PAR Laboratory Deliverable

Course 2019/20 (Fall semester)

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

Rafel-Albert Bros Esqueu, Joan Vinyals Ylla-Català

User: Par2201

October, 2019

Contents

Chapter 1

Introduction

In this laboratory project we will work on the parallelization of a sequential code (*heat.c*) which simulates heat diffusion in a solid body. As we wish to expand our knowledge two different solvers for the heat equation will be used: the ***Jacobi*** solver and the ***Gauss-Seidel*** solver. Each of them has different numerical properties and therefore a different parallel functioning that will be explored throughout this deliverable.

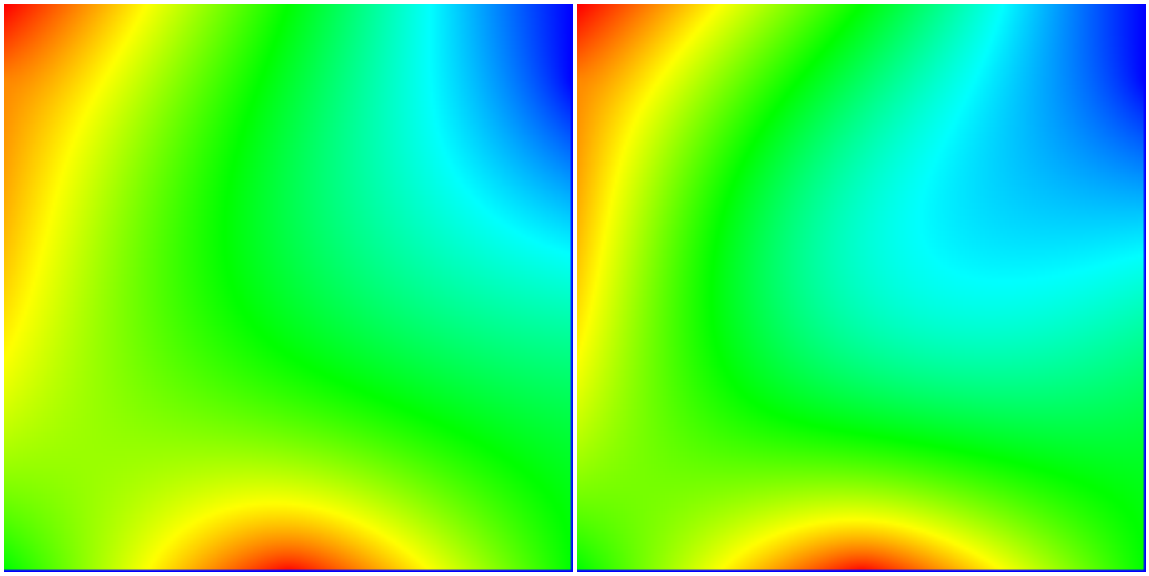

Figure 1.1: Images showing the solution as a gradient from red (*hot*) to dark blue (*cold*) for the *Jacobi* solver (right) and the *Gauss-Seidel* solver (left).

Chapter 2

Analysis of task granularities and dependences

In order to analyze the characteristics of the two different solvers we will use the task graphs generated with *Tareador*. The following compilation and execution instructions have been used, changing each time the configuration file to use each of the solvers.

```
make heat-tareador
./run-tareador.sh heat-tareador small
    .dat
```

2.1 Task graphs analysis with *Tareador*

Now we will explore the dependencies that happen when a much finer-grain task decomposition is used. The code is modified to make sure the program creates a different task per every iteration of the body of the innermost loop.

As always, the best way to study the behaviour of the newly developed code is to take a look into the task dependency graphs (TDG). Refer to *Figure 2.1* for the TDG for both solvers.

Notice how we are facing a situation with lots of dependencies between tasks. In fact, for the *Jacobi* solver the execution is completely sequential.

