# PAR Laboratory Assignment
# Lab 5: Geometric (data) decomposition using implicit tasks:heat diffusion equation

E. Ayguadé, J. R. Herrero, P. Martínez-Ferrer,
J. Morillo, J. Tubella and G. Utrera Spring
2021-22

Maria Montalvo Falcón (par 4214)
Victor Pla Sanchis (par4219)
Group: 42
Date: 02/06/22
Course: 2021-2022 Q2

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

# Index

# 1 Introduction

In this laboratory assignment we have worked the parallelisation of a sequential code for the diffusion of heat in a solid body using two different solvers for the heat equation (Jacobi and Gauss-Seidel). Each solver shows different parallel behaviours that have allowed us to work different aspects. In the first session we had a first contact with the code and used the tool of *Tareador* to inspect the TDG and its dependencies.
On the second one, we parallelised both implementations with OpenMP clauses with implicit tasks in order to control the access to the tasks and analysed the results.
Finally, in the last laboratory session we worked again the implementations with OpenMP clauses but that time, indicating the clauses necessary to create explicit tasks, which give the programmer more comfortability at the time of coding, but without allowing the control of which thread would access each task.

# 2
# Analysis of task granularities and dependencies

In this section we have worked the performances of Jacobi and Gauss-Seidel codes with the Tareador tool. This information has helped us to study how the dependencies behaved. First, we compiled and executed the code given with the command line `make` and `./heat test.dat -a 0 -o heat-jacobi.ppm` for Jacobi implementation, meanwhile to execute Gauss-Seidel the command line was `./heat test.dat -a 1 -o heat-gauss.ppm`. Here below are shown the figures related to these executions generated with `display`.

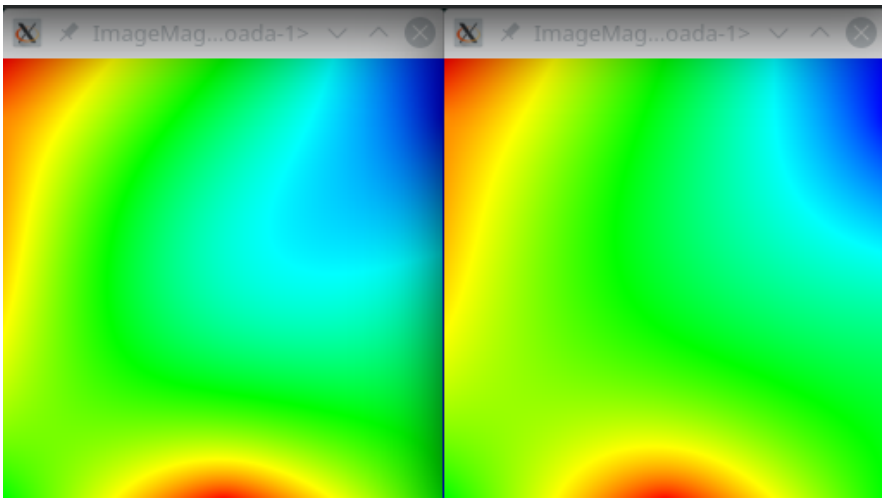



*Figure 2.0.1 On the left Jacobi output, on the right Gauss-Seidel one*

These outputs have allowed us to check the correctness of the modified implementations on the following sections. Note that images were not exactly the same for both codes.
In addition, the console executions gave us some extra information about each code.

```
par4214@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: 4.692
Flops and Flops per second: (11.182 GFlop => 2383.08 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

*Figure 2.0.2 Jacobi console output*

```
par4214@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: 6.127
Flops and Flops per second: (8.806 GFlop => 1437.31 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

*Figure 2.0.2 Gauss-Seidel console output*

As seen on the plotted information of the console outputs, Jacobi code converged later than Gauss-Seidel one in terms of iterations. The reason why this happened is because Gauss-Seidel code has an advantage due to the calculation of the data (it is done directly on the matrix). However, the time of Jacobi implementations is 4,692 seconds whether Gauss-Seidel one is 6,127 seconds.

## 2.1 Jacobi and Gauss-Seidel heat program with Tareador

After the first execution of each code, we moved on to execute Jacobi and Gauss-Seidel Tareador codes to see the TDGs created. To do that, we compiled with the command line `heat-tareador.c` and then executed the script with the command line `run-tareador.sh`. See on figure 2.1.1 the related graph.

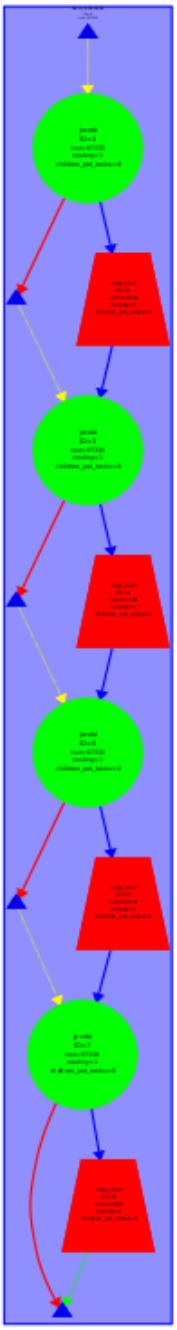


*Figure 2.1.1 Jacobi TDG*

As seen in figure 2.1.1 and 2.2.2 (here below), there wasn't almost any kind of parallelisation at that level of granularity, so the code remained almost sequential even though the creation of tasks and there is not any parallelisation that can be exploited at that level of granularity.

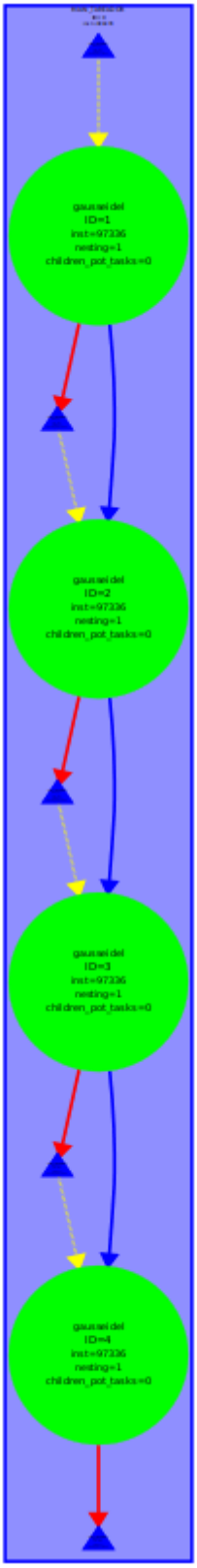



*Figure 2.1.2 Gauss-Seidel TDG*

After seeing that this granularity was not adequate, we explored finer levels into

`solver-tareador.c` code. As requested, we created one task per block modifying the code as followed:

```
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {

    double tmp, diff, sum=0.0;

    int nblocksi=4;
    int nblocksj=4;

    //tareador_disable_object(&sum);

    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) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);

        tareador_start_task("block-task");

        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("block-task");

      }
    }

    //tareador_enable_object(&sum);

    return sum;
}
```

*Figure 2.1.3 Code after modifying task granularity*

We have added the `tareador_start_task(block-task)` clause on each ij block and the `tareador_end_task(block-task)` clause to mark the end. Next, we generated the related TDGs again.

On the new graphs, one could see that there weren't again any kind of parallelisation. Indeed, we were creating more overhead as we were creating tasks but they were being executed sequentially due to the dependencies. See figures 2.1.4 and 2.1.5 here below.

