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

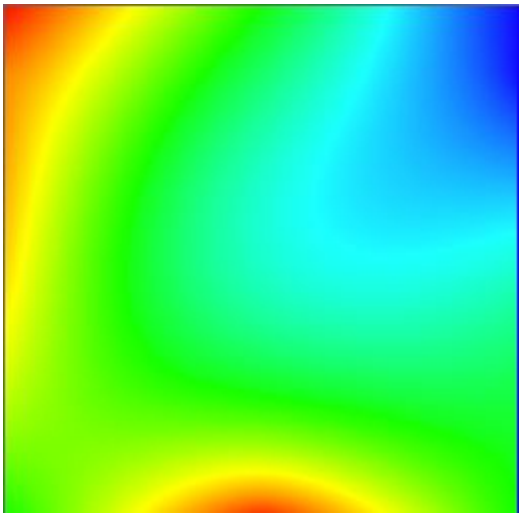
Marc Domínguez de la Rocha
Joan Sánchez García
par2103

Fall 2017-2018
09/01/17

Index

5.1 Introduction

In this session we will be working on the parallelization of a sequential code that simulates heat diffusion in a solid body using two different solvers for the heat equation: Jacobi and Gauss-Seidel. Each with different numerical properties.



The picture over we will be working is the one shown above. The image shows the resulting heat distribution when two sources are placed in the borders of the 2D solid.

The purpose of this session will be to parallelize the two solvers with the aim of gaining SpeedUp, but continue generating a valid code, that's it, to continue generating the same resulting heat image as the sequential one.

5.2 Tareador analysis

Jacobi:

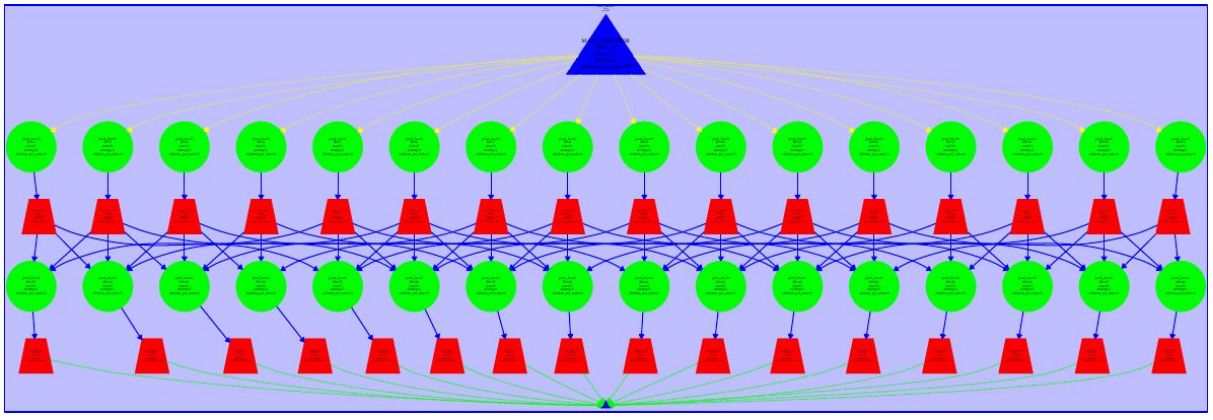


image 1: Tareador task graph for the Jacobi solver. HD image at the annexes/images/tareador/tareador-jacobi.pdf

Gauss-Seidel:

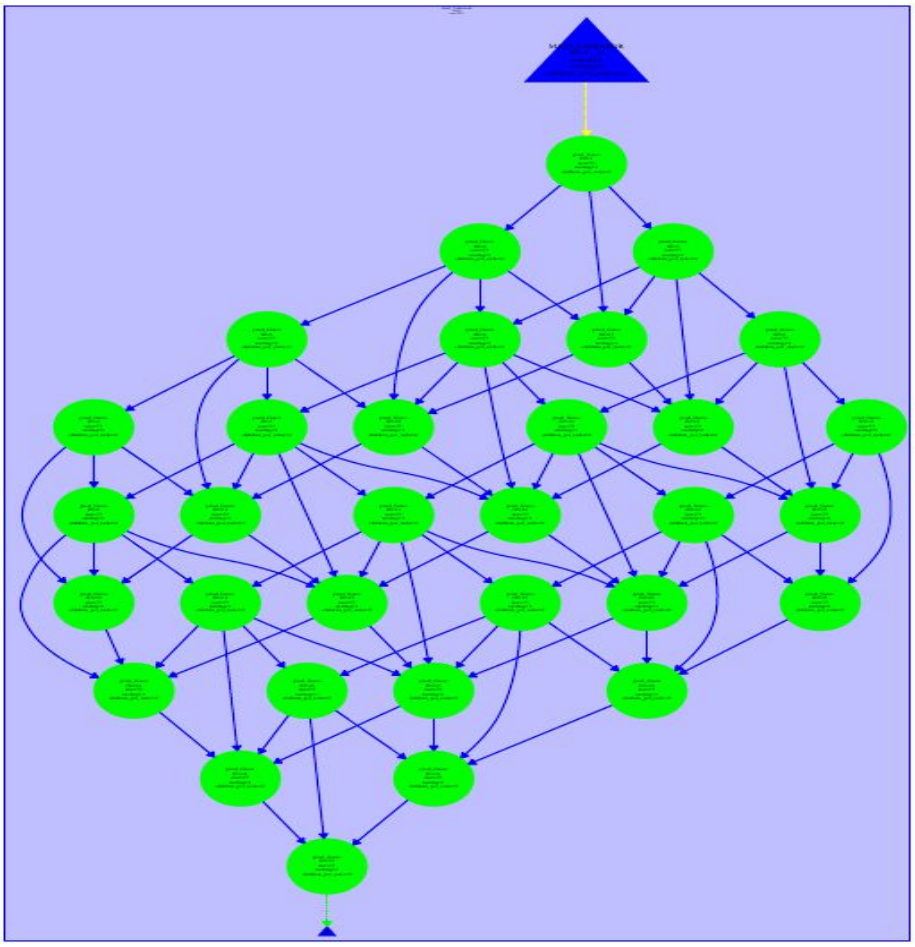


image 2: Tareador task graph for the Gauss-Seidel solver. HD image at the annexes/images/tareador/tareador-gauss.pdf

Code:

```
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("copy_mat_cell");
        v[ i*sizey+j ] = u[ i*sizey+j ];
        tareador_end_task("copy_mat_cell");
    }
}

/*
 * 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("pixel_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("pixel_Jacobi");

    }
        }
        }

        return sum;
}
```

code 1: modified solver-tareador. c code. Complete code for solver-tareador.c and heat-tareador.c can be found at annexes/src/tareador

The dependency graph appended for Jacobi and Gauss-Seidel are the final ones, both had a dependency that made the graph a complete disaster. That dependency were made by the *sum* variable used to store partial results, so a new instrumented code to remove that dependency were needed, which is the code above. As may seem confusing that in both versions there are 32 tasks and our working matrix is *4x4,* would be good to know that two iterations are being done, that's it two samples being taken. In *OpenMP* there are a lot of possible directives and ways to give private variables to our threads or do something similar with reductions.
The first choice would remove the dependence but then we need to gather together the partials results and the second choice actually does that, a reduction to sum (also allows us to substract, multiply, etc) will store partial results in a local variable for each thread and then sum all of them into one.

5.3 OpenMP parallelization and execution analysis: Jacobi

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey)
{
        int howmany = omp_get_max_threads();
        static int i, j;
        #pragma omp parallel for
        for (i=1; i<=sizex-2; i++)
        for (j=1; j<=sizey-2; j++)
                v[ i*sizey+j ] = u[ i*sizey+j ];
} //This does not matters if you parallelize it, is just for performance...
```

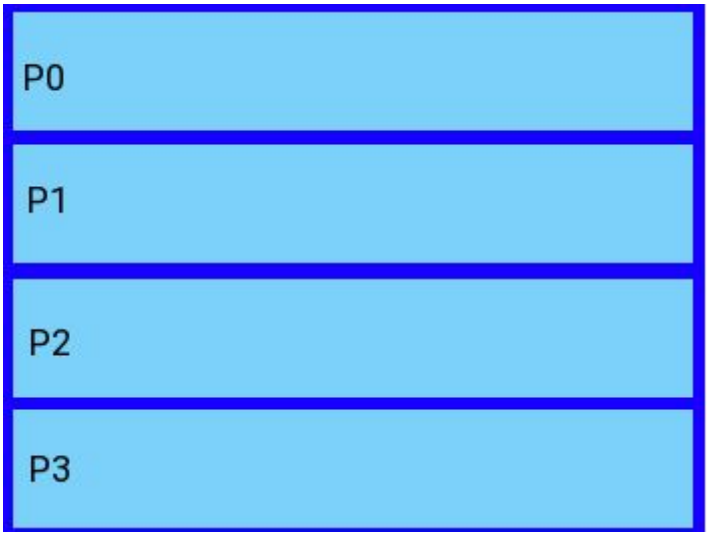
```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;

    int howmany = omp_get_max_threads();