Thanks to the *Dataview* option in *Tareador* we identified that the accesses to the variable `sum` are the reason for the serialization of all tasks. As a solution, we have used the calls seen in *Figure 2.3* to temporarily filter the analysis for the `sum` variable.

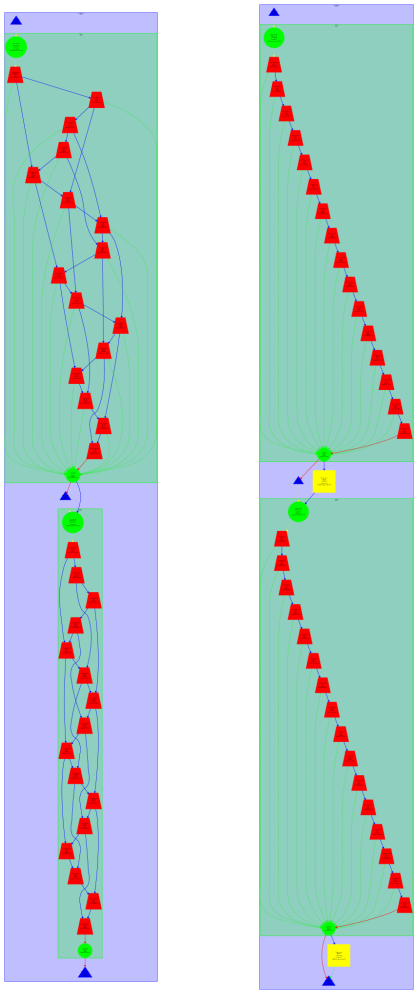

Figure 2.1: TDG for the *Jacobi* solver (right) and the *Gauss-Seidel* solver (left).

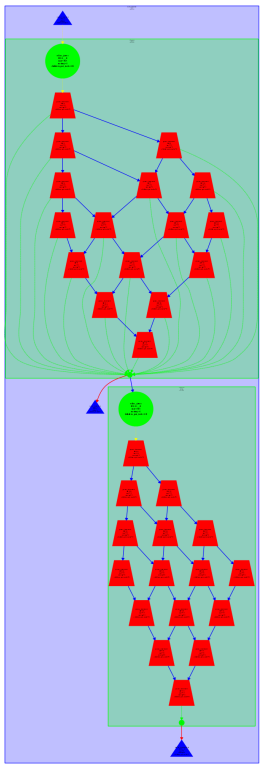

Figure 2.2: TDG of the *Gauss-Seidel* solver with the `sum` variable filtered.

```
tareador_disable_object(&sum);
sum += diff * diff;
tareador_enable_object(&sum);
```

Figure 2.3: Calls necessary to temporarily filter the analysis for the `sum` variable.

The new task graphs obtained are found in *Figure 2.2* (for the *Gauss-Seidel* solver and in *Figure 2.4* for the *Jacobi* solver.

They show an important improvement in the parallelization of the program: we no longer have dependencies related to the `sum` variable although we still observe an important amount of dependencies related to the global access order into the matrix.

As a solution for the *Jacobi* solver, `OpenMP` offers several possibilities, we can use `critical`, `atomic` and `reduction`. The optimal choice would be `reduction(+:sum)`, after the `#pragma omp for`.

On the other hand, as a solution for the *Gauss-Seidel* solver using the `unordered` clause for the `#pragma omp for`. We would use the `OpenMP` clause *depends* if using `#pragma omp task`.