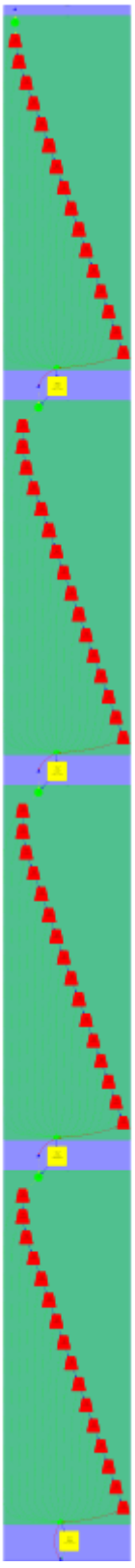


*Figure 2.1.4 Jacobi TDG with block task*

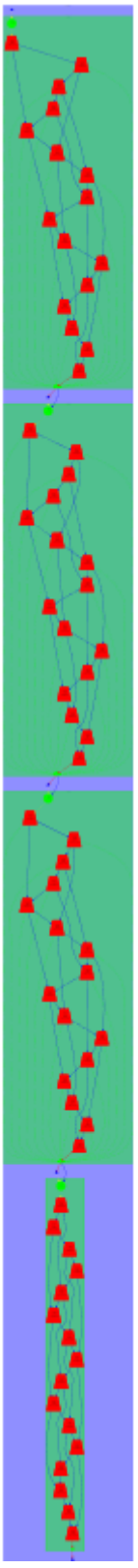


*Figure 2.1.5 Gauss-Seidel TDG with block task*

After seeing these graphs, we inspected the related code to see which was the variable creating the dependency.

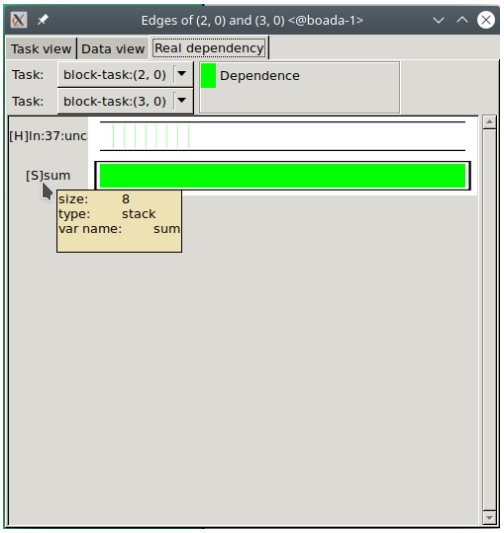



*Figure 2.1.6 Dependency variable sum*

As shown on the figure above (Figure 2.1.6) we saw that the variable that was preventing the parallelisation was the variable `sum` . After identifying the variable that was causing the serialization with the DataView option on Tareador, we were able to modify the code and solve this dependency. To do that, the code given has two commented lines that we had only to uncomment (see figure 2.1.3). The directives were:

- `tareador_disable_object(&sum)`
- `tareador_enable_object(&sum)`

With OpenMP clauses, this dependency could be solved with clauses such as critical, atomic, or reduction among others. In the following section related to OpenMP we will see this with more detail.
Anew, we generate the TDGs related to this implementation to see how the parallelisation worked with this dependency managed.

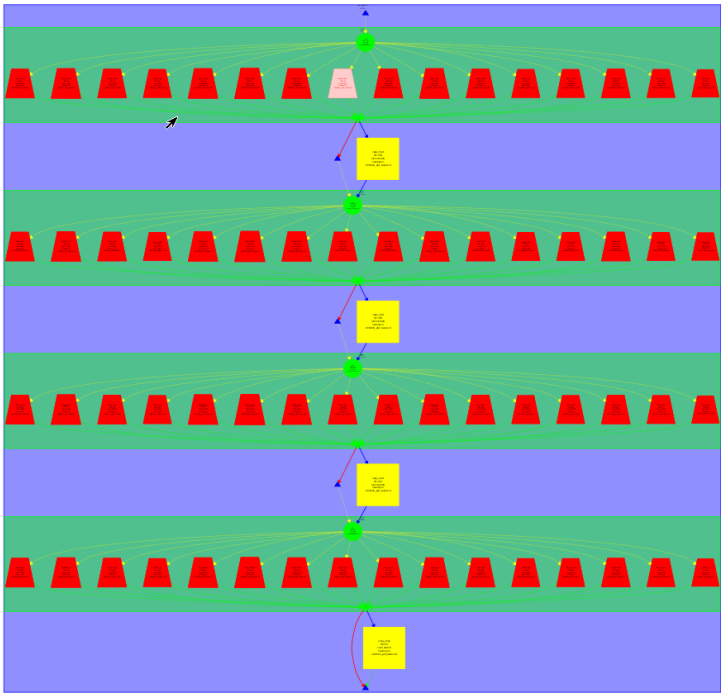


*Figure 2.1.7 TDG of Jacobi implementation with variable sum managed*

Now, it is shown on the graph dependency that the geometric block parallelisation was allowed. In the case of the Jacobi option, the tasks generated in red colour were the ones related with the $block_{ij}$ tasks of the loop, whether the yellow ones were identifying the `copy_mat` function (a potential region to parallelise, as is the part of the code that is introducing serialisation)

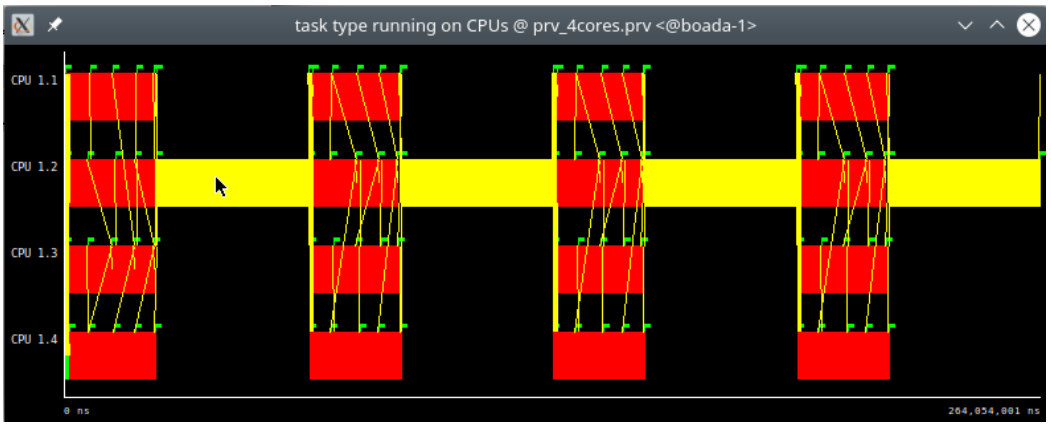
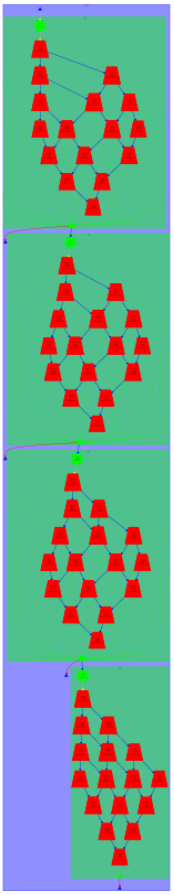



*Figure 2.1.8 Simulation of Jacobi code parallelised with 4 processors*

After that, we simulated the execution with four processors. As seen and as previously said, the `copy_mat` function is introducing serialisation whether the rest of the tasks were being executed in parallel with their related synchronisations.

We did the same work with Gauss-Seidel implementations. Here below is shown the graph after managing sum dependencies.

*Figure 2.1.9 TDG of Gauss-Seidel implementation with variable sum managed*



As shown, managing sum dependencies allowed us to parallelise Gauss-Seidel code too. In this case, as

there is no need to copy the matrix because Gauss-Seidel code rewrites the same matrix that is reading, there wasn't any region creating serialisation as before, but it implies that some tasks were obligated to wait until other ones were finished. This kind of dependency is identified as wave dependencies as it is the manner to run the matrix.

