        blockid = omp_get_thread_num(); // Identificador del thread actual

        // Marcamos zona de inicio y fin para el thread iesimo (static scheduling)
        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;
}
```

First of all, we tested the result correctness (it was):

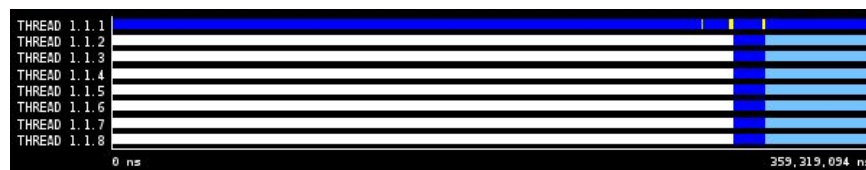
```
par2301@boada-1:~/lab5$  
par2301@boada-1:~/lab5$ diff heat-jacobi-omp.ppm S1/heat-jacobi-seq.ppm  
par2301@boada-1:~/lab5$
```

Parallelizing this part, the results have been better than the sequential ones:

- The MFLOP/S past from approximately 2300 to 3880:

```
par2301@boada-1:~/lab5/S2/jacobiV1/exec$ cat heat-omp_8.times.txt  
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: 2.881  
Flops and Flops per second: (11.182 GFlop => 3880.78 MFlop/s)  
Convergence to residual=0.000050: 15756 iterations
```

- Now the threads work in parallel:



	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.63 %	-	0.36 %	0.00 %	0.00 %
THREAD 1.1.2	4.86 %	95.13 %	-	0.00 %	-
THREAD 1.1.3	4.86 %	95.13 %	-	0.00 %	-
THREAD 1.1.4	4.86 %	95.13 %	-	0.00 %	-
THREAD 1.1.5	4.86 %	95.14 %	-	0.00 %	-
THREAD 1.1.6	4.85 %	95.14 %	-	0.00 %	-
THREAD 1.1.7	4.86 %	95.14 %	-	0.00 %	-
THREAD 1.1.8	4.85 %	95.15 %	-	0.00 %	-
Total	133.65 %	665.97 %	0.36 %	0.02 %	0.00 %
Average	16.71 %	95.14 %	0.36 %	0.00 %	0.00 %
Maximum	99.63 %	95.15 %	0.36 %	0.00 %	0.00 %
Minimum	4.85 %	95.13 %	0.36 %	0.00 %	0.00 %
StDev	31.34 %	0.01 %	0 %	0.00 %	0 %
Avg/Max	0.17	1.00	1	0.48	1

The big initial sequential part corresponds to the code initializations, and once the threads had been created, they started to work. The parallel work is balanced (because all the iterations have the same cost). In spite of this, the code can be improved. We just need to parallelize the last part, which consists in copying the auxiliar matrix into a new one.

The following code shows how we parallelized, with the same technique as before, the copy\_mat function:

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey)
{
    int blockid, howmany;
    #pragma omp parallel private(blockid,howmany)
    {
        howmany = omp_get_num_threads();
        blockid = omp_get_thread_num(); // Identificador del thread actual

        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++)
                v[ i*sizey+j ] = u[ i*sizey+j ];
    }
}
```

After this parallelization, we obtained the following results:

- The MFLOP/S past from 3880 to 13610:

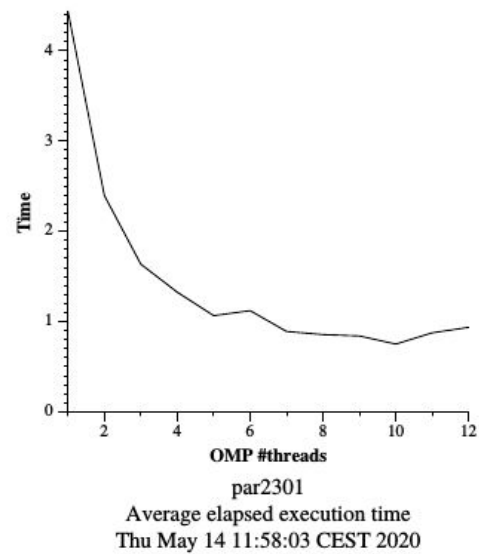
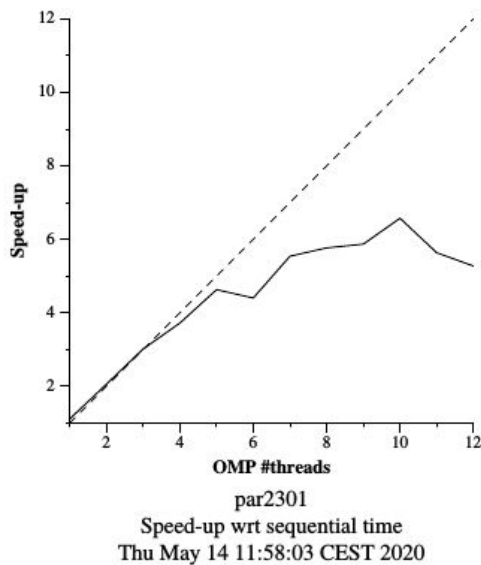
```
par2301@boada-1:~/lab5/S2/jacobiV2/exec$ cat heat-omp_8.times.txt
Iterations      : 25000
Resolution      : 254
Algorithm       : 0 (Jacobi)
Num. Heat sources : 2
  1: (0.00, 0.00) 1.00 2.50
  2: (0.50, 1.00) 1.00 2.50
Time: 0.822
Flops and Flops per second: (11.182 GFlop => 13610.21 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

- The working time percentage has increased, from 4.80% to 7%:

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.45 %	-	0.54 %	0.01 %	0.00 %
THREAD 1.1.2	7.01 %	92.98 %	-	0.00 %	-
THREAD 1.1.3	7.02 %	92.97 %	-	0.00 %	-
THREAD 1.1.4	7.02 %	92.98 %	-	0.00 %	-
THREAD 1.1.5	7.01 %	92.99 %	-	0.00 %	-
THREAD 1.1.6	6.98 %	93.02 %	-	0.00 %	-
THREAD 1.1.7	6.98 %	93.02 %	-	0.00 %	-
THREAD 1.1.8	7.01 %	92.99 %	-	0.00 %	-
Total	148.48 %	650.96 %	0.54 %	0.02 %	0.00 %
Average	18.56 %	92.99 %	0.54 %	0.00 %	0.00 %
Maximum	99.45 %	93.02 %	0.54 %	0.01 %	0.00 %
Minimum	6.98 %	92.97 %	0.54 %	0.00 %	0.00 %
StDev	30.57 %	0.02 %	0 %	0.00 %	0 %
Avg/Max	0.19	1.00	1	0.36	1



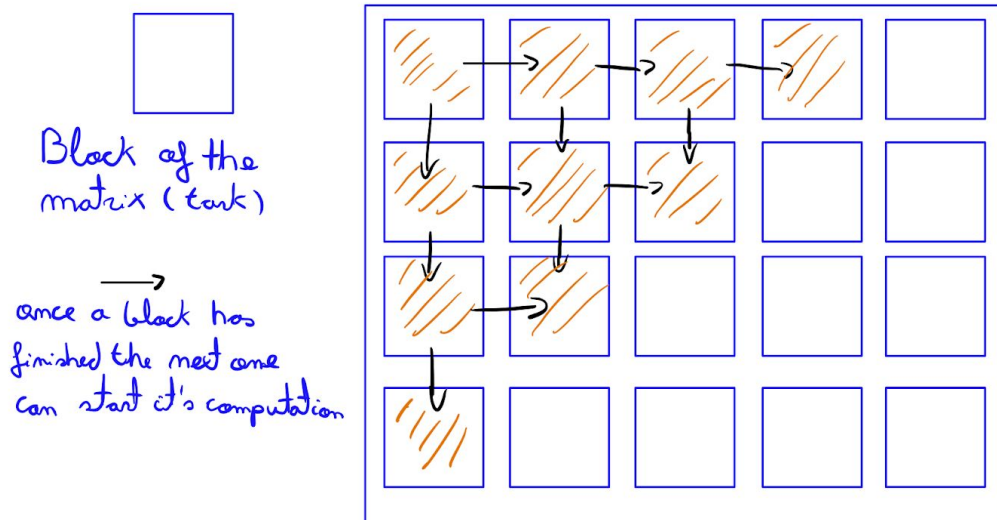
The initialization part continues as is, because cannot do anything, but in the computation part, we increased the speed of the program. The following plots show the strong scalability of this code:



As we can see in the graphs, the speedup is almost ideal from 1 to 5 threads. From 6 to 10 threads, the slope decreases but the curve continues increasing. Once we use 10 threads, due to the associated overheads, the curve starts to decrease. To sum up, the program in this circumstances scales while using a number of threads between 1 and 10.

