# PAR — Deliverable for Laboratory Session 5

Víctor Jiménez Arador, Miguel Moreno Gómez

Group $12_{02}$ — Fall semester 2017/2018

## 1 Introduction

The purpose of this laboratory is to evaluate and implement a parallel strategy for two iterative algorithms for solving the heat equation problem, that is, applying parallel strategies for two different algorithms that solve linear equations (Jacobi and Gauss-Seidel). We'll describe these algorithms:

- **Jacobi method:** Is a numeric linear algebra method for finding solutions in a matrix that represents a system of equations if and only if the matrix is diagonally-dominant. A diagonally-dominant matrix is a matrix such that $\forall i : \sum_{i \neq j} |a_{ij}| \leq |a_{ii}|$. In Jacobi, each diagonal element is solved iteratively and stops with a solution the moment it converges, and it decomposes the matrix into two: one that has values different to zero on its diagonal and its complementary (that is, a matrix where the diagonal values are set to zero but the other elements don't).

- **Gauss-Seidel method:** another numeric linear algebra method for solving systems of equations, except this time matrices have to be either diagonally-dominant or symmetric. The matrix, instead, is decomposed a lower-upper decomposition (also knows as $LU$ decomposition): one that includes the diagonal and all the the elements above it are equal to zero, and one where the diagonal and the lower values are equal to zero.

## 2 Parallelization Strategies

There are two parallelization strategies for this algorithm, we'll describe them briefly:

- **Parallelization using `for` strategy:** For Jacobi and Gauss-Seidel, we have to generate implicit parallel tasks and give them constraints such as guaranteeing task dependencies are met, some directives may require a `private` clause, or making sure a value that could cause serialization stays protected to guarantee parallelization.

- **Parallelization using `task` strategy:** As an optional exercise for Gauss-Seidel, we'll define explicit tasks that perform according to the task and data decompositions. Moreover, we have to define a set of blocks/dependencies in order to guarantee the correctness of the code and its execution.

## 3 Performance evaluation

There are several ways of evaluating the performance of our strategies:

- **Strong scalability:** submitting a program to be run between 1 and 12 cores to evaluate the speed-up compared to the sequential time and the average time elapsed per core execution.
- **Task and data decompositions:** as we'll see later, each algorithm has different kinds of dependencies. So, we can't just apply the same procedures on the premise that both of these algorithms solve the same problem. We have to evaluate the original task and data decompositions and modify them if necessary to improve performance.
- **Traces:** we can see the traces of the executions of our programs, using 8 threads, to get an idea of what each thread is doing. In this deliverable we just used the default *Paraver* setup, which draws a timeline for each thread. In layman's terms, we are interested in seeing if cores as much dark blue as possible (executing code), a minimum of yellow (scheduling and fork/join) and as little red (synchronization) and white (not started yet) as possible; all of this has to happen to as many cores as possible at the same time. Also, we can look if any kind of imbalances are present, which will help us in order to find mechanisms to minimize them.

## 4 Analysis with Tareador

### 4.1 Include the relevant parts of the modified `solver-tareador.c` code and comment where the calls to the Tareador API have been placed. Comment also about the task graph generated and the causes of the dependences that appear in the two solvers: ***Jacobi*** and ***Gauss-Seidel***. How will you protect them in the parallel `OpenMP` code?

```
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("JACOBI_IM_LOOP");
                     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("JACOBI_IM_LOOP");
             }
         }
    }
    return sum;
}
```