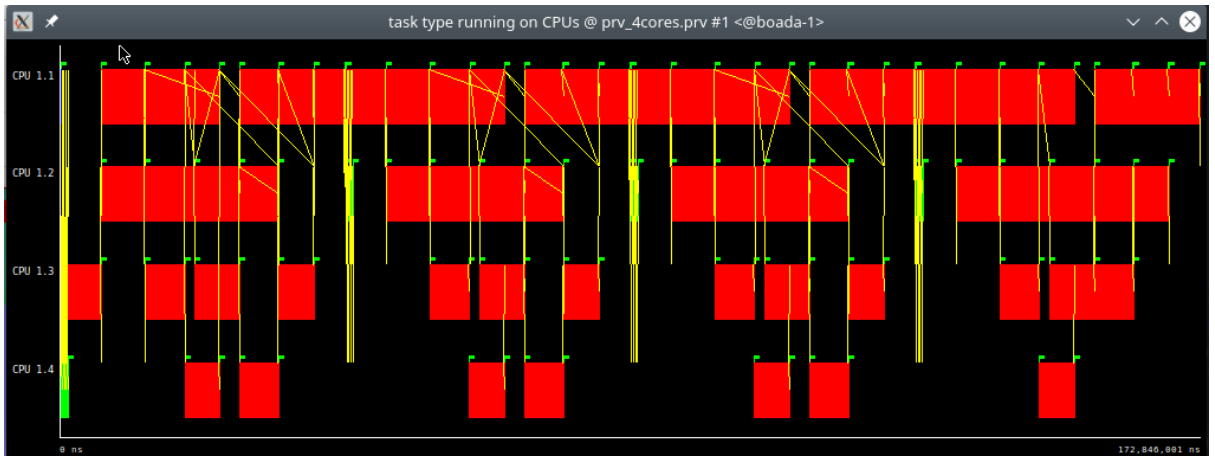



*Figure 2.1.10 Simulation of Gauss-Seidel code parallelised with 4 processors*

We simulated this code with Tareador and 4 processors. As said before there was parallelisation but there also was some time-lapses where the thread had to wait for synchronisations. Compared to the one of Jacobi this execution is not as symmetric as the previous one, in which all threads were executing at the same time the tasks and then copying it to the matrix. In this case, the wave dependency does not benefit at all of the 4 processors.

Recapturing Jacobi code, we moved on to parallelise the `copy_mat` function. Here below is shown the code proposed. We opted to create a task for each block as we did previously with the function solve.

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned
sizey) {

    int nblocksi=4;
    int nblocksj=4;

    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) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);

        tareador_start_task("copy-block-task");

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

        tareador_end_task("copy-block-task");

      }
    }
}
```

*2.1.11 Code of function copy_mat*

We created the graph dependency of Jacobi implementation and with the function copy_mat parallelised. In this case, we didn't work the Gauss-Seidel code as it does not make sense because it doesn't use the copy_mat function. Here below is the graph generated.

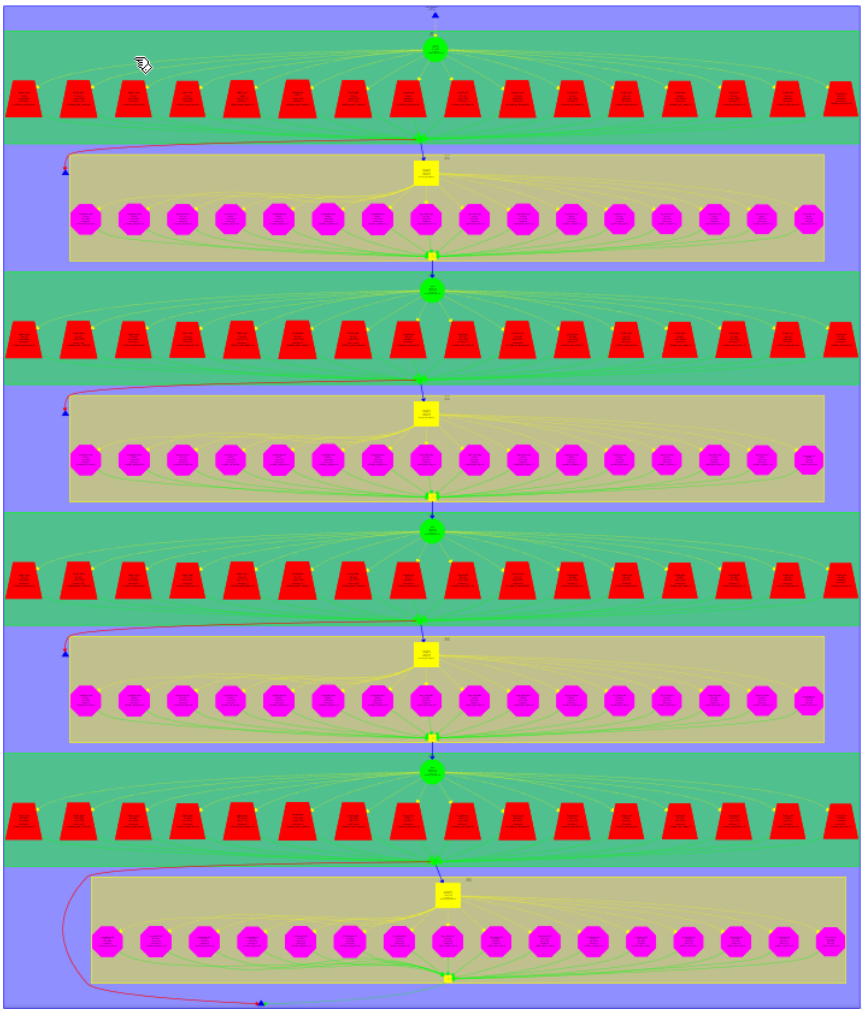


*Figure 2.1.12 TDG of Jacobi implementation with variable sum managed and copy_mat parallelised*

After that, we explore the simulations with 4 and 16 processors. Here below is shown both traces created. Looking into these figures, one could notice that as the level of parallelism increased because of the parallelisation of the function `copy_mat`, the previous serialisation generated by this function disappeared and more parallelism was exploited.

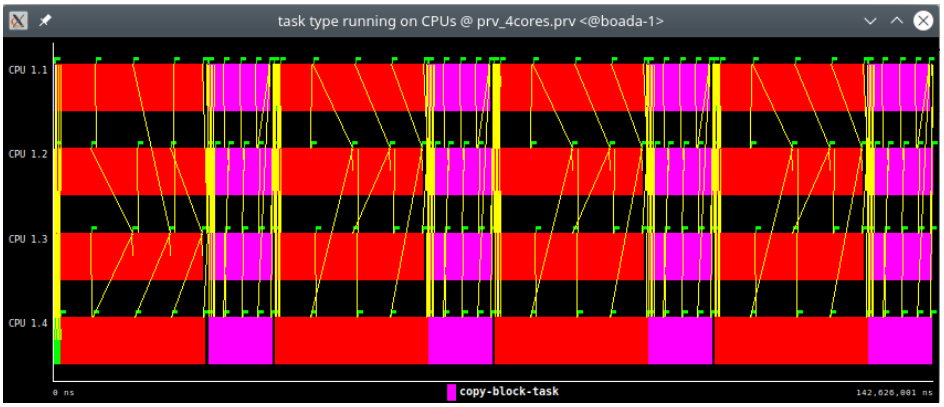



*Figure 2.1.13 Simulation of Jacobi code (with copy_mat) parallelised with 4 processors*

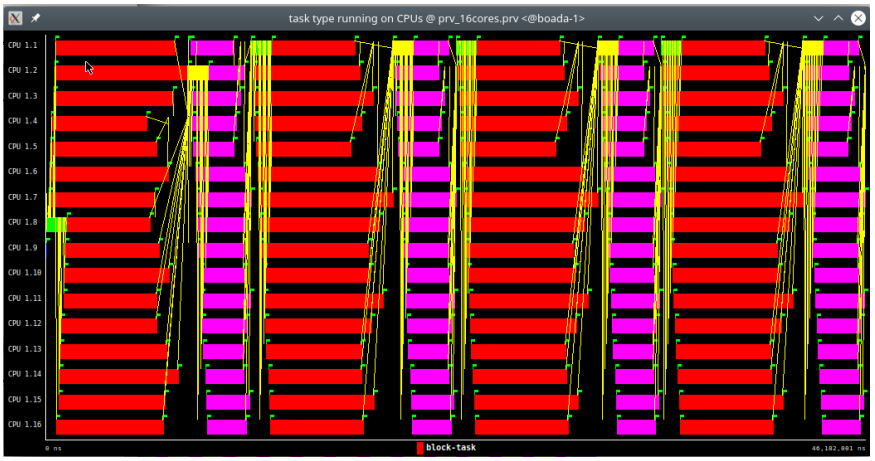



*Figure 2.1.14 Simulation of Jacobi code (with copy_mat) parallelised with 16 processors*

# 3

# OpenMP parallelization and execution analysis: Jacobi

In this section we were able to explore the parallelisation and execution analysis with OpenMP clauses for the Jacobi code. In order to do that, we started parallelising the Jacobi program method in order to see how the program behaves. We had to implement all the task creation structure and then the dependencies we found in Tareador (with the variable sum) and the serialisation made by the `copy_mat` function. Then we were analysing those results that create the implementation.

First of all we looked at the task generation, we started creating a task for a set of rows as the following figure shows (see figure 3.1).

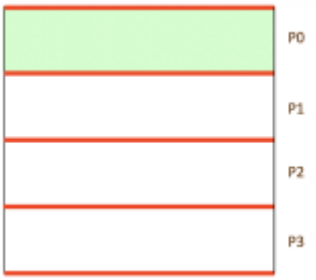



*Figure 3.1 Geometric block data decomposition by rows for Jacobi method*

To do that we completed the code as shown on the following image.

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

    int nblocksi=omp_get_max_threads();
    int nblocksj=1;

    #pragma omp parallel private(tmp, diff) reduction(+:sum) // complete data sharing constructs here
    {
      int blocki = omp_get_thread_num();
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      for (int blockj=0; blockj<nblocksj; ++blockj) {
        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;
          }
        }
      }
    }

    return sum;
}
```

