PAR Laboratory Assignment

Lab 5: Geometric (data) decomposition using implicit tasks: heat diffusion equation

31/12/2021

Fall 2021-22

Index

1. Introduction

In this final laboratory assignment, we are going to work on the parallelisation of a sequential code that simulates the diffusion of heat in a solid body using two different solvers for the heat equation: Jacobi and Gauss- Seidel.

Each solver has different numerical properties which are not relevant for the purposes of this laboratory assignment; we use them because they show different parallel behaviours.

The difference between the two solvers is that Jacobi uses a temporary matrix to store the new computed matrix while Gauss-Seidel directly updates the same matrix.

2. Sequential heat diffusion program and analysis with Tareador

We are going to start by executing the sequential version of the program for both algorithms and comparing the results.

Executing the program using the Jacobi solver shows us the output of figure 2.1 and the image on the left of figure 2.3. When we execute it with the Gauss-Seidel solver it shows us the output of figure 2.2 and the image on the right of figure 2.3.

```
par2119@boada-1:~/lab5$ ./heat test.dat -a 0 -o heat-jacobi.ppm
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 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: 4.584
Flops and Flops per second: (11.182 GFlop => 2439.53 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Figure 2.1 – Execution output using the Jacobi solver

```
par2119@boada-1:~/lab5$ ./heat test.dat -a 1 -o heat-gauss.ppm
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 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: 6.124
Flops and Flops per second: (8.806 GFlop => 1437.98 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

Figure 2.2 – Execution output using the Gauss-Seidel solver

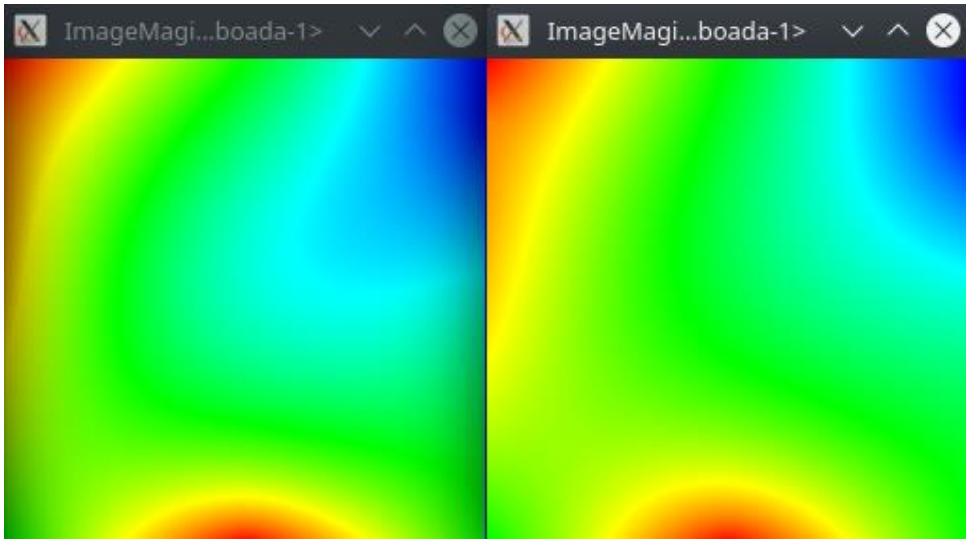


Figure 2.3 - Image file generated with Jacobi solver (left) and Gauss-Seidel solver (right)

At first glance we can see that the images generated by both algorithms are slightly different. We can check it with the diff command (Figure 2.4).