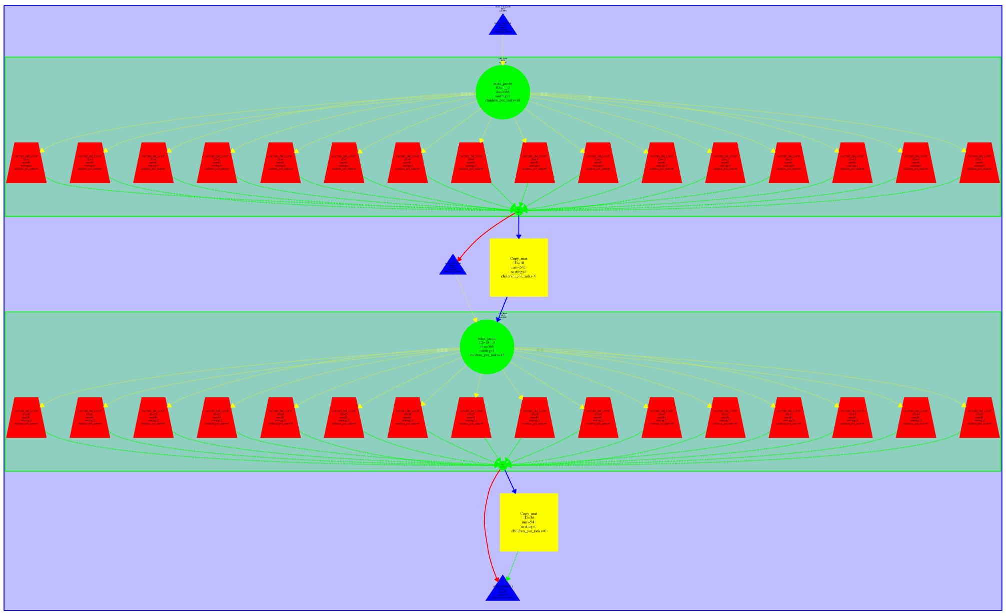


Figure 1: Ideal TDG for the Jacobi solution

For Jacobi, there are dependencies only with the `sum` variable. To protect this dependencies we can set this variable to the directive `private(diff)` to force `diff` to have different values on each thread, and `reduction(+ :  sum` for each thread, so each thread has its own sum and then, once calculations have finished, their values are summed to the original sum.

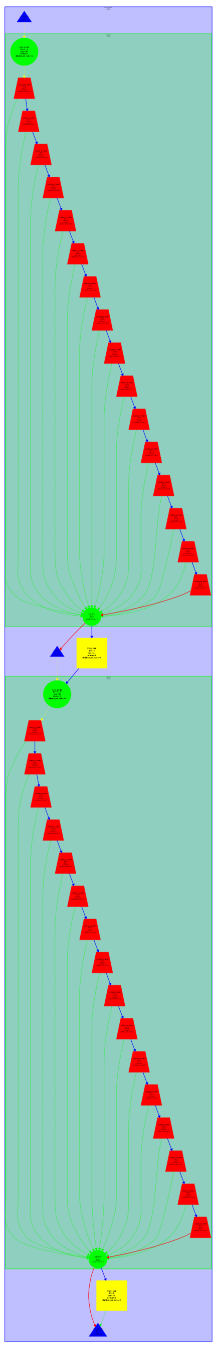


Figure 2: TDG for the Jacobi solution when taking sum into consideration

```
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("GAUSS_IM_LOOP");
            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;
            u[i*sizey+j]=unew;
            tareador_enable_object(&sum);
          tareador_end_task("GAUSS_IM_LOOP");
        }
      }
    }
    return sum;
}
```

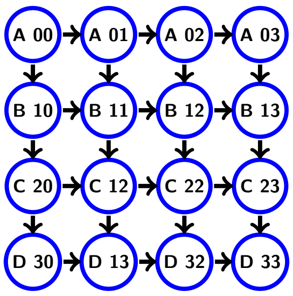


Figure 3: Data decomposition on Gauss-Seidel (in blocks of four). Note that $A_{00}$ is the starting node (source), and $D_{33}$ is the ending node (sink)

For Gauss-Seidel, instead, there are dependencies for each item at the left item and at the top item (unless no such elements exist). We protect these dependencies using the `depend (sink: i-1, j)` and `depend (source: i, j-1)` directives for the horizontal dependencies and vertical dependencies, respectively, and a source for the next threads: `depend (source)`.