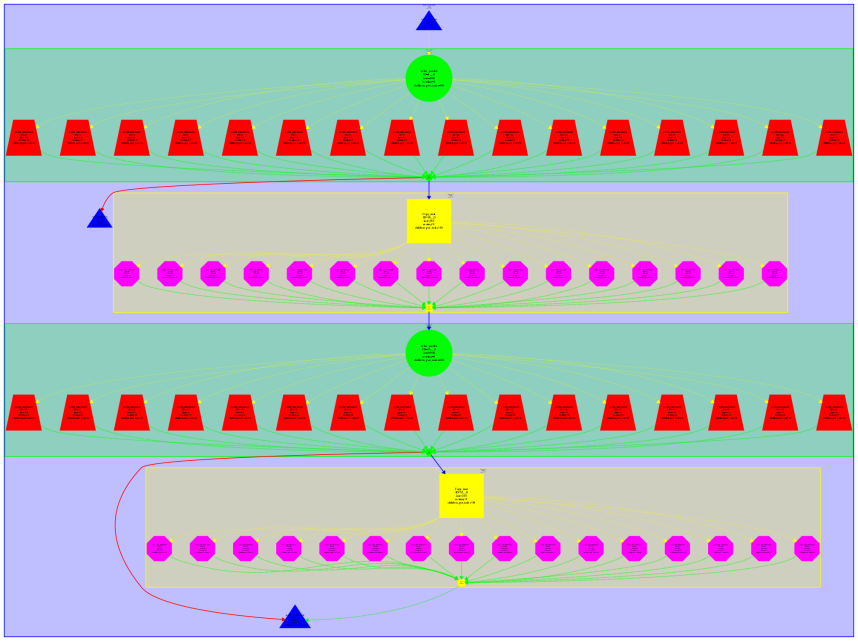

Figure 2.4: TDG of the *Jacobi* solver with the `sum` variable filtered.

Chapter 3

OpenMP parallelization and execution analysis: *Jacobi*

In this chapter we will parallelize the sequential code of the *Jacobi* solver.

Our parallel version is based on the programs `solver.c` and `heat.c` which are respectively copied and later on modified into `solver-omp.c` and `heat-omp.c`.

After understanding the macros defined in `heat.h` that are used in `solver-omp.c` we drew an example of the geometric data decomposition generated when the macros are used in the *Jacobi* solver for `howmany=4` (image seen in *Figure 3.1*).

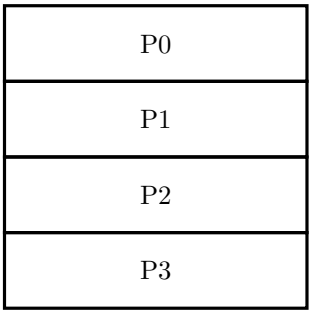

Figure 3.1: Block data decomposition for *howmany=4*.

3.1 Initial Parallelization of the code

We are now going to start parallelizing the code by specifically modifying the function `relax_jacobi`. We have implemented a `#pragma omp parallel`, specifying the variable `diff` as private and we use a `#pragma omp atomic` before modifying the `sum` variable for reasons explained in previous chapters. However, as atomic is too restrictive, performance is not good enough. As a result, we prefer to design a custom reduction as seen in *Figure 3.3*.

To make sure the parallel version works correctly, we have made a `diff` between the heat map generated with the sequential version and the one generated with the parallel version; the result is that there is no difference at all. Both heat maps have been attached in *Figure 3.2*.