# Parallelization and execution analysis: Gauss-Seidel

Due to dependencies in the gauss-seidel stencil we thought of a possible solution in order to parallelize the code, firstly we tried the same scheme as jacobi but the execution ended in a minute while in sequential only elapsed 2.38 seconds so we ended up with splitting the matrix in blocks where each one is computed as a task, which in our opinion is the best way to exploit parallelism in this algorithm.



In order to make this block splitting we made each thread compute a serial of blocks depending on the iteration it's given, at first we made the howmany variable be the number of threads involved in the execution.



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

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

            int row_start = lowerb(block_i, howmany, sizex);
            int row_end = upperb(block_i, howmany, sizex);

            int col_start = lowerb(block_j, howmany, sizey);
            int col_end = upperb(block_j, howmany, sizey);

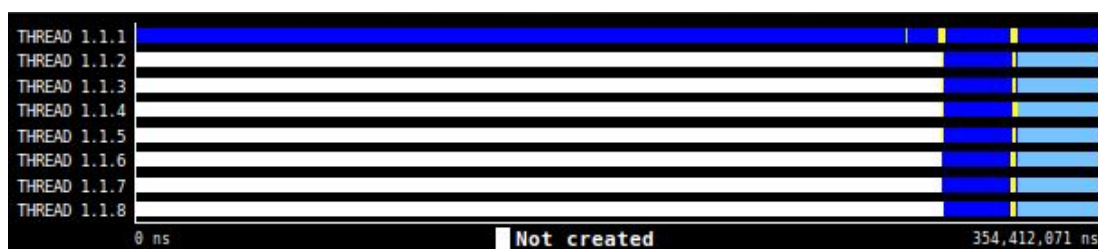
            #pragma omp ordered depend(sink: block_i-1,block_j) depend(sink: block_i, block_j-1)
            for (int i=max(1, row_start); i<= min(sizex-2, row_end); i++) {
                for (int j=max(1, col_start); j<= min(sizey-2, col_end); j++) {
                    unew= 0.25 * ( u[ i*sizey + (j-1) ]+ // 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;
}
```