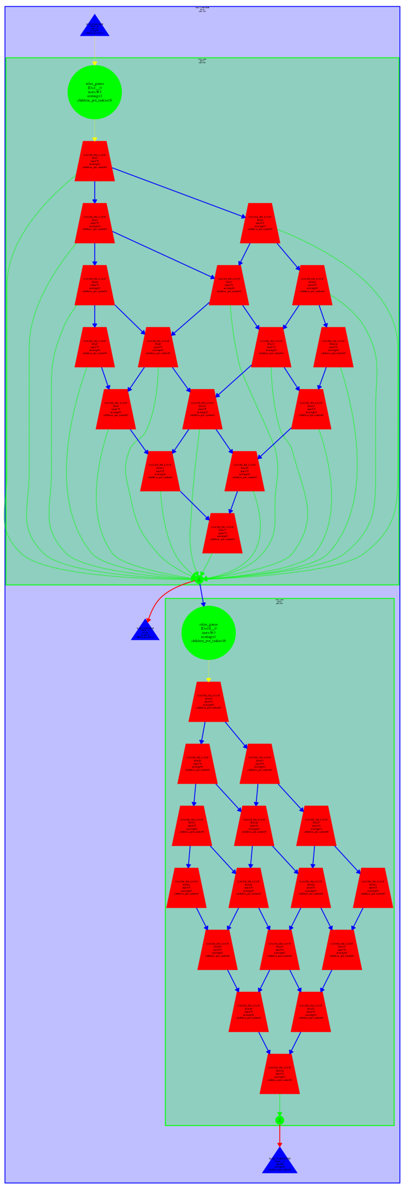


Figure 4: Ideal TDG for the Gauss-Seidel solution (sum is not processed)

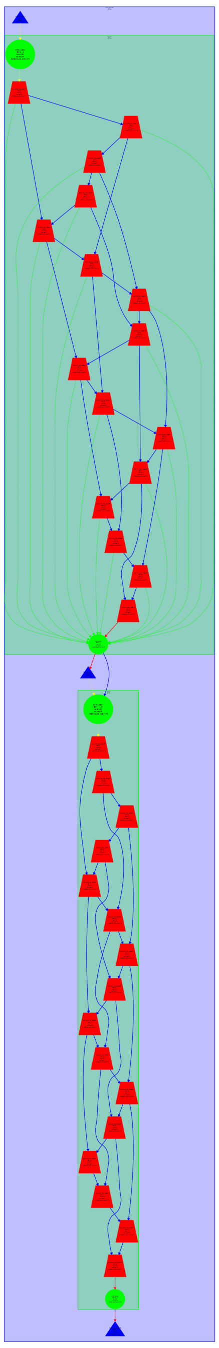


Figure 5: TDG for the Gauss-Seidel solution when taking sum into consideration

## 5 Parallelization of Jacobi with OpenMP `for`

### 5.1 Describe the data decomposition strategy that is applied to solve the problem, including a picture with the part of the data structure that is assigned to each processor.

Note that each `howmany` is assigned to a processor, and of it's assumed that there are as many `howmany` as processors at a given moment. As stated in the deliverable, the data decomposition has to be drawn with the value of `howmany` being 4:

```
    Howmany=0     Howmany=1     Howmany=2     Howmany=3

P0   X X X X       . . . .       . . . .       . . . .
P1   . . . .       X X X X       . . . .       . . . .
P2   . . . .       . . . .       X X X X       . . . .
P3   . . . .       . . . .       . . . .       X X X X
```

Data decomposition for the `howmany`, given a 4x4 matrix. Dots represent unvisited positions, and crosses ('`X`') represent the positions that will be accessed. In other words, it's a geometric data decomposition where each processor is assigned a block size of size $MatrixSize/NumProcs$.

### 5.2 Include the relevant portions of the parallel code that you implemented to solve the heat equation using the *Jacobi* solver, commenting whatever necessary. Including captures of Paraver windows to justify your explanations and the differences observed in the execution.

We use `reduction(+ :  sum)` to make sure that each thread calculates a sub-sum, which then will be accumulated into the bigger `sum` (that is, the one we want).