*Figure 3.2 Code for OpenMP implementation on Jacobi method*

As seen above, we added the clause `private(tmp, diff)` to protect the variables and make sure that every thread had a private variable for *tmp* and *diff* and any other thread was rewriting it provided

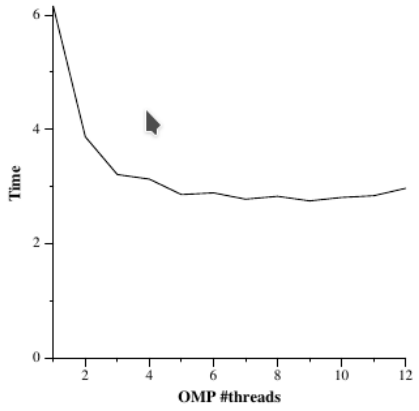
themselves. Also, to maintain the correctness of the program we had to add the clause `reduction(+:sum)` to remove the dependencies made by this variable (as seen on section 2) and reduce it at the end of the tasks. Then, we executed the program with the command line `make heat-omp` and submitted with the help of the script and the line `sbatch submit-omp.sh heat-omp 0 8`. Then we compared the results with the sequential one, to validate the output of the new implementation, as we see on figure 3.3, with the command `diff` we can see the output didn't differ.

```
par4214@boada-1:~/lab5$ diff heat-jacobi.ppm heat-jacobi-seq.ppm
par4214@boada-1:~/lab5$
```

*Figure 3.3 diff command for Jacobi implementation*

After validating the correctness of the code, we submitted the execution with the command line `sbatch ./submit-strong-omp.sh 0` to generate the plots of the strong scalability from 1 up to 12 processors indicating the version of Jacobi with number 0. To do a better analysis the script is modified with a problem size (-s 1022) but with less simulation steps (-n 1000).

As seen the time execution is reduced from 4,692 seconds of the sequential version of Jacobi implementation to around 3 seconds of the parallelised one. With the overhead of the creation of tasks, when executing with one thread the parallelised version goes up to 6 seconds. When increasing the number of threads to 5 the time is reduced to 3 seconds and then it seems to reach a ceiling at this point. With that, the speedup achieved is 2. See below the plots commented and generated.



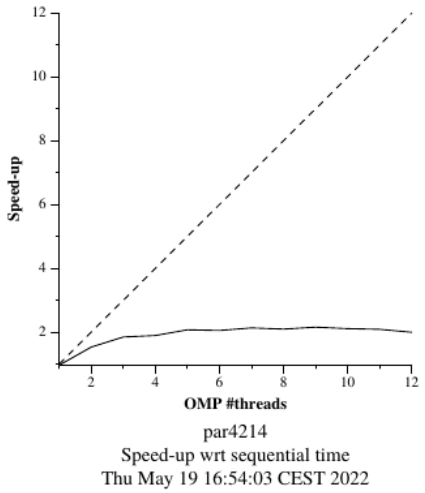



*Figure 3.4 Plots strong scalability Jacobi implementation*

To have a better understanding of what was happening, after the execution of the plots we moved on to generate the trace of the *Parever* tool. Again, we submitted the work with the command line `submit-extrae.sh 0` and with the same arguments.

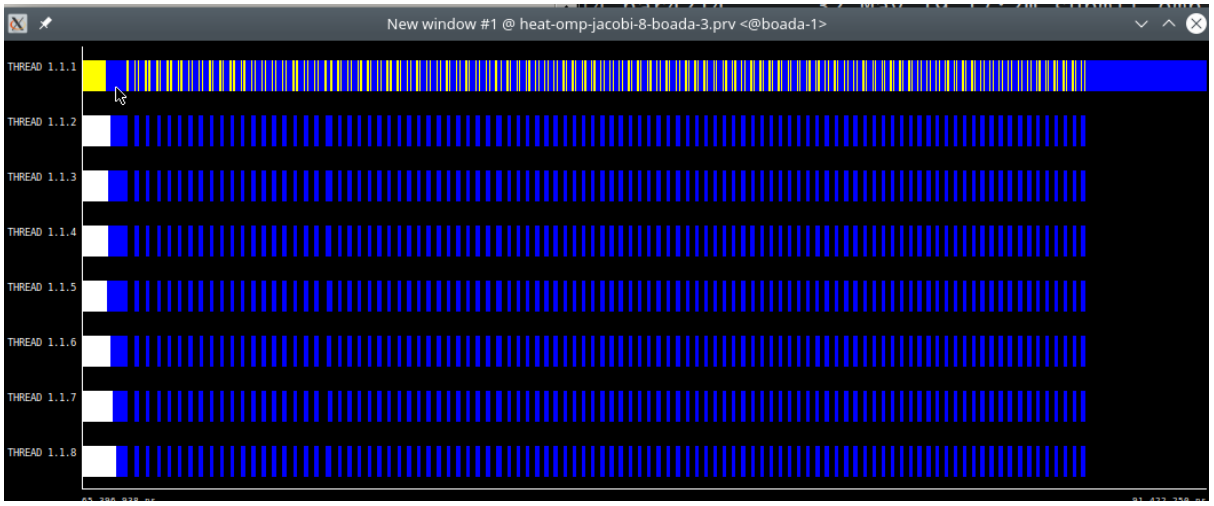



*Figure 3.5 Execution trace of Paraver with Jacobi code parallelised*

On the figure above it is shown the execution of Jacobi implementation with 8 threads. It is seen that the parallelisation made every thread work the same amount of time except thread one. The blue colour represents the part of time that threads were running and the black spaces are the time in which threads were not executing anything. Moreover, even though the parallelisation improved the performance of the execution, one can see how the threads are obligated to wait for the sequential part executed by thread one after continuing running. This is due to the fact that at this point we were still not parallelising the function that was copying the matrix that was adding serialisation.

After analysing the behaviour of the program with Paraver, we decided to parallelise the function `copy_mat`, as it was the one introducing serialisation to the program as said before.
Here below is shown the code of the `copy_mat` function modified with the adequate parallelisation: modifying nblocksi equal to the number of threads (it would be taking into account in the *lowerb* and *upperb* boundaries) and adding the clause `#pragma omp parallel`.

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

    int nblocksi=omp_get_max_threads();
    int nblocksj=1;

    #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 blockj=0; blockj<nblocksj; ++blockj) {
        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++) {
              v[i*sizey+j] = u[i*sizey+j];
          }
        }
      }
    }

}
```

*Figure 3.6 Code of copy_mat function parallelised*

After the modifications were applied, we checked the correctness of our program with the diff command line as shown below.