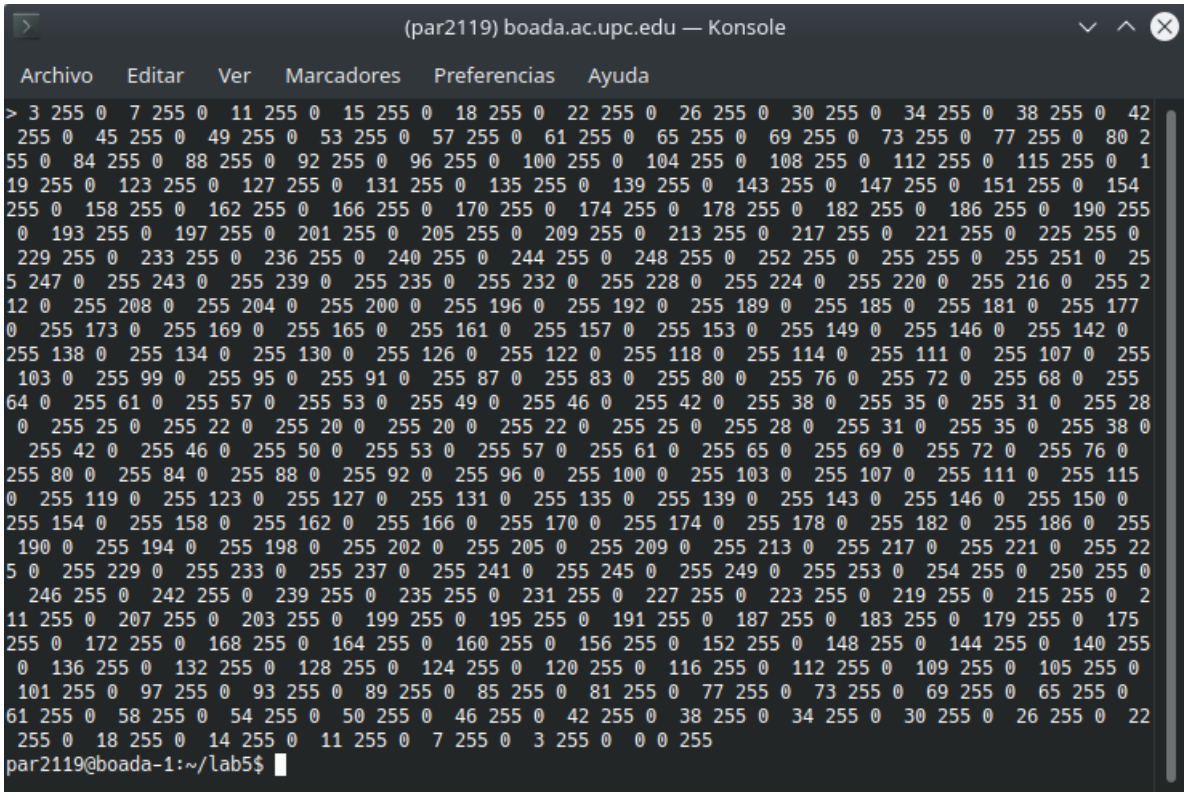


Figure 2.4 - Result of executing the command: ***diff heat-gauss.ppm heat-jacobi.ppm***

Now we are going to use Tareador to analyse the task graphs generated when using the two different solvers.

We compile the heat-tareador.c program that is initially provided and execute it for each solver to obtain their corresponding task graphs (Figure 2.5).