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

    //int howmany=4; SEE BELOW FOR EXPLANATIONS
    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;
}
```

At first, the workload is not balanced in all threads. The main bottleneck is on the `howmany` variable, which doesn't scale properly for a variable number of threads. Our idea is to use the `omp_get_max_threads()` directive to take profit of the threads available at a certain moment and scale accordingly.

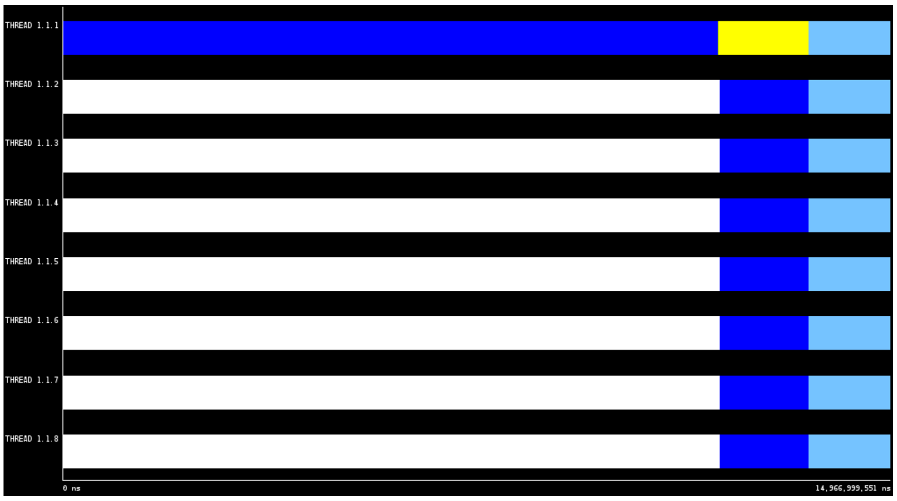


Figure 6: First version of an improved *Jacobi* trace

Now we find another issue, and is that three quarters of the execution is still stuck on the first thread. We can solve this issue by parallelizing the `copy_mat` function with a simple `#pragma omp parallel for collapse(2)` (as there are two nested loops), which will give us an improved *Jacobi* performance:

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {
    #pragma omp for collapse(2)
    for (int i=1; i<=sizex-2; i++)
        for (int j=1; j<=sizey-2; j++)
            v[ i*sizey+j ] = u[ i*sizey+j ];
}
```

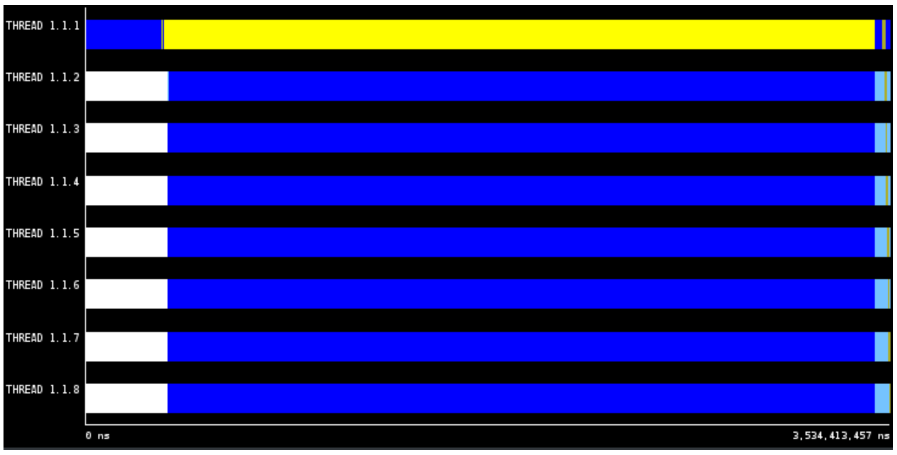
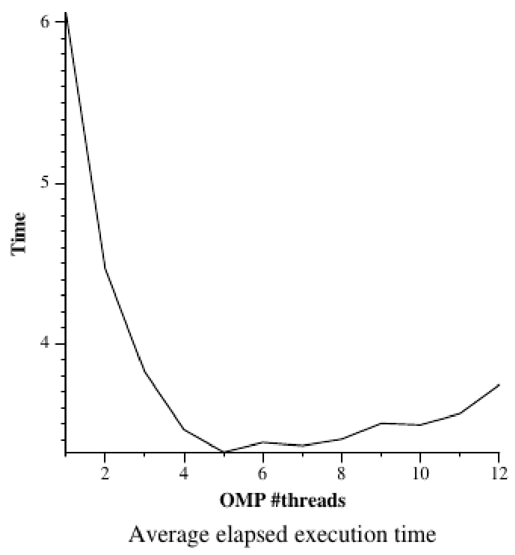


Figure 7: Final version of an improved *Jacobi* trace

## 5.3 Include the speed–up (strong scalability) plots that have been obtained for the different numbers of processors. Reason about the performance that is observed.