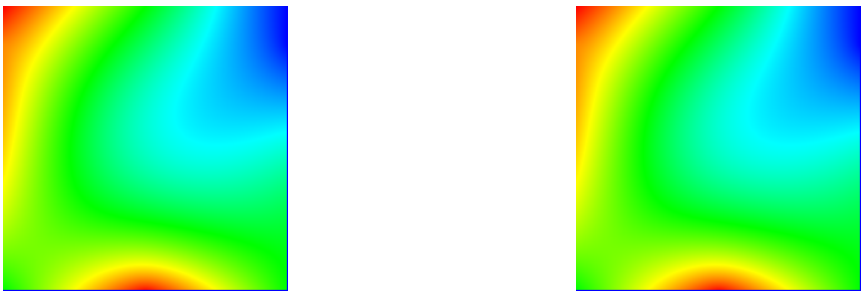

Figure 3.2: Heat maps for the sequential version (left) and the parallel version (right).

```
  double diff;
  int howmany=omp_get_max_threads();
  double *sum = calloc( howmany, sizeof(double));

# pragma omp parallel private(diff)
  {
    int blockid = omp_get_thread_num();
    int i_start = lowerb(blockid, howmany, sizex);
    int i_end = upperb(blockid, howmany, sizex);
    for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
      for (int j=1; j<= sizey-2; j++) {
        utmp[i*sizey+j]= 0.25 * ( u[ i*sizey     + (j-1) ]+  // left
            u[ i*sizey     + (j+1) ]+  // right
            u[ (i-1)*sizey + j     ]+  // top
            u[ (i+1)*sizey + j     ]); // bottom
        diff = utmp[i*sizey+j] - u[i*sizey + j];
        sum[blockid] += diff * diff;
      }
    }
  }

  double ret_val = 0.0;
  for (int k = 0; k < howmany; k++) ret_val += sum[k];
  return ret_val;
```

Figure 3.3: Code of the parallelized version of the function `relax-jacobi`.

3.1.1 Instrumentation with *Extrae*

In order to further improve the performance of the *Jacobi* solver, we have used Extrae and Paraver to better understand the behaviour of our first approach to parallelize this solver.

Whereas there is an improvement of the performance of the Jacobi solver, it is very mild; this is due to the serial execution of the *copy_mat* function, which is the bottleneck of the solver, hence creating a load imbalance.

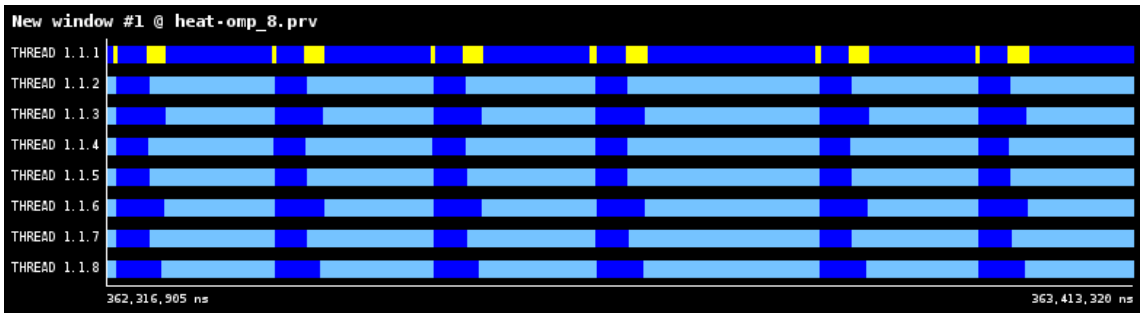

Figure 3.4: Trace of the execution of the final version of *Jacobi* solver obtained with *Paraver*.

Notice in *Figure 3.4* how considerable the load imbalance is, this could be expected from what we already have observed in the TDG in *Figure 2.4* where it appears clear the highly parallel nature of the function. To solve this we will apply the same technique of data decomposition to this function, as we explain in the following section.

3.2 Final parallelization process

After the initial parallelization, we have realised there is still room for improvement by parallelizing other parts of the code. We will now modify the function `copy_mat`. The final result can be seen in *Figure 3.5*.

```
1   int howmany=omp_get_max_threads();
2
3 # pragma omp parallel
4   {
5     int blockid = omp_get_thread_num();
6     int i_start = lowerb(blockid, howmany, sizex);
7     int i_end = upperb(blockid, howmany, sizex);
8     for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++)
9       for (int j=1; j<= sizey-2; j++)
10        v[ i*sizey+j ] = u[ i*sizey+j ];
11  }
```

Figure 3.5: Code of the parallelized version of the function `copy_mat`.

As now the function *copy_mat* is parallelized, there has been a huge reduction of load imbalance. Therefore, the resulting trace in *Figure 3.6* reveals that now most of the time all the threads are doing relevant computation.