    #pragma omp parallel for private (diff) reduction(+: sum)
    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++) {
            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;
}
```

code 2: modified solver-omp.c code for Jacobi version. Complete code for solver-omp.c and heat-omp.c can be found at annexes/src/jacobi-omp/



The data decomposition is supposed to be balanced, that means that the data is equally distributed. Indeed the data decomposition could result unbalanced if the *size* is not divisible by the number of processors used. Specifically, the remainder of that division will be distributed among all the processors. For example, our parallel program process nine elements, then two elements per core will be assigned, but the last element left is going to be assigned to the first processor *P0*, and if the number of elements was ten, then one element would be assigned to both processors *P0* and *P1*, and so on.

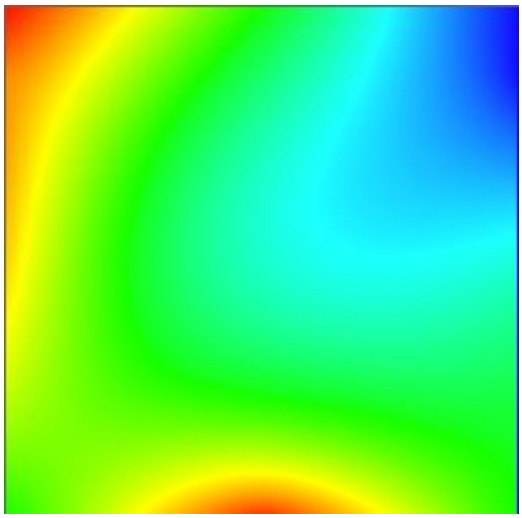


image 4: heat-jacobi.ppm for the original version

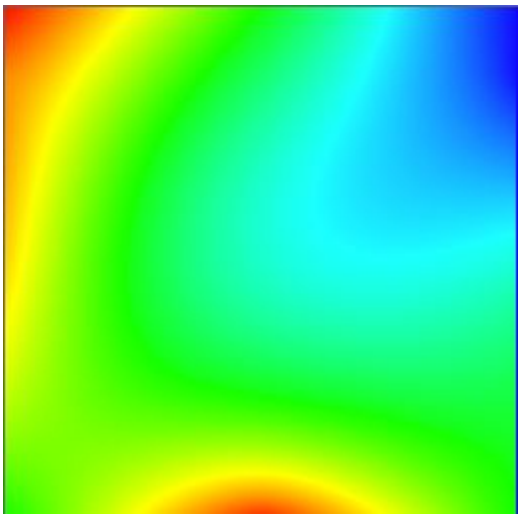


image 5: heat-jacobi.ppm for the parallel version