```
par4214@boada-1:~/lab5$ diff heat-jacobi-copy.ppm heat-jacobi.ppm
par4214@boada-1:~/lab5$
```

*Figure 3.7.1 diff command for Jacobi implementation with copy_mat parallelised*

Next, as our program was generating the correct output, we studied the parallelisation results of it. We compiled and submitted the work to generate the new trace with the *Paraver* tool.

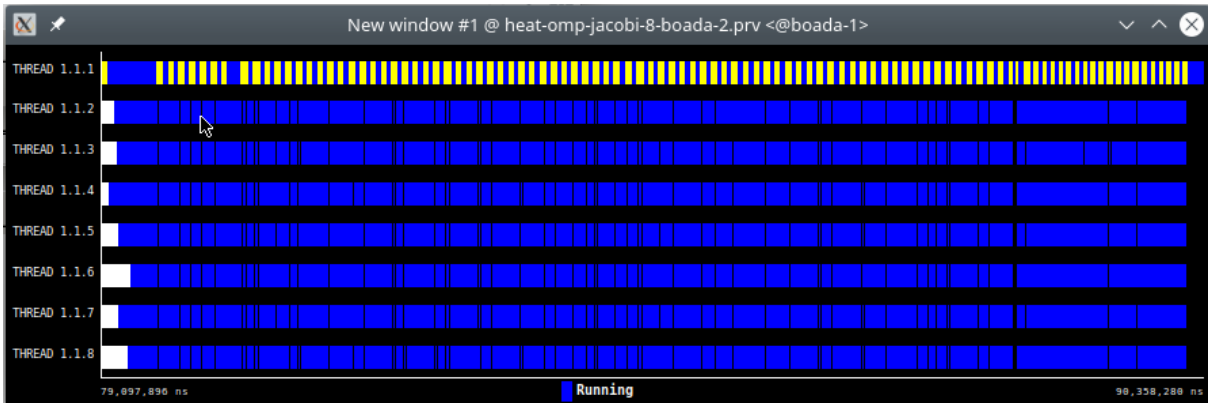



*Figure 3.8 Execution trace of Paraver with Jacobi code and copy_mat parallelised*

Comparing the results with the previous trace of *Paraver*, it is seen that the new trace of figure 3.8 was reducing the black spaces where the threads were not running. It means that the parallelism was increased so we expected to have better results on the strong scalability plots. We executed the script to see if that was true with the command line `sbatch ./submit-strong-omp.sh 0`. Here below the plots resulted.

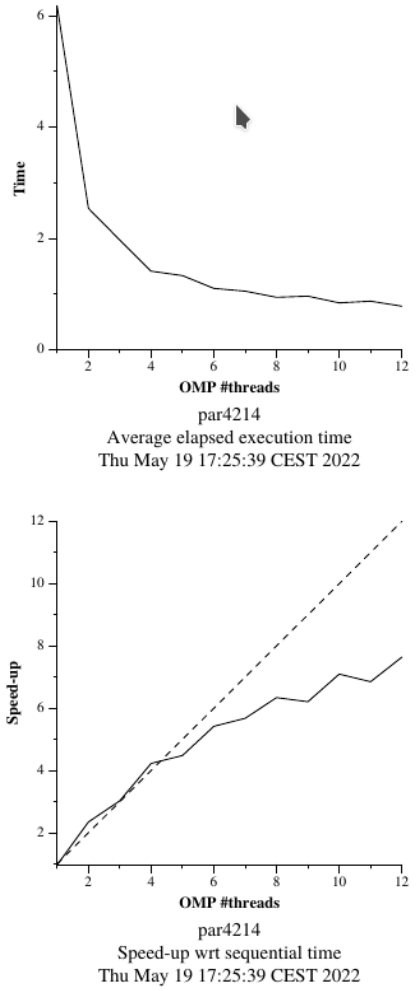



*Figure 3.9 Plots strong scalability Jacobi implementation with copy_mat parallelised*

As seen, with the code parallelising copy_mat function, the time execution was reduced to 1 second when the number of threads reached 8. On the other hand, speedup was increased to the ideal (in some cases, a little bit better because of super linear properties) and remains as the ideal until 4-5 threads. Then it started to be lower than the ideal because of overheads but remained proportionally high. This implementation exploited a higher level of parallelism and gave better results without any doubt.

# 4
# OpenMP parallelization and execution analysis: Gauss-Seidel

After inspecting Jacobi code, we moved on the parallelisation and analysis of the Gauss-Seidel method. As we did with Jacobi, we had to implement the task generation structure and then take care of the dependencies that Tareador has previously shown us. We started parallelising the matrix by rows as shown on Figure 4.1.

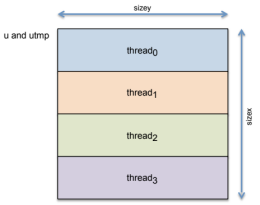



*Figure 4.1 Geometric block decomposition by rows*

To parallelise this version we created a vector called *next* in order to keep the information of which blocks had been calculated. The reason why this is necessary is because in Gauss-Seidel implementation the dependencies in the matrix (which is accessed to read and to write the computation) are of the wave type. It means that to start executing the block, you need the calculation of the related upper block, so the execution draws a kind of diagonal execution of the blocks of the matrix. This vector *next,* was initialised with the values of 0 (thread 0 could execute all the block without any dependency) and with P positions (number of threads). Then, we added a do while structure that allowed us to lock the threads in that region until the next values of their position reached at least their number of threads indicating they can start to calculate. Finally, to control the data races, we add the related clauses `#pragma omp atomic read` and `#pragma omp atomic` as shown below. Finally we divided the i block row with the same number of blocks of the j dimension with the instruction `nblocksj=nblocksi`.

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

    int nblocksi=omp_get_max_threads();
    int nblocksj=nblocksi;

    const int P = omp_get_max_threads();
    int next[nblocksj];
    for(int i=0; i<P; i++) {
        next[i] = 0;
    }

    #pragma omp parallel private(tmp, diff) reduction(+:sum) // complete data sharing constructs here
    {
      int blocki = omp_get_thread_num();
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      for (int blockj=0; blockj<nblocksj; ++blockj) {
        int tmpRead;
        do {
          #pragma omp atomic read
          tmpRead = next[blockj];
        } while(tmpRead < omp_get_thread_num());

        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;
          }
        }

        #pragma omp atomic
        next[blockj]++;

      }
    }

    return sum;
```

*Figure 4.2 Code for OpenMP implementation on Gauss-Seidel method*

As always, we checked the correctness of the method with the `diff` command line and using in this

case, the ppm related to the sequential execution of Gauss-Seidel.

```
par4214@boada-1:~/lab5$ diff heat-gausseidel.ppm heat-gausseidel-seq.ppm
par4214@boada-1:~/lab5$
```

*Figure 4.3 diff command for Gauss-Seidel implementation parallelised*

So as the code was correct, we generated the trace of *Paraver* to see the behaviour of this new parallelised version. Looking at the image 4.4 is shown that the majority of the time the threads were running, which means they were doing useful work. Comparing this version to the previous one, the black spaces were reduced so there is less scheduling time.

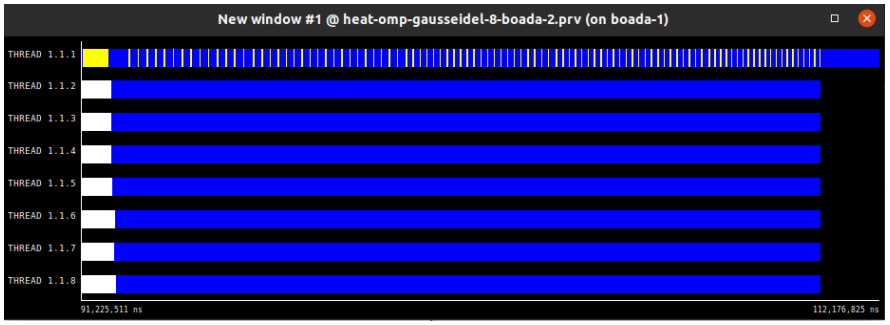



*Figure 4.4 Execution trace of Paraver with Gauss-Seidel implementation*

The behaviour seen with the *Paraver* tool is reflected in the scalability of the program; as seen on image 4.5 the parallelised program is increasing the speed-up in relation to the sequential version, but not as desired. That's because our program can't reach a better scalability for the data dependencies generated between the blocks of the matrix. The time is reduced from 6,127 seconds of the sequential Gauss-Seidel code to a little more than 1 second of the parallelised one as shown below. The speedup reaches 5 with 12 threads but as said, is not near the desired ideal speedup.