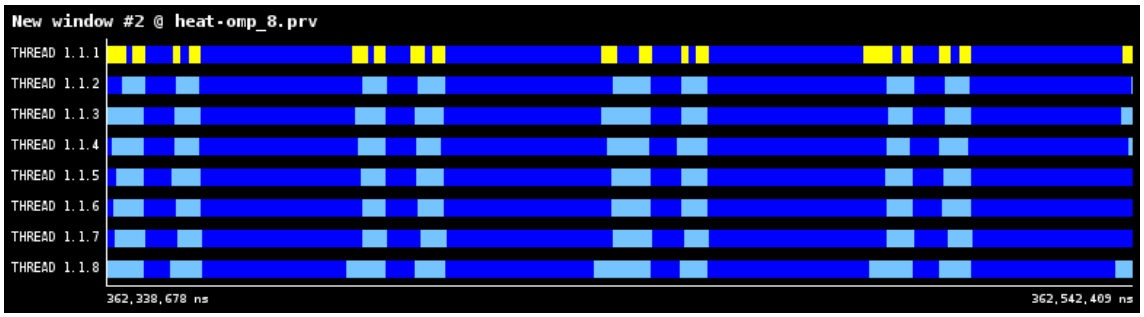

Figure 3.6

3.2.1 Scalability analysis

Finally, we will analyze the speed-up and execution time plots that have been obtained for different number of processors. Refer to *Figure 3.7* for it. The two plots on the left of the figure are the ones obtained from the initial parallel version, in other words, the version developed before the function *copy_mat* has been parallelized. The two plots on the right of the figure are obtained from the version that has a parallelized *copy_mat* function.

By looking into the plots it is obvious that when increasing the number of threads there is a huge reduction of the execution time for the second version, while there is little improvement for the initial version. Consequently, the speed-up plots show similar results, with the initial version keeping a constant speed-up while the second version achieves a speed-up noticeably close to the ideal one.