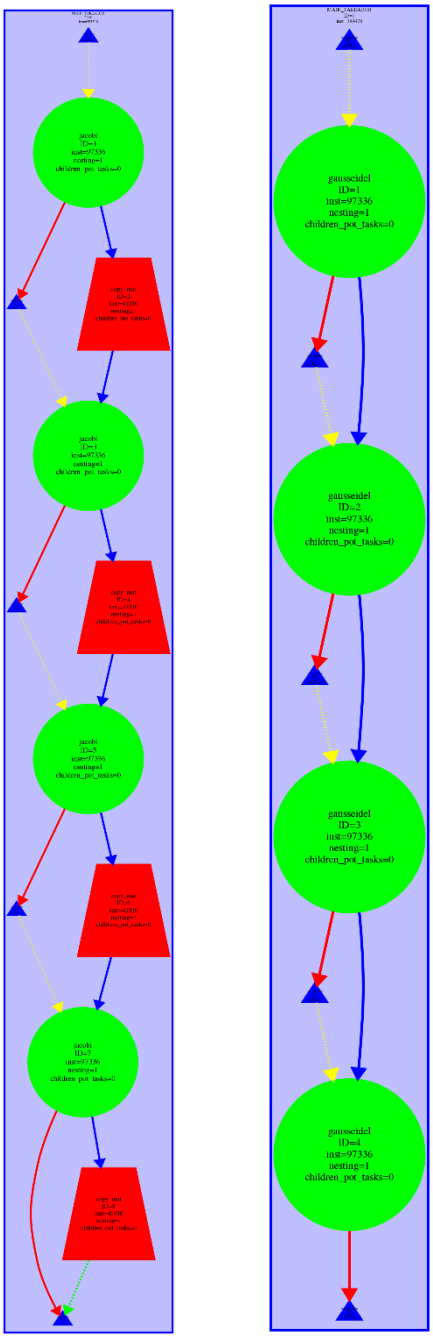


Figure 2.5 - Jacobi dependency graph (left) and Gauss-Seidel dependency graph (right)

As we can see in Figures 2.5, for both algorithms, there is hardly any parallelization possible, for the case in which we use Jacobi, the parallelization is almost zero while in the Gauss-Seidel one there is not even.

We have modified the given code to add the level of graininess we are looking for: **one task per block**.

```
for (int blocki=0; blocki<nblocksi; ++blocki) {
  int i_start = lowerb(blocki, nblocksi, sizex);
  int i_end = upperb(blocki, nblocksi, sizex);
  for (int blockj=0; blockj<nblocksj; ++blockj) {
      tareador_start_task("task_per_block");
    int j_start = lowerb(blockj, nblocksj, sizey);
    int j_end = upperb(blockj, nblocksj, 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++) {
          tmp = 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 = tmp - u[i*sizey+ j];
        sum += diff * diff;
        unew[i*sizey+j] = tmp;
      }
    }
    tareador_end_task("task_per_block");
  }
}
```

After the previous modification, we obtain the new dependency graphs for each solver (Figure 2.6).

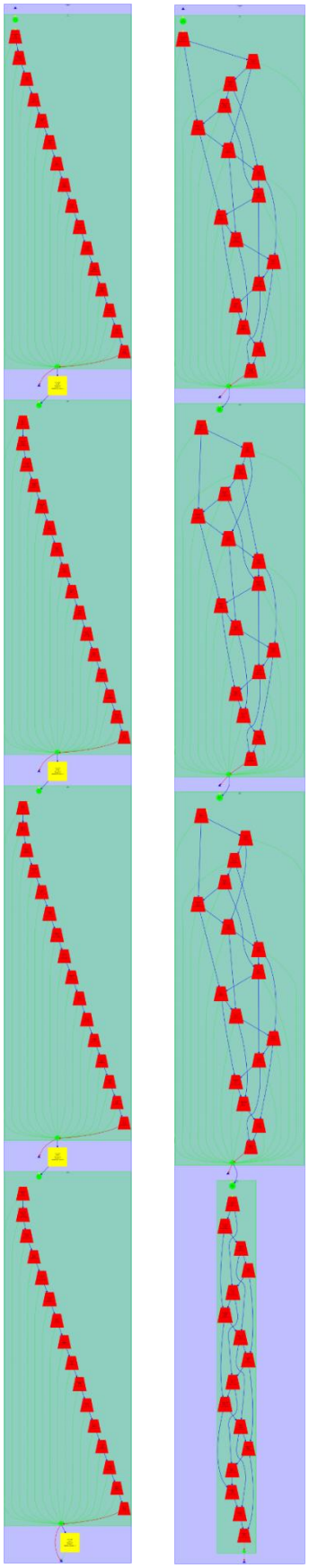


Figure 2.6 - Jacobi dependency graph (left) and Gauss-Seidel dependency graph (right)

As we can see in figure 2.6, now we have many more tasks in both task dependency graphs, but we still have a problem that prevents us from parallelizing the execution. In both cases there are many dependencies between the tasks.

If we use the Dataview option in Tareador we can identify the variable that causes dependencies between tasks (Figure 2.7).