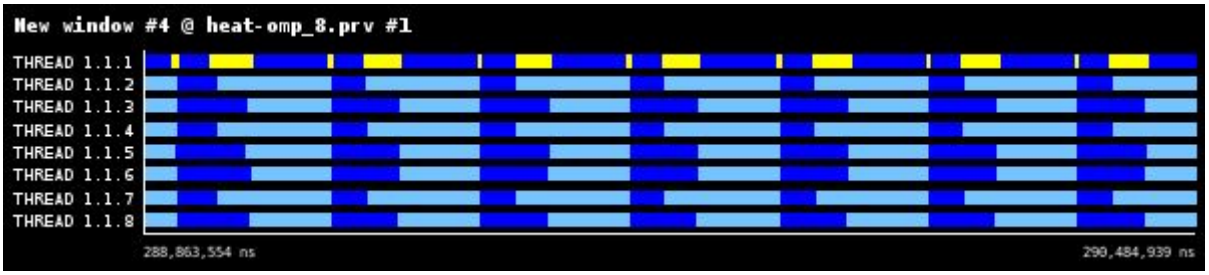


image 6: Paraver trace of the parallelized Jacobi version

Now the execution of the Jacobi iterations is done in parallel, the speed-up obtained is not ideal due to the dependencies seen with Tareador. A big bottleneck in the code is the roughly sequential copy of a matrix, which is slowing down our execution time.

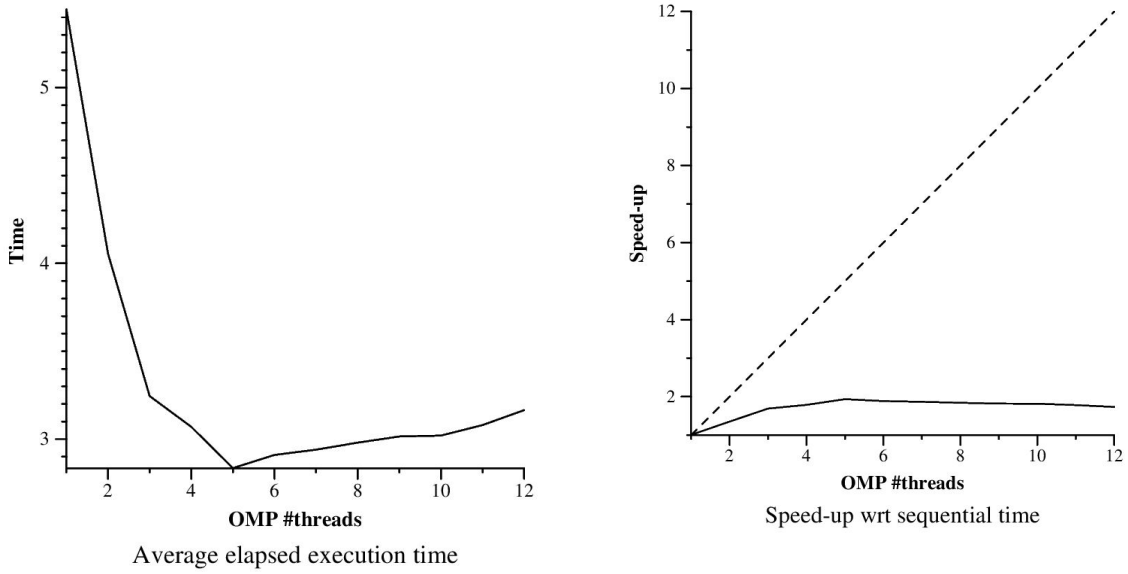


image 7: Speed-Up for the Jacobi parallelization. HD image can be found at annexes/images/jacobi/heat-omp-strong-jacobi.ps

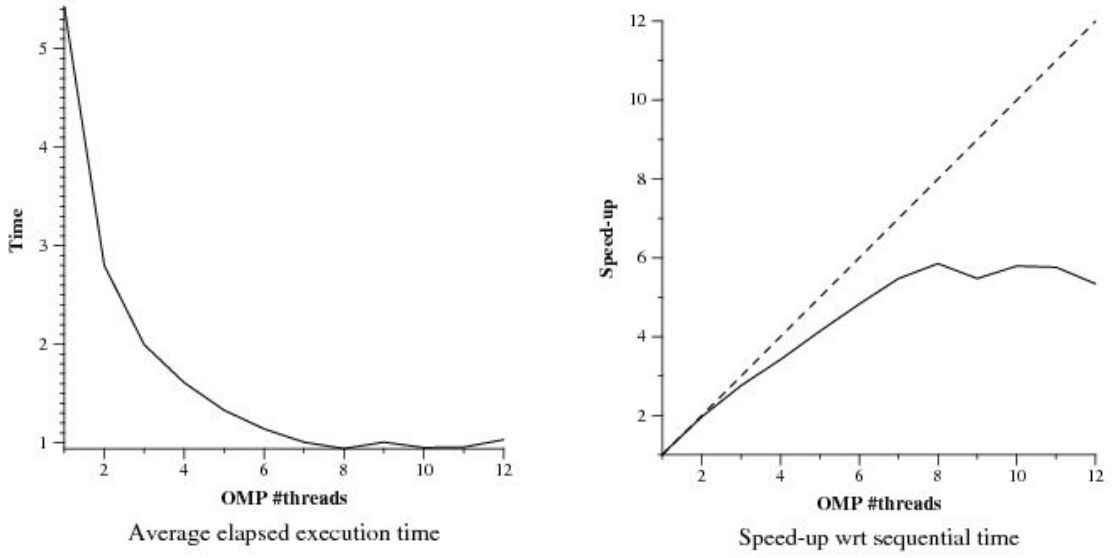


image 8: Speed-Up for the Jacobi and copy mat parallelization. HD image can be found at annexes/images/jacobi/heat-omp-strong.ps

As the images show, the difference between the speed-up parallelizinig just the jacobi solver and also parallelizing the copy_mat function are notable. This is because the time spent on the second function isn't negligible. So, as we learned in Amdahl's law, as many improvements we can apply to the most used part of our code, better speed up we will obtain.

5.4 OpenMP parallelization and execution analysis: Gauss-Seidel