As we expected the code worked as fine as the original one but with it lasted three times more than the sequential one.

```
par2301@boada-1:~/lab5$ diff heat.ppm paral.ppm
par2301@boada-1:~/lab5$
```

Sequential	Parallel
<pre>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: 2.387 Flops and Flops per second: (8.806 GFlop =&gt; 3688.81 MFlop/s) Convergence to residual=0.000050: 12409 iterations</pre>	<pre>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: 6.487 Flops and Flops per second: (8.806 GFlop =&gt; 1357.50 MFlop/s) Convergence to residual=0.000050: 12409 iterations par2301@boada-1:~/lab5\$</pre>

The results obtained from paraver are very good, we have achieved a good balance in the task distribution because all threads have worked almost the same taking out the sequential part of the execution.



	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.43 %	-	0.56 %	0.01 %	0.00 %
THREAD 1.1.2	8.51 %	91.47 %	0.02 %	0.00 %	-
THREAD 1.1.3	8.52 %	91.46 %	0.01 %	0.00 %	-
THREAD 1.1.4	8.36 %	91.46 %	0.18 %	0.00 %	-
THREAD 1.1.5	8.52 %	91.47 %	0.01 %	0.00 %	-
THREAD 1.1.6	8.33 %	91.49 %	0.18 %	0.00 %	-
THREAD 1.1.7	8.35 %	91.48 %	0.17 %	0.00 %	-
THREAD 1.1.8	8.46 %	91.53 %	0.01 %	0.00 %	-
<b>Total</b>	158.47 %	640.37 %	1.14 %	0.02 %	0.00 %
<b>Average</b>	19.81 %	91.48 %	0.14 %	0.00 %	0.00 %
<b>Maximum</b>	99.43 %	91.53 %	0.56 %	0.01 %	0.00 %
<b>Minimum</b>	8.33 %	91.46 %	0.01 %	0.00 %	0.00 %
<b>StDev</b>	30.09 %	0.02 %	0.17 %	0.00 %	0 %
<b>Avg/Max</b>	0.20	1.00	0.25	0.43	1

Probably the reason behind the execution time being higher is that we don't have enough blocks to exploit the parallelism, in this case we should create more tasks than the number of threads involved.

For some unknown reason when the submit-strong doesn't finish all iterations and it does not give the plots, only eight iterations are done. The results obtained are as bad as expected, if the sequential code is already faster than the parallel in all cases it will be worse.

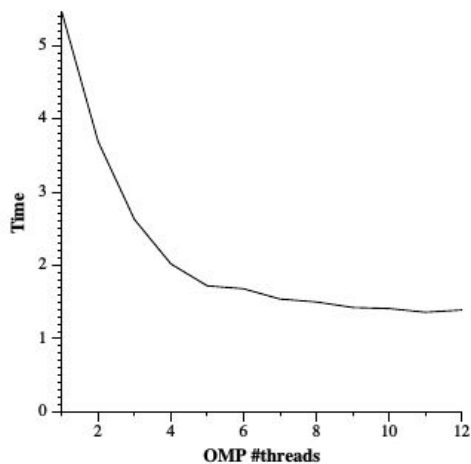
Elapsed (each index is #threads)	Speed Up (each index is #threads)
<pre> 1 6.075 2 6.120 3 6.210 4 6.160 5 6.455 6 6.475 7 6.505 8 6.560 9 6.660 </pre>	<pre> 1 0.403 2 0.400 3 0.394 4 0.397 5 0.379 6 0.378 7 0.376 8 0.373 9 0.367 </pre>



If we want to achieve better performance we should do what we mentioned before, increase the total number of blocks we have tried with block size 8, 16, 32, 64 and 128.

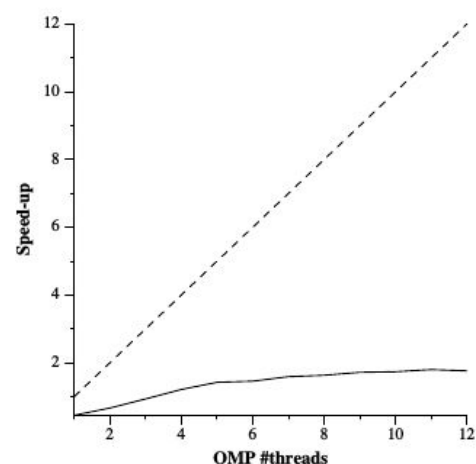
<b>Seq</b>	<b>8</b>	<b>16</b>	<b>32</b>	<b>64</b>	<b>128</b>
2.39s	6.48s	1.38s	1.88s	5.68s	18.541s

As we can see on the table above the best splitting of the matrix is in 16 blocks. We wondered how the 16 by 16 block splitted matrix would behave on different threads numbers and these are the results.



par2301  
Average elapsed execution time  
Fri May 15 14:45:48 CEST 2020

**Execution time plot**



par2301  
Speed-up wrt sequential time  
Fri May 15 14:45:48 CEST 2020

**Speedup plot**

## Optional: Using explicit tasks: Gauss-Seidel

We had been experimenting with implicit tasks and then we have tried to use explicit tasks. We made use of the depend clauses, in order to ensure the correct execution. We used the same structure as before, a block division.