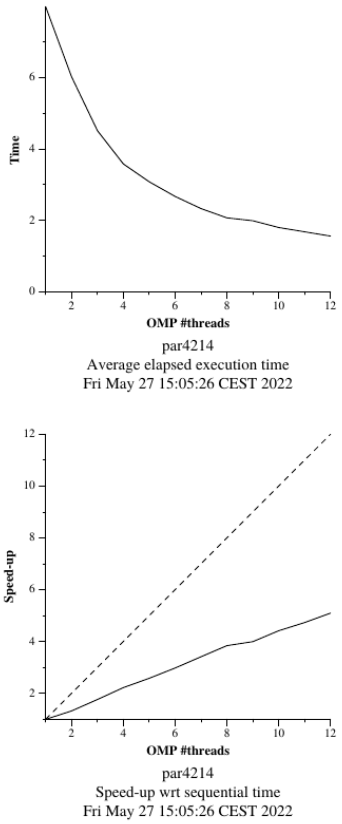



*Figure 4.5 Strong scalability of Gauss-Seidel*

To explore different levels of exploiting parallelism, we started to modify the number of blocks in the j

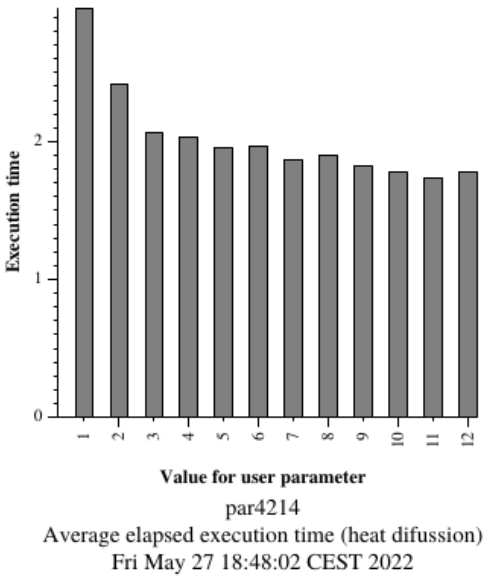
dimension with the parameter userparam, that parameter is defined with the option -u on the execution of the program. With that global variable set, we were allowed to use it on the solver program because we had the code line extern int userparam on the top. We modified the code to use that new parameter and we decided to do it as the following image shows.

```
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
    double tmp, diff, sum=0.0;

    int nblocksi=omp_get_max_threads();
    int nblocksj=userparam * nblocksi;
```

*Figure 4.6 Code modified with userparam*

Now that we have changed the number of blocks to exactly the same number of userparam variable, we have executed the program with the `submit-userparam.sh` script to see how the new performance behaves. Here below are shown the time execution plots for values of userparam from 1 to 12 and with 8 threads on the left figure and 4 on the right one.



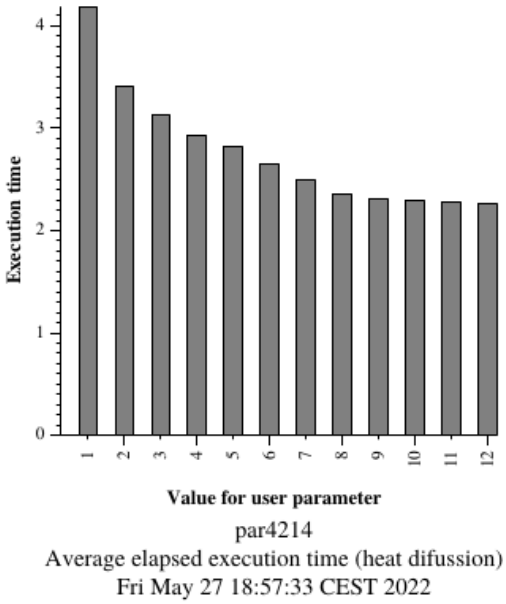




*Figure 4.7 Time execution plot with 8 threads*

*Figure 4.8 Time execution plot with 4 threads*

As seen on the figures above, it seemed that as we increased the value of userparam variable more level of parallelism is exploited. The reason why this happens is because as more blocks are created, the little they are and also the time to wait. So threads can start to execute before. It is obvious that the time execution of 4 threads is higher than with 8 because of the tasks that can be executed at the same time. To validate this proof we also decided to evaluate this script with 2 and 12 threads. Here below, both plots with the information extracted.

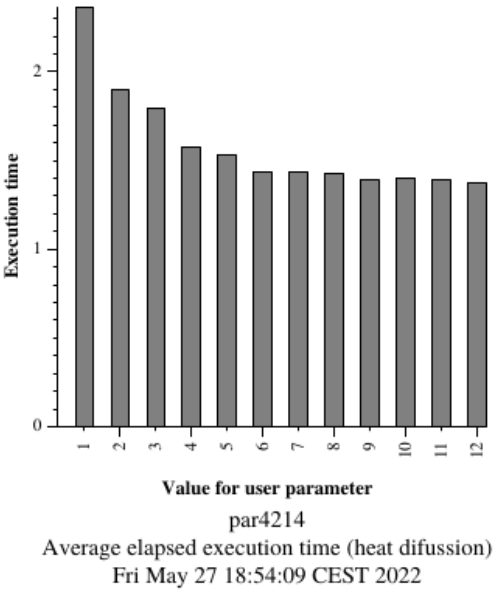



*Figure 4.9 Time execution plot with 12 threads*

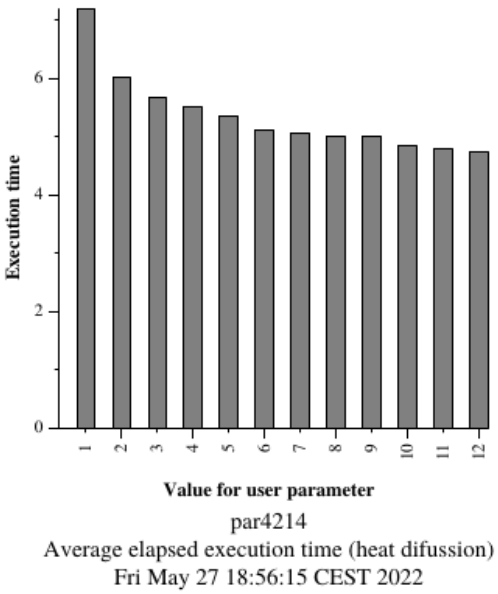



*Figure 4.10 Time execution plot with 2 threads*

As expected, the results with 12 and 2 threads validate the previous reasoning. When incrementing the value of userparam, the time execution is reduced. And again, it is observed that the time execution with 12 threads is much lower than with 2 because of the tasks that can be performed at the same time.

# 5
# Optionals

## Optional 1

In this optional part we were asked to work the parallelisation of the program with explicit tasks. To do that, we were able to use clauses such as: `#pragma omp task`, `#pragma omp taskloop`, etc.

First of all, we worked on the proposed exercise with Jacobi implementation. To do that, we added the clause `#pragma omp single`, and the clause `#pragma omp taskloop`. It was necessary to keep the `private(diff, tmp)` as they were variables of each thread, and the reduction on variable sum because of dependencies justified with Tareador on previous sections.