```
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_row = 0; blockid_row < howmany; ++blockid_row) {
                for (int blockid_col = 0; blockid_col < howmany; ++blockid_col) {
                        int i_start = lowerb(blockid_row, howmany, sizex);
                        int i_end   = upperb(blockid_row, howmany, sizex);
                        int j_start = lowerb(blockid_col, howmany, sizey);
                        int j_end   = upperb(blockid_col, howmany, sizey);
                        #pragma omp ordered depend (sink: blockid_row-1, blockid_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;
}
```

code 3: modified solver-omp.c code for Gauss version. Complete code for solver-omp.c and heat-omp.c can be found at annexes/src/gauss-omp/

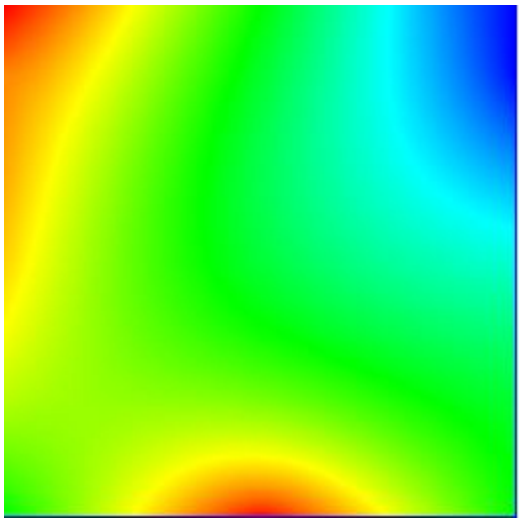


image 8: heat-gauss.ppm for the original version

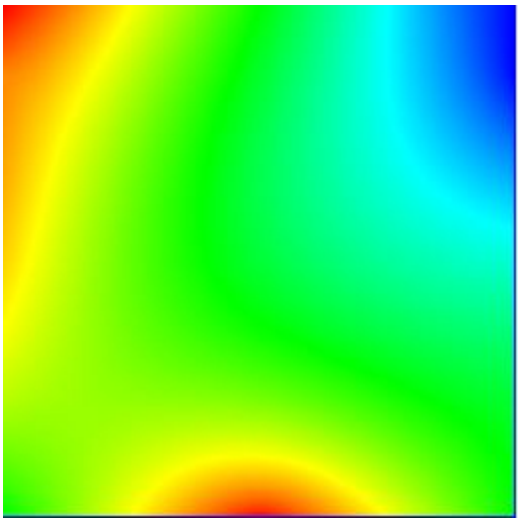


image 10: heat-gauss.ppm for the parallel version

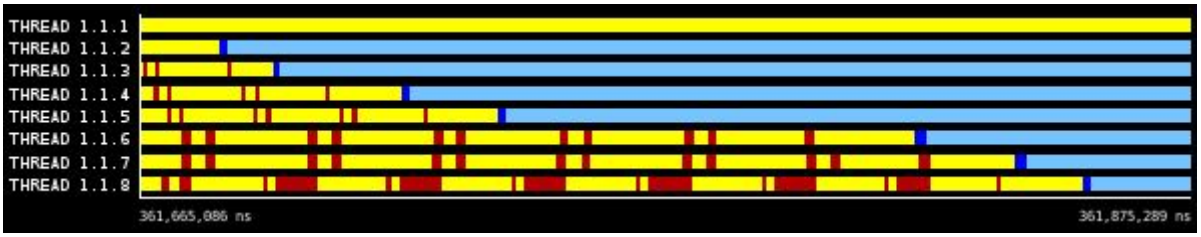


image 11: Paraver trace of the parallelized Gauss version

For every computed block we release two dependences, the ones related to the lower and right blocks, but as one thread compute a whole blocked row we won't talk about the right dependence.
So we expect to see work interleaved between threads because one thread will compute a block, start blocking the next block and another thread will start to compute its first block. But the shown trace it's not exactly that way, only the first block seems to be appearing.

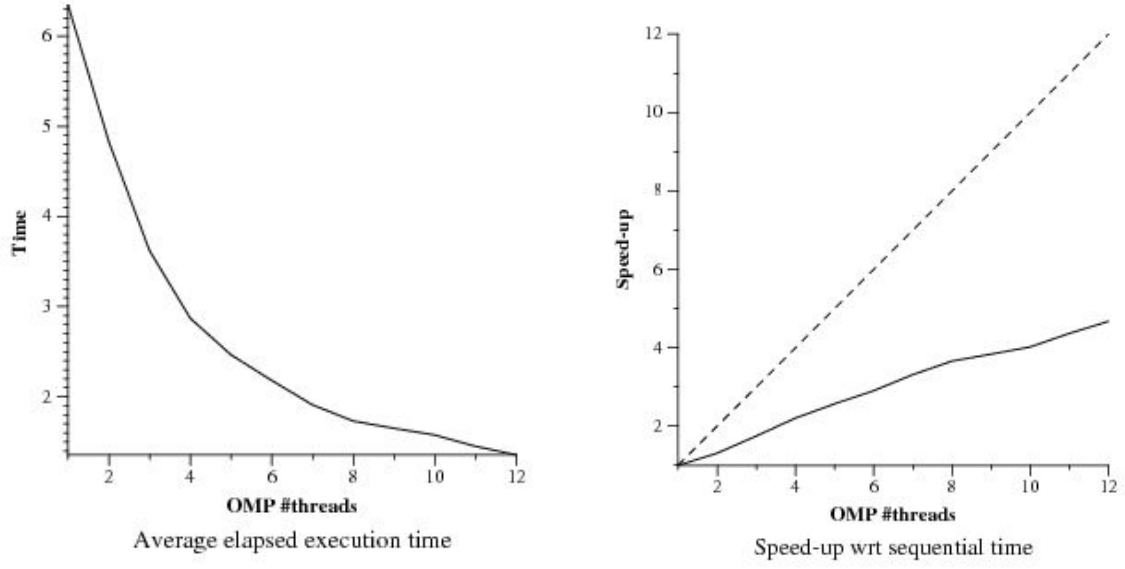