As always, we used the `#pragma omp parallel` and `#pragma omp single` to create the tasks just once. We manually created a local variable to store the sum of each block and once the task is finished, add the local value to the global one with an atomic clause. We needed to create a simple structure to manage the dependences, so we decided to use a two-dimensional array of chars just to reference each memory position (we used a char because its size, a single byte, is the minimum allowed to deal with).

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;
    int howmany = 2;

    char salu2[howmany][howmany];

    #pragma omp parallel
    #pragma omp single
    for (int block_i=0; block_i < howmany; ++block_i) {
        for (int block_j=0; block_j < howmany; ++block_j) {
            #pragma omp task depend(in: salu2[block_i-1][block_j]) \
                                depend(in: salu2[block_i][block_j-1]) \
                                depend(out: salu2[block_i][block_j])
            {
                int row_start = lowerb(block_i, howmany, sizex);
                int row_end = upperb(block_i, howmany, sizex);

                int col_start = lowerb(block_j, howmany, sizey);
                int col_end = upperb(block_j, howmany, sizey);

                double local_sum = 0.0;
                for (int i=max(1, row_start); i<= min(sizex-2, row_end); i++) {
                    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];
                        local_sum += diff * diff;
                        u[i*sizey+j]=unew;
                    }
                }
                #pragma omp atomic
                sum += local_sum;
            }
        }
    }
    return sum;
}
```

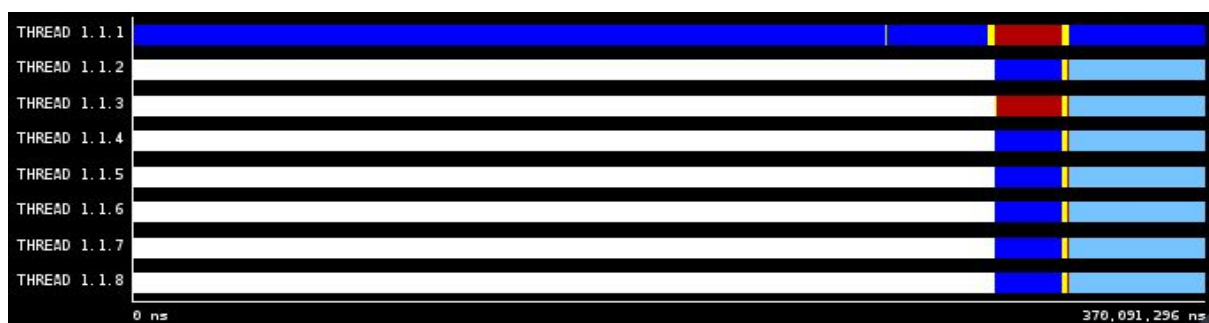
After testing some block sizes, we obtained the following results:  
(With a size = N we have  $N^2$  blocks and, therefore,  $N^2$  tasks)

Seq	2	4	8	16	32
2.39s	7.807s	6.333s	5.171s	5.227s	13.218s

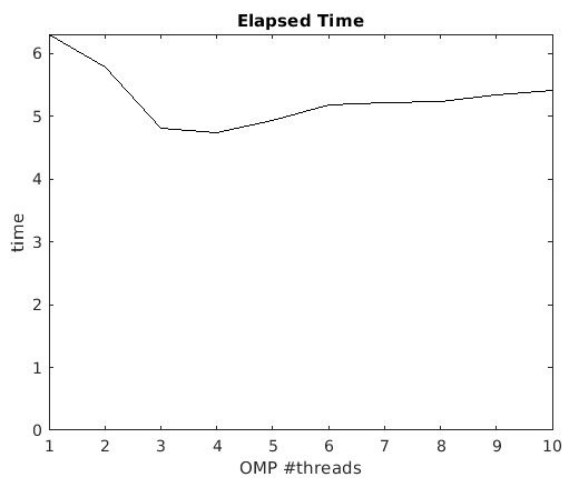
The best result was obtained with  $N = 8$ . We think that, as we use 8 threads, and maximum number of simultaneously running tasks is the size of the diagonal, which has 8 elements; creating more, due to the overheads of task creations, would be worse.