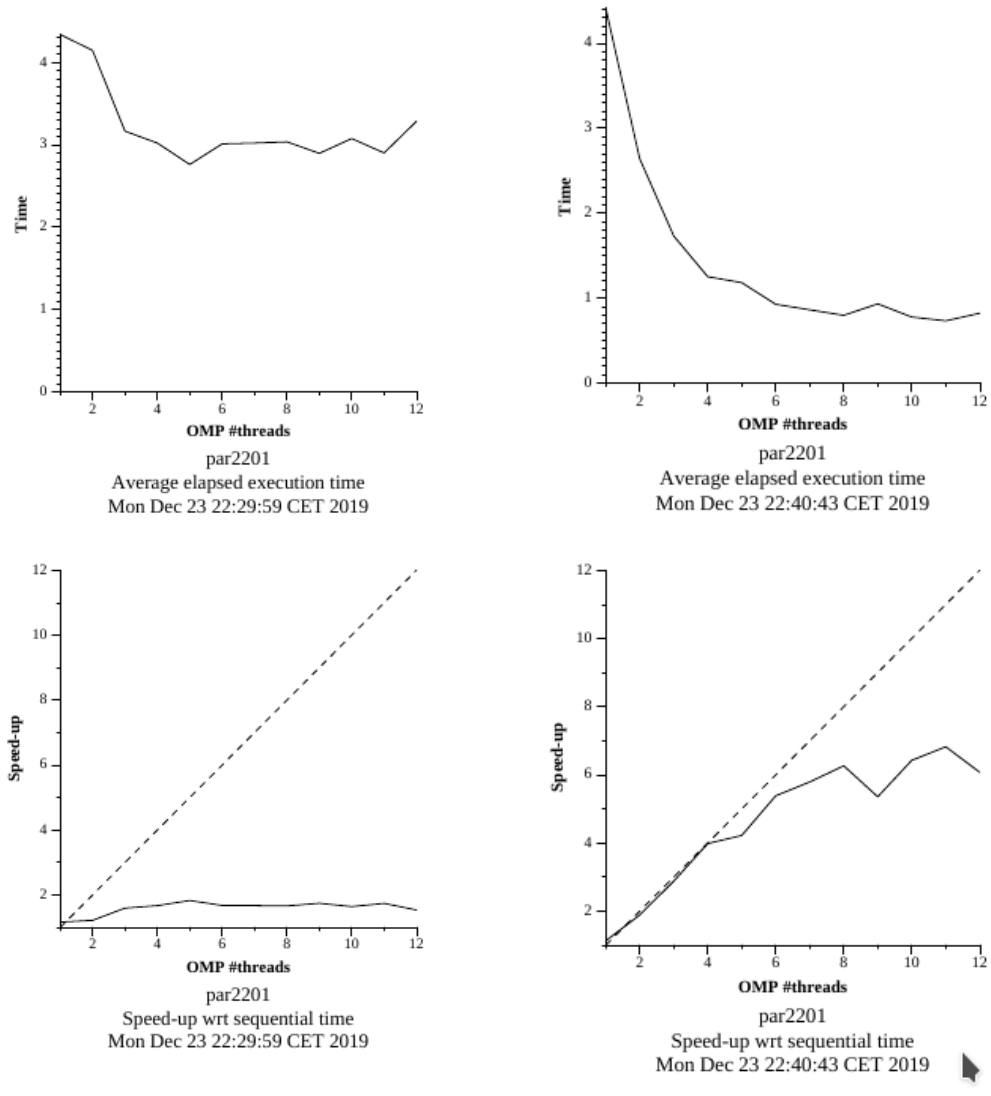

Figure 3.7: Scalability plots of the different parallelized versions of the *Jacobi* solver.

Chapter 4

OpenMP parallelization and execution analysis: *Gauss-Seidel*

Finally, after the whole parallelization process and deep analysis for the *Jacobi* solver, it is now time to do the same procedure with the *Gauss-Seidel* solver.

4.1 Parallelization of the code

Our initial parallel version of the *relax_gauss* function can be seen in *Figure 4.1*. There are several `Open MP` *ordered* directives being used throughout the code.

```
  double unew, diff, sum=0.0;
  int howmany=4;

# pragma omp parallel for ordered(2) private(unew, diff) reduction(+:sum)
  for (int i = 0; i < howmany; i++) {
    for (int j = 0; j < howmany; j++) {
    int i_start = lowerb(i, howmany, sizex);
    int i_end = upperb(i, howmany, sizex);
    int j_start = lowerb(j, howmany, sizey);
    int j_end = upperb(j, howmany, sizey);


#     pragma omp ordered depend(sink: i-1, j)
      for (int ii=max(1, i_start); ii <= min(sizex-2, i_end); ii++) {
        for (int jj=max(1, j_start); jj <= min(sizey-2, j_end); jj++) {
          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;
        }
      }
#     pragma omp ordered depend(source)
    }
  }

  return sum;
```

Figure 4.1: Code of the parallelized version of the function `relax-gauss`.

4.1.1 Instrumentation with *Extrae*

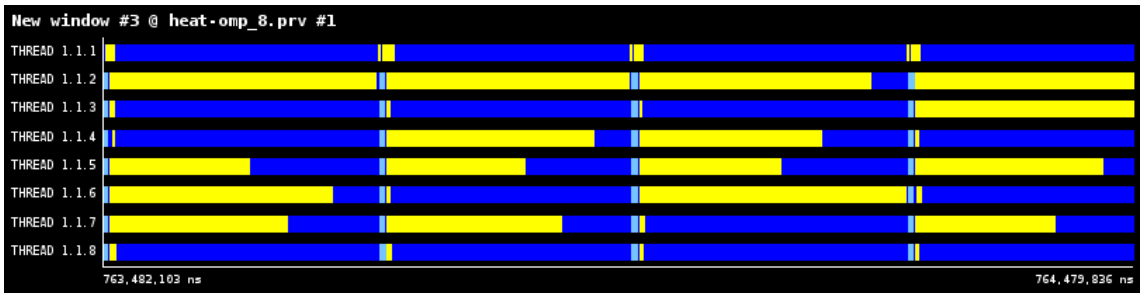

Figure 4.2: Trace of the execution of *Gauss-Seidel* solver obtained with *Paraver*.