image 12: Speed-Up for the Gauss parallelization. HD image can be found at annexes/images/gauss/speedup-gauss.png

As we can see comparing the speedup plots for Jacobi and Gauss paralelization, we realize that the Gauss version is better than the Jacobi alone, but behaves worst than the Jacobi with copymat parallelization.
This can be explained with the Gauss grainsize. As the blocks are smaller than the Jacobi and the depend clauses allows a dependent block to start its execution as soon as the block it depends finisheds, we can execute the matrix faster, improving the SpeedUp consequently.

We will study the Gauss parallelization for different blocksizes for the columns. Powers of two will be used for these sizes. Note that *blocksize = 1* would be meaningless because despite having more threads than just one, the dependences between them provoke a not desired sequential execution.

Blocksize = 2

First we are going to try with a blocksize (from now on we refer to blocksize as the blocksize for the columns) equal two.

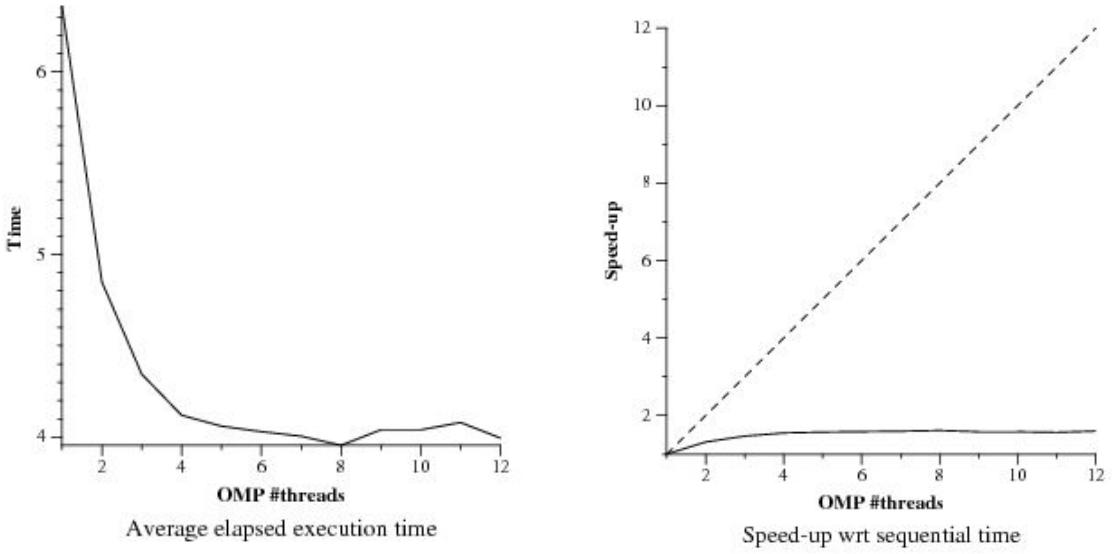


image 13: Speed-Up for the Gauss parallelization with blocksize = 2.

The sequential version time to execute is near 6 seconds, now the less value of time we obtained with this blocksize is 4 seconds with 8 threads. For more threads than 8, the time starts to increase, this could be because of load unbalance problems, but this version is worse than the one we made before taking omp_get_max_threads() as a reference.

Blocksize = 4

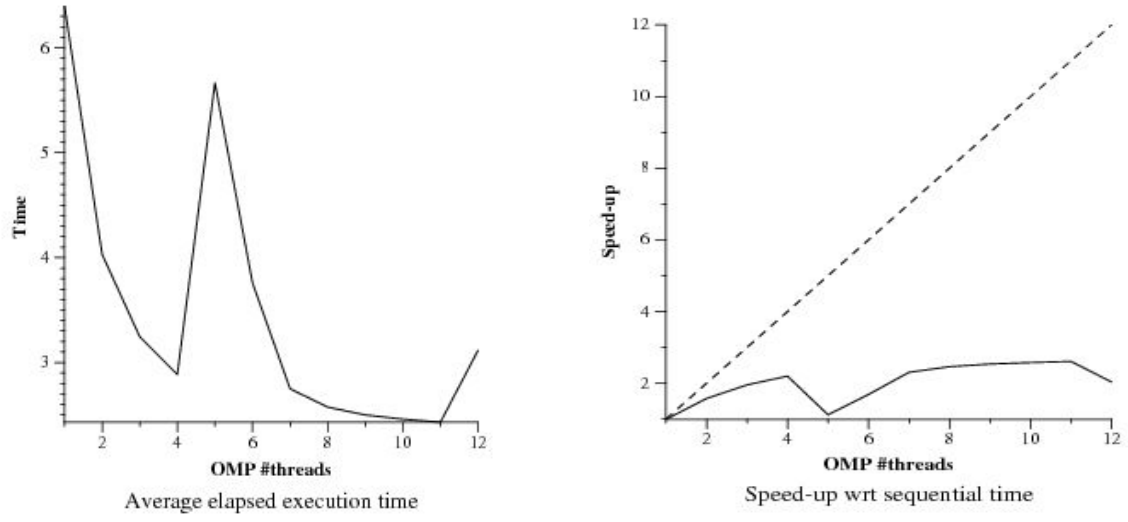


image 14: Speed-Up for the Gauss parallelization with blocksize = 4.