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {
    #pragma omp parallel
    #pragma omp single
    {
        int nblocksi=omp_get_max_threads();
        int nblocksj=nblocksi;
        #pragma omp taskloop
        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) {
            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++) {
                        v[i*sizey+j] = u[i*sizey+j];
                    }
                }
            }
        }

    }
}

// 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
    #pragma omp single
    {
        int nblocksi=omp_get_max_threads();
        int nblocksj=nblocksi;
        #pragma omp taskloop private(diff, tmp) reduction(+:sum)
        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) {
            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;
            }
            }
        }
        }
    }
    return sum;
}
```

*Figure 5.1 Code of optional 1, Jacobi explicit tasks*

As seen on Figure 5.1, the most of the time of execution, threads are scheduling or doing synchronisation but not doing actual work. Comparing this with the Figure 3.8 of the previous section 3, one can expect a worse performance on time execution and speedup. The reason why this could happen is because the tradeoff between the comfortability of coding with explicit tasks and the loss of control of which tasks will be executed by which thread, is reflected in the time execution of the program.

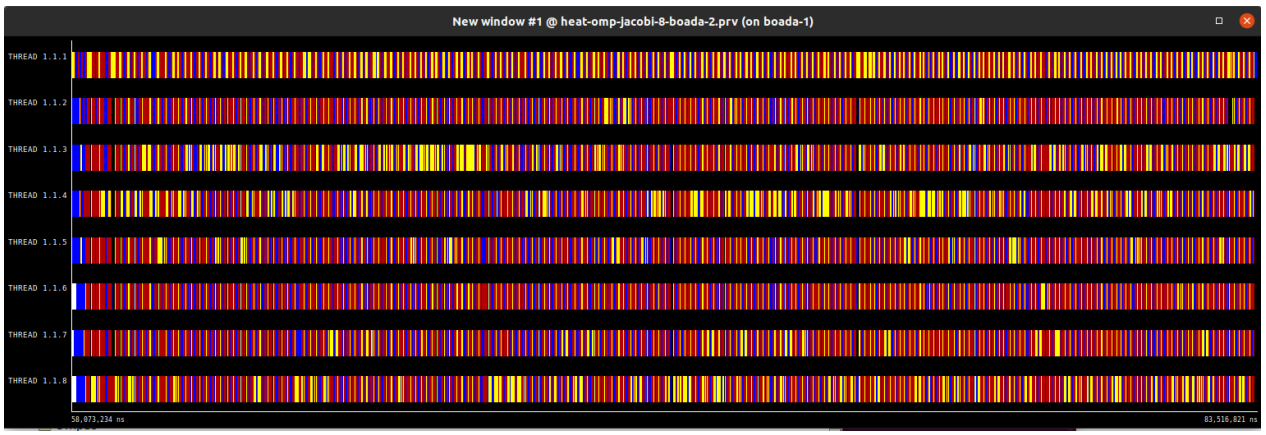



*Figure 5.2 Paraver Jacobi explicit tasks*

In these photos here below, are shown the explicit tasks function creation and execution. As seen, there were not mainly thread creating tasks, as there are created among the number of threads (in this case 8). In addition, in the execution one can see how the program is well parallelised as there are no dependencies of the matrix in the case of Jacobi.

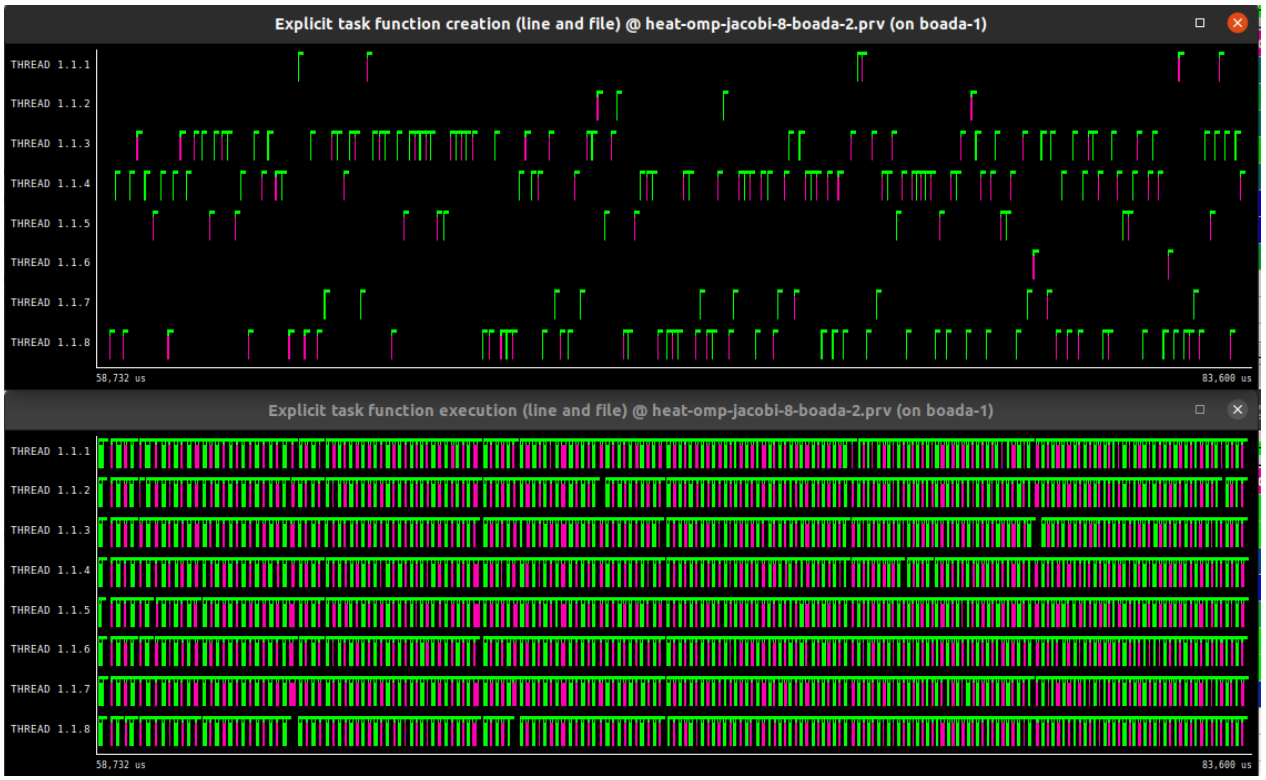



*Figure 5.3 Paraver Jacobi explicit function creation and execution*

Finally, in order to see how the performance was, we considered a good option to execute the submit strong script to generate the plot from 1 up to 12 threads.
If we compare this plot with the one on figure 3.9, we can see that as expected, the program performance has worsened. If we recall plot 3.9, the time decreased to under a second and the speedup reached the ideal until 6 threads and then, remained under the ideal but higher. In this case with explicit tasks, the time has decreased to 2 seconds and the speedup is far from the ideal.

After considering both photos, we can conclude that in this case, controlling the access of the threads to the blocks of tasks improved the performance of the program so doing the  parallelisation with implicit tasks is a better option.

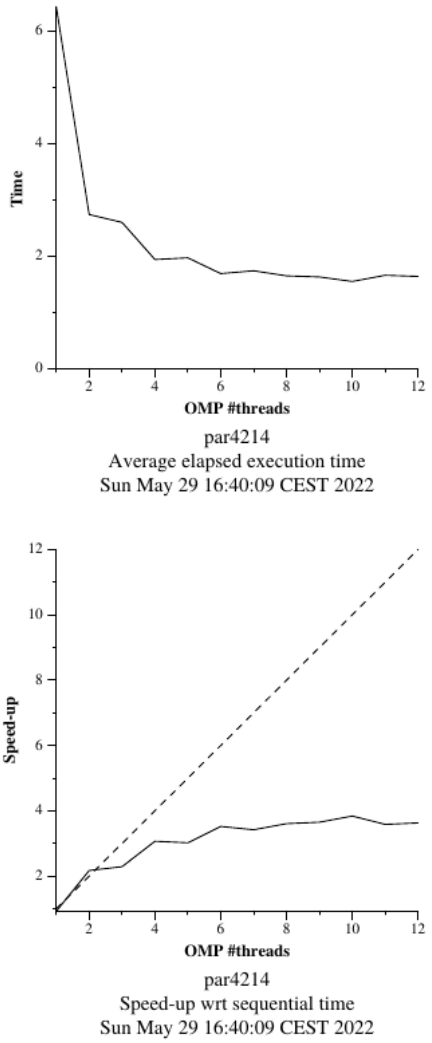



*Figure 5.4 Strong scalability plots Jacobi code with explicit tasks*

# Optional 2

In that section we had parallelised the Gauss-Seidel method with explicit tasks. First of all we had to add the external loop to iterate for all blocks on dimension i. We added the clause `#pragma omp single` for the creation of the explicit tasks (done just by one thread) and then we used *taskgroup* instead of *taskwait* for the synchronisation. Furthermore, inside of the parallel code, we had to define the dependencies for every task with the depend clauses (task created before the third loop) as we were creating a task for every block of the matrix. Also remark that we have to still indicate that variables diff and sum had to be private and declare the final reduction of the variable sum for the correctness.