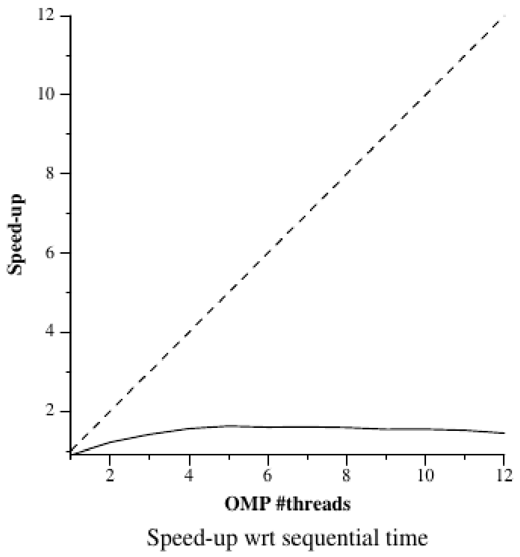


Figure 8: Strong scalability plots for *Jacobi*

Note that when there's just one thread, the execution time is really slow, as there can't be any parallelization, but once it starts executing code in parallel (number of cores = 2), time suddenly decreases, and begins increasing when the number of threads is $\geq 5$. Perhaps this is due to the forks/joins and its synchronization, which cause an overhead on the parallel computations.

Also, speedup is on a steady rate because each parallel execution is causing a linear decrease of operations per core; and then decreases slightly due to the overheads mentioned before.

## 6 Parallelization of Gauss-Seidel with OpenMP `for`

### 6.1 Include the relevant portions of the parallel code that implements the *Gauss-Seidel* solver, commenting how you implemented the synchronization between threads.

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

    #pragma omp parallel for
    for (int i = 0; i < howmany; ++i) {
        procesados[i] = 0;
    }
    int nb = 8;

    #pragma omp parallel for schedule(static) private(diff, unew)
    reduction(+ : sum)
    for (int i = 0; i < howmany; ++i) {
        int ii_upper_bounds = upperb(i, howmany, sizex);
        int ii_low_bounds = lowerb(i, howmany, sizex);
        for (int j = 0; j < nb; j++){
            int jj_upper_bounds = upperb(j, nb, sizey);
            int jj_low_bounds = lowerb(j, nb, sizey);
            if (0 < i){
                while (procesados[i-1] <= j){
                    #pragma omp flush
                }
            }