This version reaches a better speed-up than the one with blocksize = 2, but just for 4, 8, 10, 11, 12 threads. It has many peaks of time between 4 and 8, and for more than 12 threads, the execution time triggers up.

Blocksize = 8

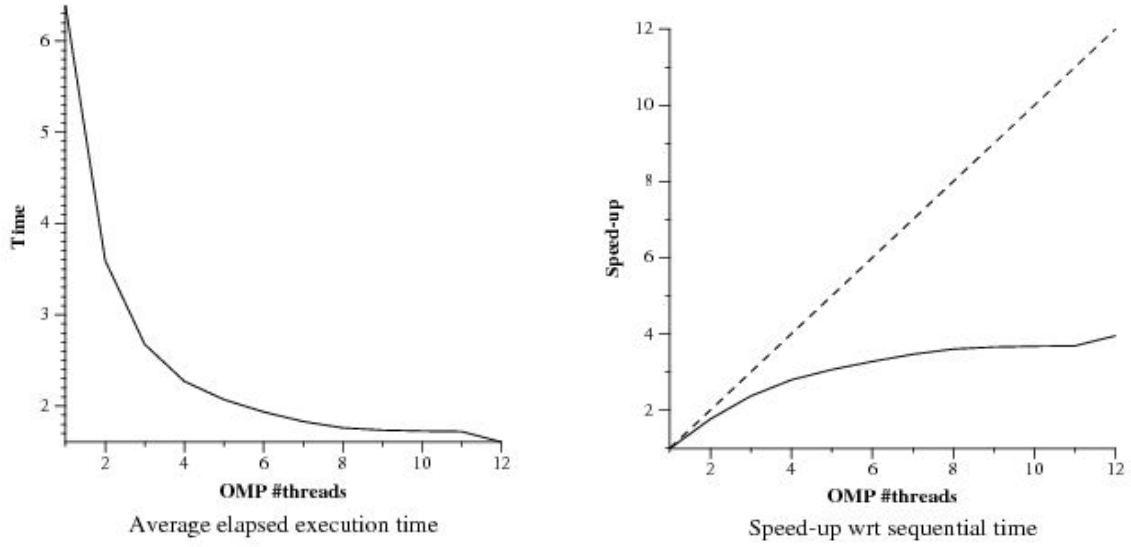


image 15: Speed-Up for the Gauss parallelization with blocksize = 8.

This speed-up is quite near from the original version, by the moment we could say that increasing the blocksize is good.
Theoretically that's true because for every computed block we can start two new blocks, but then, having a thread per memory position of the matrix would lead the program to a better performance? Actually, this is not true, the time each synchronization between threads take versus the workload every thread does is not worth it. So blocking the matrix is a good practice but let's find the optimum value.

Blocksize = 16

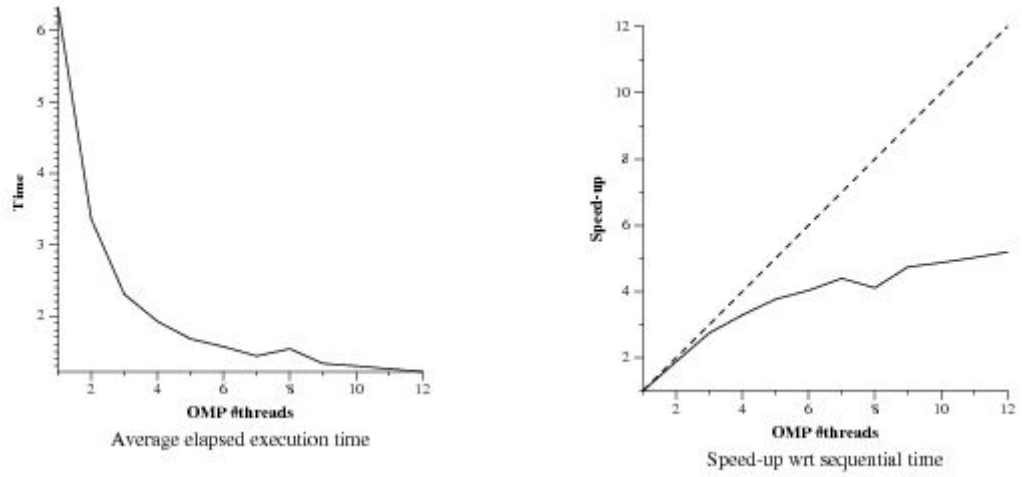


image 16: Speed-Up for the Gauss parallelization with blocksize = 16.

Here we got a better speed-up than the original version, giving a row divided into 16 blocks to each thread achieves a good performance.
Just to illustrate that at some point the performance stops growing up and start to go down look at this graph with a blocksize = 128.

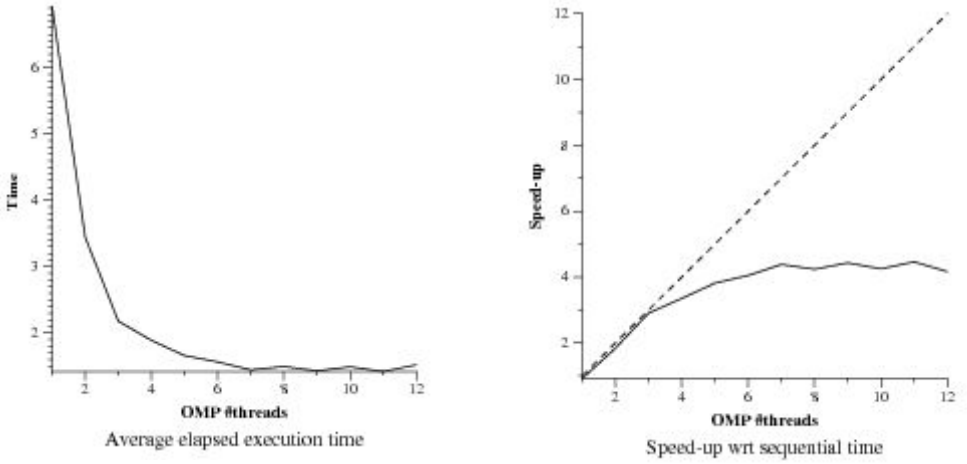


image 17: Speed-Up for the Gauss parallelization with blocksize = 128.

So we can conclude this section knowing that more blocks are not always better and that the first optimum value we found is blocksize = 16.

5.5 Optional