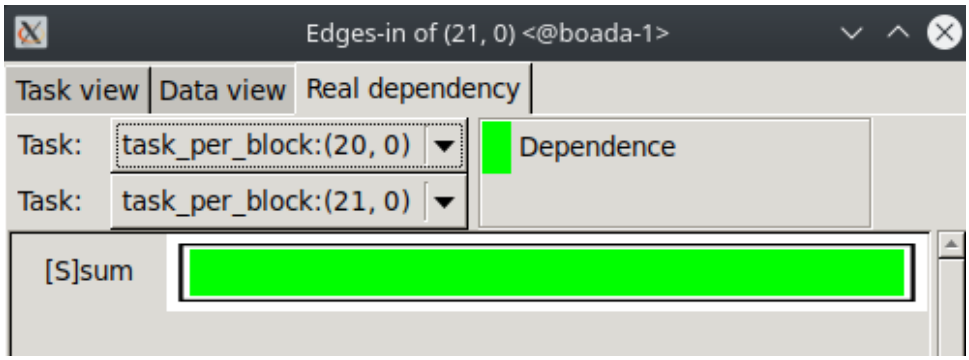


Figure 2.7 - Dataview option in Tareador

To solve this problem between tasks we have to protect the dependencies caused by this variable. To achieve this, we are going to uncomment the *tareador_disable_object* and *tareador_enable_object* calls, that is already provided to us in the code as comments.

Once the code has been modified, we get the task dependency graphs once more (Figure 2.8 and Figure 2.9).