In the instrumentation with Extrae we found, as expected, a lot of synchronization (yellow bursts). This is due the `ordered` clause which is waiting to meet the data dependency. Though it is wasting CPU time it is absolutely necessary for the correct behaviour of the solver.

4.2 Alternative parallelization

As an alternative parallelization version we will use `#pragma omp task` and task dependences to ensure their correct execution. The resulting code can be seen in *Figure 4.3*.

```
  double unew, diff, sum=0.0;
  int howmany=4;

# pragma omp parallel
# pragma omp single
  for (int i = 0; i < howmany; i++) {
    for (int j = 0; j < howmany; j++) {
    int i_start = lowerb(i, howmany, sizex);
    int i_end = upperb(i, howmany, sizex);
    int j_start = lowerb(j, howmany, sizey);
    int j_end = upperb(j, howmany, sizey);


#     pragma omp task \
        depend(in: u[(i_start-1)*sizey + j_start], u[i_start*sizey + (j_start-1)])
    \
        depend(out:u[ i_start   *sizey + j_end  ], u[ i_end *sizey +  j_start   ])
      for (int ii=max(1, i_start); ii <= min(sizex-2, i_end); ii++) {
        for (int jj=max(1, j_start); jj <= min(sizey-2, j_end); jj++) {
          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;
        }
      }
    }
  }

  return sum;
```

Figure 4.3: Code of the alternative parallel version of the function `relax-gauss`.

Our approach is to create a task for every given block, where the number of blocks is $howmany^2$. And once again in order not to break the data dependencies we can use directives given by OpenMP, in this case we are using the `depend` clause, to specify dependencies between tasks.

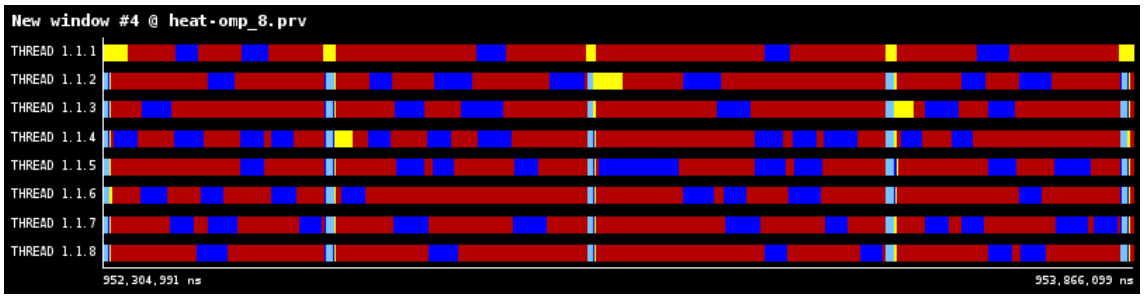

Figure 4.4: Trace of the execution of the alternative version of *Gauss-Seidel* solver obtained with *Paraver*.

4.2.1 Scalability analysis

Finally, we have obtained the speed-up and execution time plots for both parallel versions of the *Gauss-Seidel* solver. As seen in *Figure 4.5* both versions are very similar. Besides, the performance for both versions do not seem to differ from the final version parallel version developed for the *Jacobi* solver.