```
#pragma omp parallel reduction(+: sum)
{

#pragma omp for
for (k = 0; k < 512; ++k) vector_blocks [k] = 0;

int howmany=omp_get_num_threads();

#pragma omp single
for (int blockid = 0; blockid < howmany; ++blockid) {
  for (int blockid_column = 0; blockid_column < howmany; ++blockid_column) {
       #pragma omp task private(unew, diff) firstprivate(blockid, blockid_column) \
       depend(in:  vector_blocks [(blockid-1)*howmany + blockid_column], vector_blocks [blockid*howmany + (blockid_column-1)]) \
       depend(out: vector_blocks [blockid*howmany + blockid_column])
       {
       printf("Soy el thread %d, con el bloque: %d, %d\n", omp_get_thread_num(), blockid, blockid_column);
       int i_start = lowerb(blockid, howmany, sizex);
       int i_end   = upperb(blockid, howmany, sizex);
       int j_start = lowerb(blockid_column, howmany, sizey);
       int j_end   = upperb(blockid_column, howmany, sizey);
       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;
           }
       }
       vector_blocks [blockid*howmany + blockid_column] = 1;
       printf("Bloque: %d, %d, computado\n", blockid, blockid_column);
```

Despite the version we made does not work, we wanted to show at least our trial.
The idea is to create a task per block, depending on every upper and left block to itself. So, every block is going to start computing at the time its dependencies are computed, like a doacross. We allocated memory for a vector to know which block is computed, establishing dependencies in base that vector values are changed so the execution is done like we mentioned above. Possible problems for that version is that howmany*howmany tasks are created, for a large number of threads this could be a greater overhead, diminishing the potential speed-up a task version versus a parallel for version can achieve.
But it is interesting because despite the task overhead, this version can get benefit from the number of the total number of cores (so, number of threads) the machine have.

5.6 Conclusions

As we have shown all across the document, choosing a strategy or another can imply getting quite different SpeedUps.

If we just look to the solver algorithm parallelization isolated, we can see that the Gauss-Seider behaves better than the Jacobi. As we have said, this can be explained with the Gauss grainsize. As the blocks are smaller than the Jacobi and the depend clauses

allows a dependent block to start its execution as soon as the block it depends finishes, we can execute the matrix faster, improving the SpeedUp consequently.

But in the first version of the code we used the number of processors as a block size both for rows and columns, and we were asked if that was the optimal value. To check it, we obtained the SpeedUp for block sizes multiples of 2, from 2 to 16 and then jumping to 128 just to show that not bigger block size means better SpeedUp. We concluded that 16 processors, or what is the same, 16 iterations as block size was the optimal value.

In the Jacobi solver we identified that the part which takes a bigger portion of the execution time is not the one solving the image itself, if not the copy mat function. So as soon as we parallelized this part, the SpeedUp increases considerably.

As a conclusion of this course, we have seen that parallelization can help improving the execution time by far, but again, as we learned in Amdahl law, the smaller improvement we can apply to the most used portion in the code can be more notable than improving to perfection a not so used portion, as we have seen in Jacobi with copymat parallelization.