            for (int ii = max(1, ii_low_bounds); ii <= min(sizex-2, ii_upper_bound
                for (int jj = max(1, jj_low_bounds); jj <= min(sizey-2, jj_upper_b
                    unew = 0.25* (u[ii * sizey + (jj-1)] +  // left
                                  u[ii * sizey + (jj+1)] +  // right
                                  u[(ii-1) * sizey + jj] +  // top
                                  u[(ii+1) * sizey + jj]); // bottom
                    diff = unew - u[ii * sizey + jj];
                    sum += diff*diff;
                    u[ii*sizey+jj] = unew;
                }
            }
            ++procesados[i];
            #pragma omp flush
        }
    }
    return sum;
}
```

We have a `procesados` vector that stores the information about which thread-blocks are already computed. The matrix is divided in `howmany` pieces (or blocks), which is the maximum number of threads available. Each thread starts processing at `ii_lower_bounds` and ends at `ii_upper_bounds`, which are calculated using the `lowerb()` and `upperb()` function macros given, respectively.

Each threadwise divided block is then divided in sub-blocks, 8 in this case, traversed by the `for(int j = 0; ...)` loop.

The `ii_lower_bounds - ii_upper_bounds` space delimits the whole space the thread is going to process and the `jj_lower_bounds - jj_upper_bounds` delimits the block that is going to be processed in this $j$ iteration by this processor $i$.

At the start of the $jj$ block we do a `#pragma omp flush` in order force consistency on memory and get the previous row/column information needed to calculate this block from the other threads that may have it (note that all threads can see these memory values, that's why we need to flush!). Then we get to the $ii - jj$ double for loop which is the one who really calculates the data in the end. After this processing is done, we store a `1` in the corresponding `procesados` vector position and we do a `flush`. This way, we ensure that by dividing the rows of the matrix to all threads equally we get a balanced task (and data) decomposition.

## 6.2 Include the speed–up (strong scalability) plot that has been obtained for the different numbers of processors. Reason about the performance that is observed, including captures of `Paraver` windows to justify your explanations.

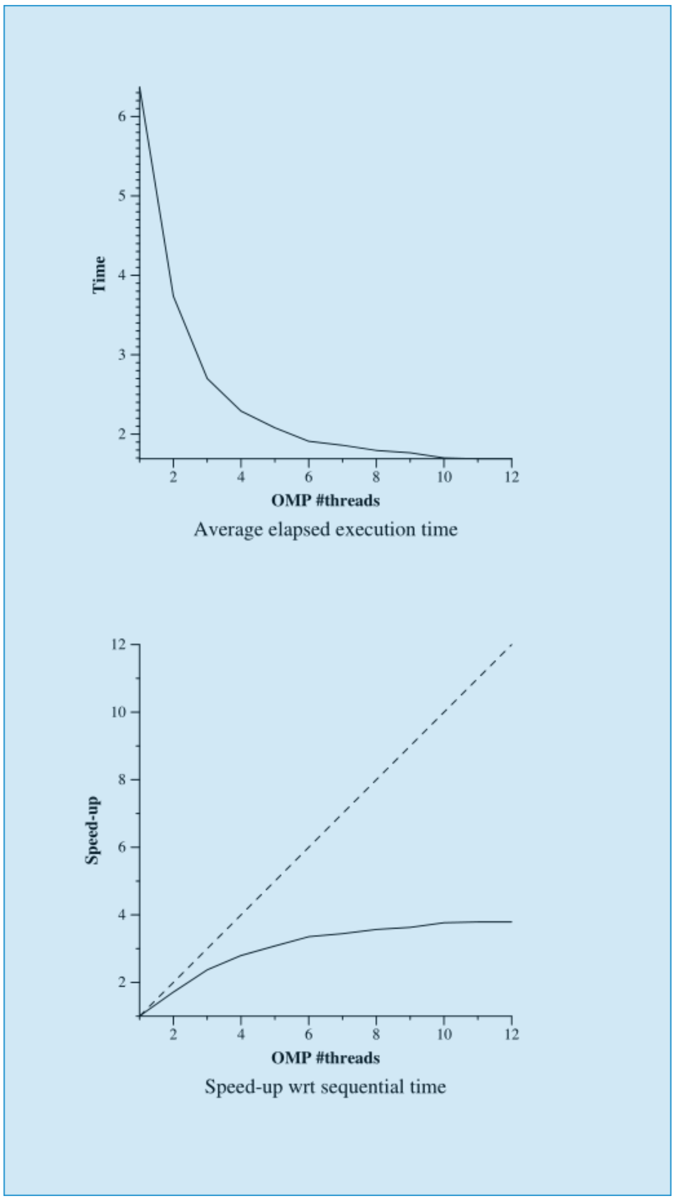


Figure 9: Speed-up for the *Gauss* − *Seidel* algorithm

This plot shows a pseudo-exponential decay on execution time per thread, that is, the execution of the program is strictly better each time. Speed-ups are always incremental, but in a pseudo-logarithmic way. It seems the maximum speedup it can go is bounded at 4.

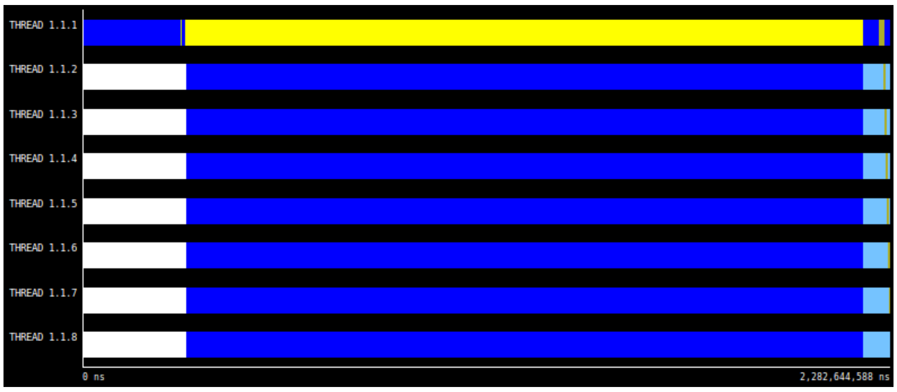


Figure 10: Paraver trace for the $Gauss - Seidel$ algorithm, with `nb` set to 8

## 6.3 Explain how did you obtain the optimum value for the ratio computation/synchronization in the parallelization of this solver for 8 threads.

| Value of `nb` | **Time (in seconds)** |
|---|---|
| 2 | 3.908 |
| 4 | 2.482 |
| 8 | 1.747 |
| 16 | 1.310 |
| 32 | 1.213 |
| 64 | 1.257 |
| 128 | 1.772 |
| 256 | 2.350 |

We obtained the value by experimentation, as seen in the table above: we set a new value `nb` that controls the block size of the algorithm, and we submitted to boada the Gauss solver with `nb` with values ranging from $2^2$ up to $2^8$. The optimum value for this computation/synchronization is $2^5$ =32, which has the shortest time overall, but numbers surrounding it (range from 16 to 64) produce good values as well.

## 6.4 *OPTIONAL*: Implement an alternative parallel version for ***Gauss-Seidel*** using `#pragma omp task` and task dependencies to ensure their correct execution. Compare the performance against the `#pragma omp for` version and reason about the better or worse scalability observed.

We need to add an auxiliary matrix to set the task dependencies of the code. The matrix has no calculations inside, it's just used to ensure that the `depend` directives are met properly.

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

    //Block number
    int nb = 8;

    // Aux matrix for dependencies, data doesn't matter (use char for less space u
    char auxDeps[howmany][howmany];

    #pragma omp parallel private(diff, unew)
    #pragma omp single
    for (int i = 0; i < howmany; ++i) {
        int ii_low_bounds = lowerb(i, howmany, sizex); //Lower bound for ii
        int ii_upper_bounds = upperb(i, howmany, sizex); //Lower bound for jj
        for (int j = 0; j < nb; j++){
            int jj_low_bounds = lowerb(j, nb, sizey);
            int jj_upper_bounds = upperb(j, nb, sizey);


        #pragma omp task depend(in: auxDeps[i-1][j])  depend(in: auxDeps[i][j-1])
        {
            double sum2 = 0.0;
            for (int ii = max(1, ii_low_bounds); ii <= min(sizex-2, ii_upper_bound
                    for (int jj = max(1, jj_low_bounds); jj <= min(sizey-2, jj_uppe
                        unew = 0.25 * (u[ii * sizey + (jj-1)] +  // left
                                      u[ii * sizey + (jj+1)] +  // right
                                      u[(ii-1) * sizey + jj] +  // top
                                      u[(ii+1) * sizey + jj]); // bottom
                        diff = unew - u[ii * sizey + jj];
                        sum2 += diff*diff;
                        u[ii*sizey+jj] = unew;
                    }
                }
                #pragma omp atomic
                sum += sum2;
            }
        }
    }
    return sum;
}
```