```
// 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
    #pragma omp single
    #pragma omp taskgroup task_reduction(+:sum)
    {
        int nblocksi=omp_get_max_threads();
        int nblocksj=nblocksi;
        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) {
            int j_start = lowerb(blockj, nblocksj, sizey);
            int j_end = upperb(blockj, nblocksj, sizey);

            #pragma omp task private(diff, tmp) in_reduction(+:sum) depend(in:unew[(max(1, i_start)-1)*sizey+(max(1, j_start))], unew[(max(1, i_start))*sizey+(max(1, j_start)-1)]) depend(out:unew[min(sizex-2, i_end)*sizey+(max(1, j_start))],
            unew[max(1, i_start)*sizey+min(sizey-2, j_end)])

            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;
            }
            }
        }
        }
    }
    return sum;
}
```

*Figure 5.2.1 Explicit tasks code for Gauss-Seidel*

To maintain a good synchronisation we had to define the *depend* clauses. For every block we have two variables that depend on to start the execution (*depend in*) and two variables that we generate for those next blocks (*depend out*). To see it clearly, we can see on Figure 5.2.2 how we decided to define that *depend* variables.

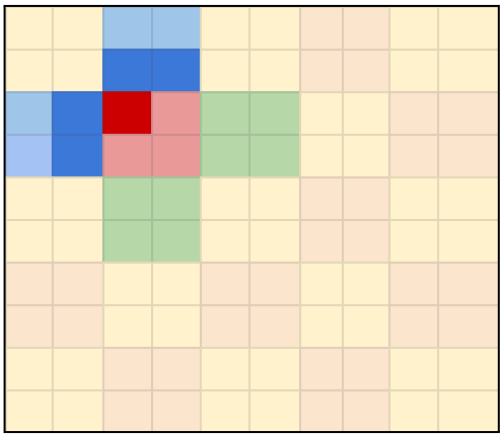


*Figure 5.2.2 Visualising depend clauses*

As we see on Figure 5.2.2, if the light red square are the block that we want to execute, we have to wait until the light blue squares ends their execution (where the dark blue squares are the exactly variables that we are waiting for), and then, when we execute our block, we can communicate that the dark red squares are computed, so furthermore, the other blocks that depend on our block to be executed are waiting for the results of dark red squares. In that case, the light green squares are waiting for our computation.
One time we have defined all that we need to maintain the correctness of the data, we can execute the program and study his behaviour.

```
par4214@boada-1:~/lab5$ diff heat-gausseidel.ppm ./displays/gauss-seq.ppm
par4214@boada-1:~/lab5$
```

*Figure 5.2.3 Visualising depend clauses*

We firstly checked that the program generates the correct output with the diff command, comparing it with the Gauss-Seidel sequential program. We see on Figure 5.2.3 that there's no difference. Then we decided to analyse the program with *Paraver*, executing the *submit-extrae.sh* script.

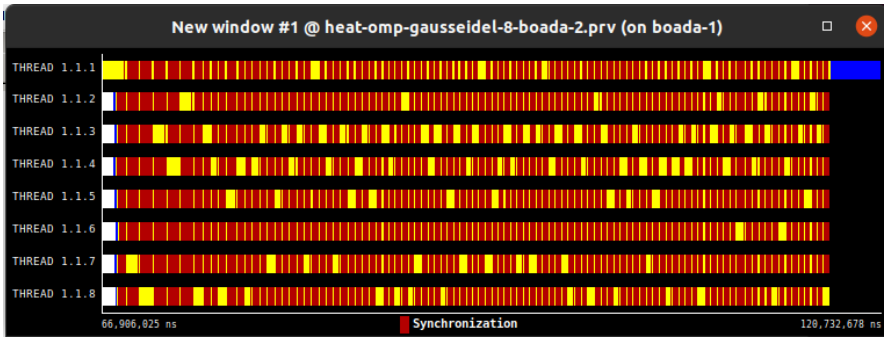



*Figure 5.2.4 Execution with 8 threads of Gauss-Seidel explicit tasks in Paraver*


*Figure 5.2.5 Explicit tasks creation and execution of Gauss-Seidel explicit tasks program with Paraver*

With Figure 5.2.4 we can see how the program is wasting a lot of time scheduling and synchronising all the tasks created, the same problem that we had with Jacobi explicit implementation, so in this case we can also expect a bad execution time.

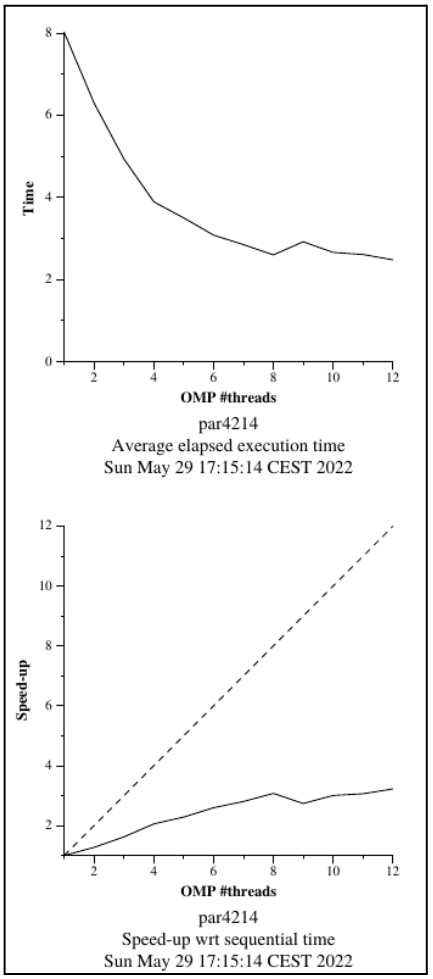



*Figure 5.2.5 Strong scalability analysis of Gauss-Seidel with explicit tasks*

When we had executed the strong scalability of the program, we were surprised by the bad performance that it had. We clearly see that the parallelisation of the program with explicit tasks is worse than the implicit tasks implementation.

# 6
# Conclusions

In this session we have explored how to parallelize with implicit tasks two different methods of a program that generates a 2D heat image, the Jacobi and Gauss-Seidel ones. With those solvers we have learned the data and tasks dependencies that both had and how to manage it with an implementation of our own synchronisation object.

We firstly studied with the Tareador tool how the TDG looks with both methods. We saw that Jacobi showed a TDG with a high dependency when the matrix is copied and otherwise the Gauss-Seidel solver when the matrix was modified. When we saw the TDGs with 8 processors we see that Jacobi method dependencies could allow us to parallelise the matrix with row parallelisation, but in the other hand, Gauss-Seidel forced us to do it with blocks because of some new dependencies that have been appeared, so we had to implement a synchronisation object in the future.

With all the Tareador tool information we were ready to implement the parallelization on both solvers. We firstly implemented the implicit tasks generation and the clauses to preserve the correctness of the data. We see that the performance of the Jacobi solver was depending on both methods of solver.c code, the solver method and the copy_mat method. The scalability of Jacobi solver was that good, that was better than the ideal speed-up. In the other hand Gauss-Seidel had a worse performance on scalability because of his wave dependencies.

Finally, we did both optional exercices, the main idea of them was to reproduce the same parallelism with explicit tasks. Taking care of all dependencies, we implemented that new solution that in performance was very worse compared to implicit tasks implementation. That’s because the tasks are generated in a very low level and have to do it one by one. We can see it clearly how the scalability of that implicit programs are both very below of the ideal speed-up.