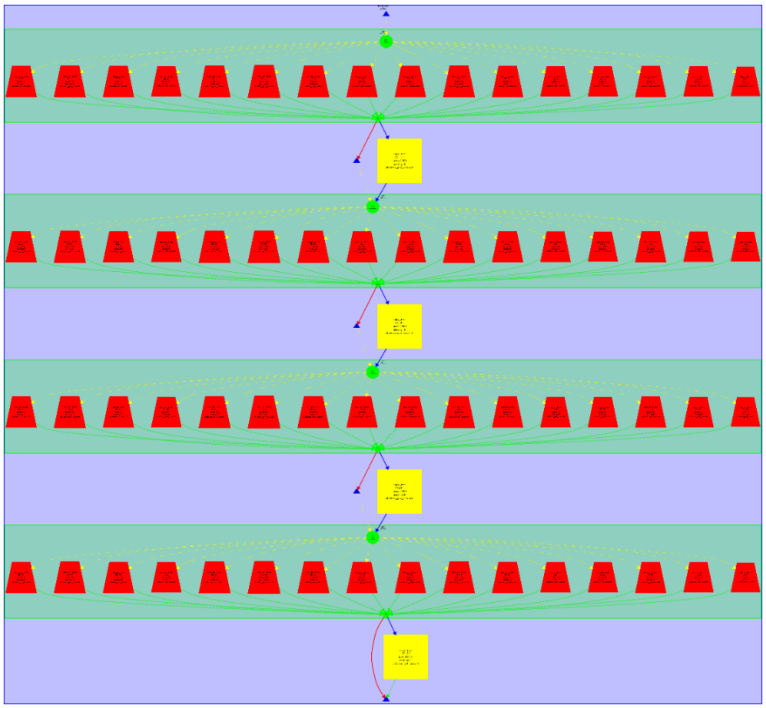


Figure 2.8 - Jacobi dependency graph

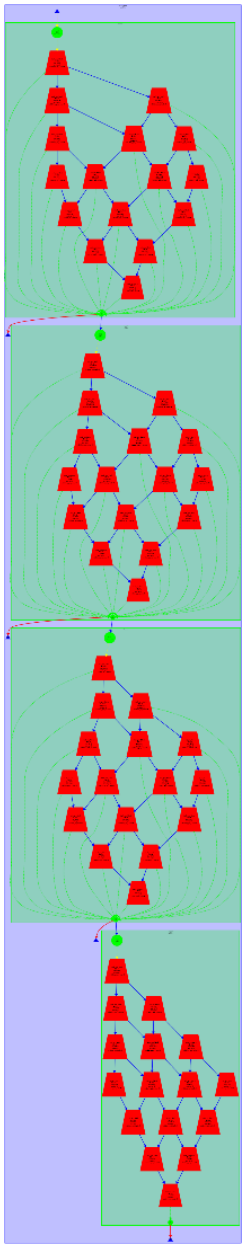


Figure 2.9 - Gauss-Seidel dependency graph

With this change, both solvers show an important improvement in the parallelization of the program (Figure 2.8 and Figure 2.9).

To protect access to the sum variable (the cause of dependencies between tasks) in our implementation we can use several options that OpenMP offers us. In Jacobi's case we can use atomic, critical or reduction, this last is the best option, using *reduction (+: sum)* after the *#pragma omp for*.

In the case of Gauss-Seidel we can use the unordered clause for the *#pragma omp for*. We would use the OpenMP clause *depends* if using *#pragma omp task*.

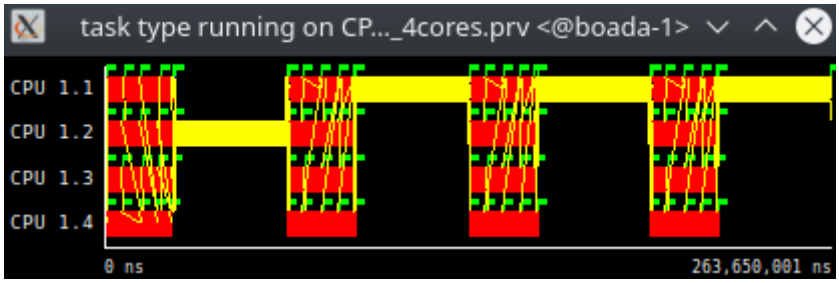


Figure 2.10 - Simulation of Jacobi execution using 4 processors

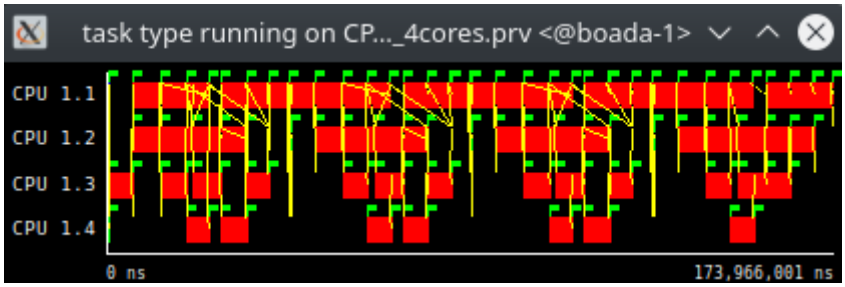


Figure 2.11 - Simulation of Gauss-Seidel execution using 4 processors

3. Parallelisation of the heat equation solvers

3.1. Jacobi solver

To do this part we parallelize the loop, mark the variable diff as private to create a new variable inside the construct and add a reduction clause to variable sum.

The parallelized Jacobi solver code:

```
// 2D-blocked solver: one iteration step
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
    double tmp, diff, sum=0.0;

    #pragma omp parallel private(diff) reduction(+:sum)
    {
    int nblocksi = omp_get_num_threads();
    int blocki = omp_get_thread_num();
    int i_start = lowerb(blocki, nblocksi, sizex);
    int i_end = upperb(blocki, nblocksi, sizex);
    for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++) {
        for (int j=1; j<=sizey-2; j++) {
            tmp = 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 = tmp - u[i*sizey+ j];
            sum += diff * diff;
            unew[i*sizey+j] = tmp;
        }
    }
    }
    return sum;
}
```

Once we validated the new code, we submitted the execution of submit-strong-omp.sh script to obtain the scalability plot for different number of processors (1 to 12), specifying with an argument the solver to be used, in this case 0 because we want to use Jacobi solver, the result can be seen in figure 3.1.1.