From testing we can see the performance is way worse in the `task` version than in the `for` one, due to the additional overheads on task dependencies and on waiting on the `atomic` directive. It's so slow that not only it isn't a viable alternative to using the `for` clause at all, the extra tasks it has to do (forks/joins, sync overheads, waiting...) makes it run worse than the serial execution!

## 7 Conclusions

When we saw the Tareador TDG we could see that the Jacobi solution was much more parallelizable than the $Gauss - Seidel$ one. Even though it may seem the best solution by far, it represents a huge space problem, as we duplicate a matrix and it starts to get nasty as the number of elements rise.

If we don't take that into account, we could consider both solutions valid looking at the performance, as the speed-up plots showed that around 8 threads reduces the execution time considerably.

The problem in both solutions was treating the dependency involving the sum variable:

- The *Jacobi* one was far easier to work on, as it only needed to be added atomically, which can be solved easily. In other words, our only worry was to make sure the sum of each parallel process sum into the main one.

- On the other hand, the $Gauss - Seidel$ solution had dependencies that involved other iterations and we needed to first modify the code to transform how the data is processed into the problem (as we specified earlier), then flush the processed positions in the matrix in order to obtain the data that we needed.

In conclusion, as we have to take everything into consideration, we would prefer using the $Gauss - Seidel$ solution because it's performance is faster and scales better the more processors it has (unlike $Jacobi$), but both algorithms in their parallel versions are very solvent when there are not too many threads available.