In general the execution lasts more time than the implicit tasks version, due to task creation and synchronization.

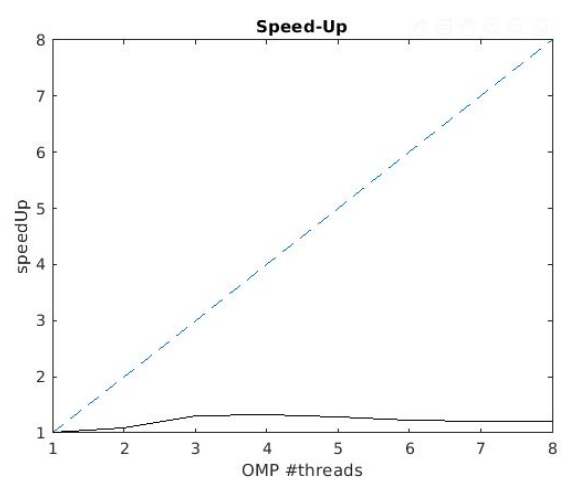
	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	92.76 %	-	6.75 %	0.49 %	0.01 %	0.00 %
THREAD 1.1.2	7.55 %	91.99 %	0.45 %	0.00 %	0.00 %	-
THREAD 1.1.3	0.40 %	91.99 %	7.60 %	0.00 %	0.00 %	-
THREAD 1.1.4	7.40 %	91.99 %	0.60 %	0.00 %	0.00 %	-
THREAD 1.1.5	7.41 %	91.99 %	0.56 %	0.03 %	0.00 %	-
THREAD 1.1.6	7.42 %	92.04 %	0.54 %	0.00 %	0.00 %	-
THREAD 1.1.7	7.36 %	92.06 %	0.56 %	0.02 %	0.00 %	-
THREAD 1.1.8	7.37 %	92.03 %	0.59 %	0.00 %	0.00 %	-
Total	137.68 %	644.09 %	17.65 %	0.55 %	0.02 %	0.00 %
Average	17.21 %	92.01 %	2.21 %	0.07 %	0.00 %	0.00 %
Maximum	92.76 %	92.06 %	7.60 %	0.49 %	0.01 %	0.00 %
Minimum	0.40 %	91.99 %	0.45 %	0.00 %	0.00 %	0.00 %
StDev	28.65 %	0.03 %	2.88 %	0.16 %	0.00 %	0 %
Avg/Max	0.19	1.00	0.29	0.14	0.50	1



In terms of scalability we have not good news. Due to overheads associated to synchronization and task creations, the performance it is so bad and does not improve respect to the implicit tasks version.



***Execution time plot***



***Speedup plot***

# Conclusion

In this session we have learned how to distribute the work through block decomposition in order to improve the parallelizable part of the programs.

Once the Jacobi has been parallelized and the `copy_mat` function the improvement are pretty clear, but when we try to parallelize the `gauss_seidel` we end up with a version which is still worse than the sequential, after some investigation we found out that we could beat the sequential time with 8 threads and splitting the matrix in 16 x 16 or 32 x 32 blocks, in this case we may have learned that parallelizing not always is the best way to improve the execution time, because in this case we could improve the execution time but it's not sure that in another case we could have achieve it.

Finally when we are told to parallelize the `gauss_seidel` algorithm with the `depend` construct none of the different block matrix distributions are good enough to beat the sequential execution time.