The scalability that we obtained with this initial parallelisation its appropriate to the part of the code that we parallelized, but is a poor performance, to increase it we need to parallelize the copy_mat function as well, because there is a serious serialisation in that part of the code.

The parallelized copy_mat code:

```
// Function to copy one matrix into another
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {
    int nblocksi = omp_get_max_threads();

    # pragma omp parallel
    {
    int blocki = omp_get_thread_num();
    int i_start = lowerb(blocki, nblocksi, sizex);
    int i_end = upperb(blocki, nblocksi, 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];
    }
}
```

If we submit the execution of submit-strong-omp.sh and we compare with the previous version:

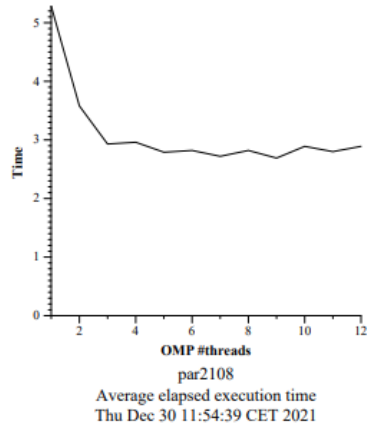
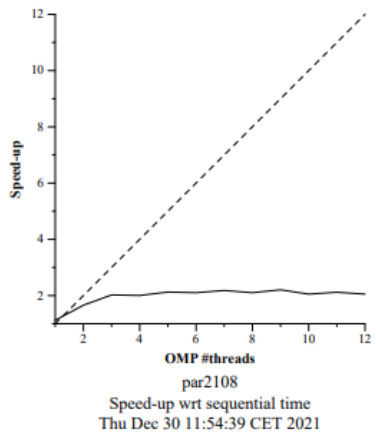


Figure 3.1.1 – Scalability with parallelized function solve



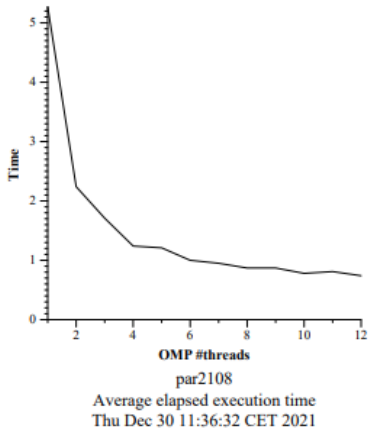
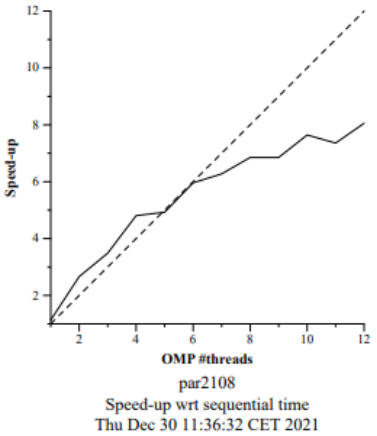


Figure 3.1.2 – Scalability with parallelized function solve and copy_mat



The improvement is noticeable, that is due after the jacobi solve function we copy one matrix to other, this is done by copy_mat function, once this function is parallelized the speed-up increases.

If we use *Paraver* we can see how the code is parallelized, we can observe both versions of the parallelized codes, notice that the second version we obtain a better time.