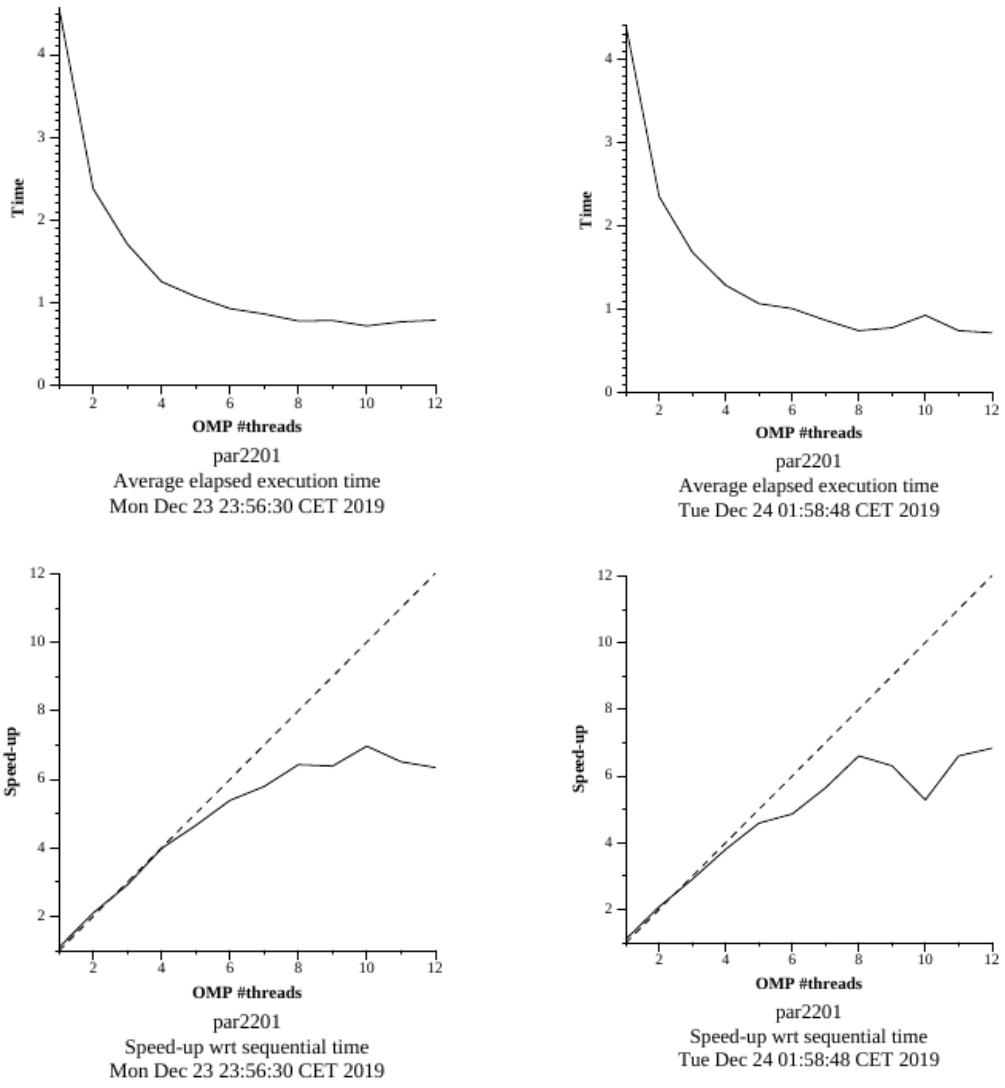

Figure 4.5: Scalability plots of the different parallelized versions of the *Gauss-Seidel* solver.

Chapter 5

Conclusions

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

An important point to remember is that several techniques are always available, but for every single problem there is typically one that fits better than others. For instance, for the *Jacobi* solver the data decomposition technique is the perfect choice, while on the other hand for the *Gauss-Seidel* solver due to the data dependencies other options might be a better fit.

The importance of understanding what everything within a code does before parallelizing it has once again been recalled, and fully understanding dependencies being of prime importance.