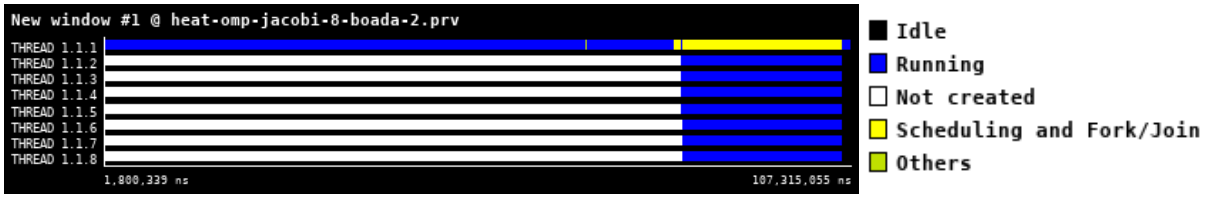


Figure 3.1.3 – Paraver with parallelized Jacobi solver

Figure 3.1.4– Paraver with parallelized Jacobi solver and copy_mat

3.2. Gauss–Seidel solver

To identify the solver that we must implement, we use the variable *u* and *unew*, by checking if *u* is equal to *unew*, if this comparison returns true then we have to use the Gauss-Seidel solver otherwise the Jacobi solver. Due to the dependencies between iterations we decided to use a block distribution, other option would be a row decomposition, but implementing a block decomposition we avoid the dependencies on the i – 1 and j – 1 elements.

In the following code we traverse over the block establishing the bounds of each block, and we make use of the ordered clause, different threads executes concurrently until they run into the ordered region, then the rest of the code is executed sequentially, which allows us some parallelization of the code.

The parallelized Gauss-Seidel solver code:

```
// 2D-blocked solver: one iteration step
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
    double tmp, diff, sum=0.0;

    if ( u==unew ) {
        int nblocksi=omp_get_max_threads();

        #pragma omp parallel for ordered(2) private(diff) reduction(+:sum)
        for (int ii = 0; ii < nblocksi; ii++) {
            for (int jj = 0; jj < nblocksi; jj++) {
                int i_start = lowerb(ii, nblocksi, sizex);
                int i_end = upperb(ii, nblocksi, sizex);
                int j_start = lowerb(jj, nblocksi, sizex);
                int j_end = upperb(jj, nblocksi, sizex);
                #pragma omp ordered depend (sink: ii-1, jj)
                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++) {
                    tmp = 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 = tmp - u[i*sizey+ j];
                    sum += diff * diff;
                    unew[i*sizey+j] = tmp;
                    }
                }
                #pragma omp ordered depend(source)
            }
        }
    }
...
```

If we take a look at the figure 3.2.1, we can see the callability of the previous code with different number of threads.

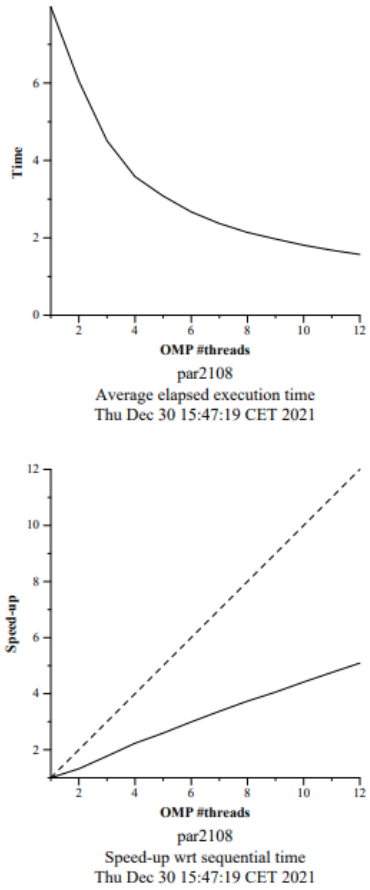


Figure 3.2.1 – Scalability with parallelized Gauss-Seidel solver

We notice that the speed-up increases but is not a spectacular improvement.

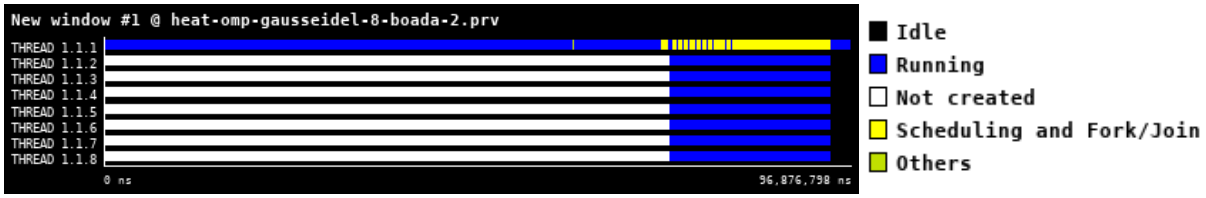


Figure 3.2.2 – Paraver with parallelized Gauss-Seidel solver

The above figure shows us the execution of Gauss-Seidel solver parallelized function with 8 threads using the *Paraver* tool. The parallel execution can be seen in yellow colour in the first thread, also how split the tasks with the other threads.

To exploit more parallelism in the execution of the solver we changed the number of vlocks in the j dimension, that changes the ratio between computation and synchronisation. To do that we changed a few things of the previous code, we initialize a variable named *nblocksj* with the value of the variable *userparam*, that is received by adding -u value when heat-omp is executed, and replacing the variable nblocksi by nblocksj when we refer to the columns (j).

The modified part of the code:

```
...
int nblocksj = userparam;
...
for (int jj = 0; jj < nblocksj; jj++) {
                ...
                int j_start = lowerb(jj, nblocksj, sizex);
                int j_end = upperb(jj, nblocksj, sizex);
...
```

To explore a range of values for this argument we used submit-userparam-omp.sh script with 4 and 8 threads.

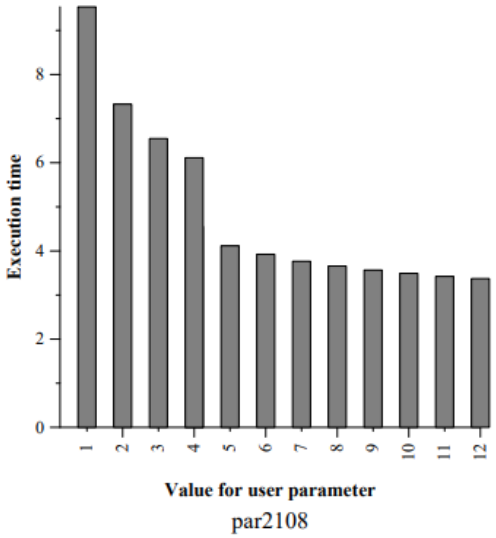


Figure 3.2.3 – `submit-userparam-omp.sh` plot with 4 threads

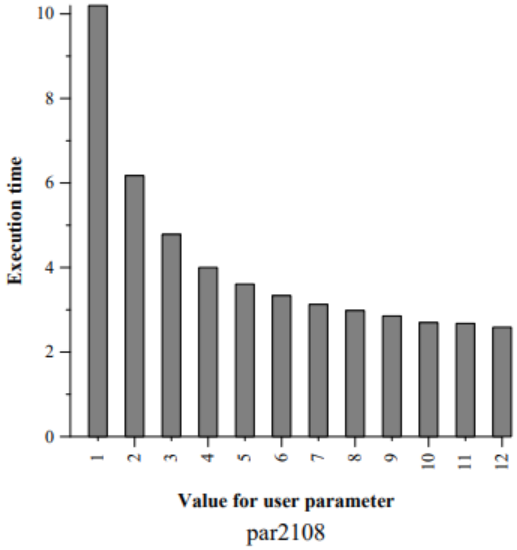


Figure 3.2.3 – `submit-userparam-omp.sh` plot with 8 threads

Notice how the execution time varies when the userparam value changes, that is because the number of tasks increases, so we parallelized more the code, but also we create blocks that not depend on the number of threads that we have and they have to synchronize.

4. Conclusions

Throughout this lab assignment we have studied and analysed the different possibilities for parallelization of two solvers for the heat equation: Jacobi solver and Gauss-Seidel solver.

In conclusion, there are several techniques that allow us to parallelize a code, but for each problem there are some that are better adapted than others. Therefore, it is of great importance to understand the code of the program and its operation, and especially the dependencies, to obtain the most efficient solution.

For example, in the case of the Jacobi solver the data decomposition technique it's a good option while on the other hand for the Gauss-Seidel solver this solution is not optimal due to data dependency.