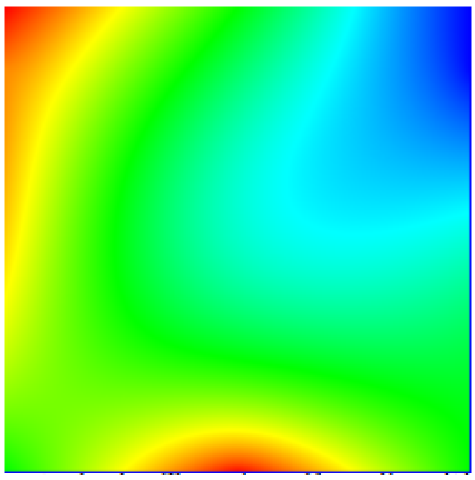
PAR Laboratory

Assignment Lab 5:

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



Guillem Dubé Quintín par3103

Ricard Guixaró Trancho par3108

Fall 2021-2022

Date: 31/12/2022

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Index

1. Introduction

In this session, we will work on the parallelisation of a sequential code (*heat.c*) that simulates the diffusion of heat in a solid body. We'll use two different solvers for the heat equation, each of which shows different parallel behaviour: ***Jacobi*** and ***Gauss-Seidel***.

We start by executing the sequential version of the program: we do ***make heat*** and then, once the binary is generated, we execute ***./heat test.dat -a 0 -o heat-jacobi.ppm***. As we can see, the program is executed with a configuration file (*test.dat*, on the right) that specifies the number of heat sources, their position, size and temperature. The *-a* option* lets us choose the solver to be used: 0 for *Jacobi* (default) and 1 for *Gauss-Seidel(./heat test.dat -a 0 -o heat-gauss.ppm).*

Also, with the execution, a *heat.ppm* file is created. It shows the resulting heat diffusion when two heat sources are placed in the borders of the 2D solid (one in the upper left corner and the other in the middle of the lower border). We create a copy, for validation purposes. We visualise the image file generated with ***display heat-jacobi.ppm***. Next, we do the same, but with the *Gauss-Seidel* solver (***./heat test.dat -a 1***). Again, we create a copy, as we will need both versions later to check the correctness of the parallel versions we will program. We visualise it; we can see both images in the table on the next page.

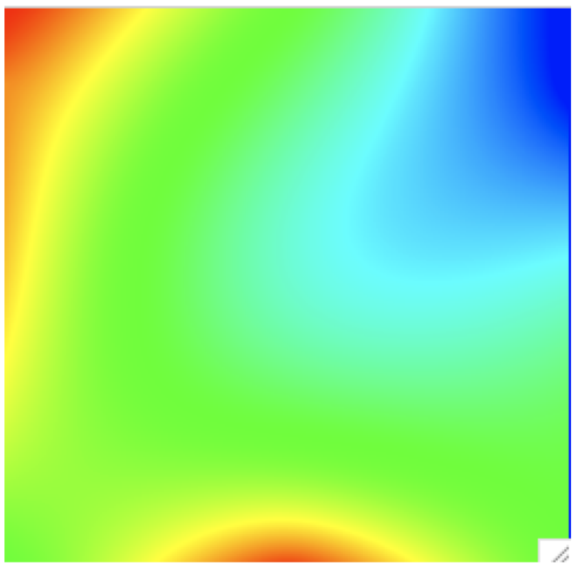
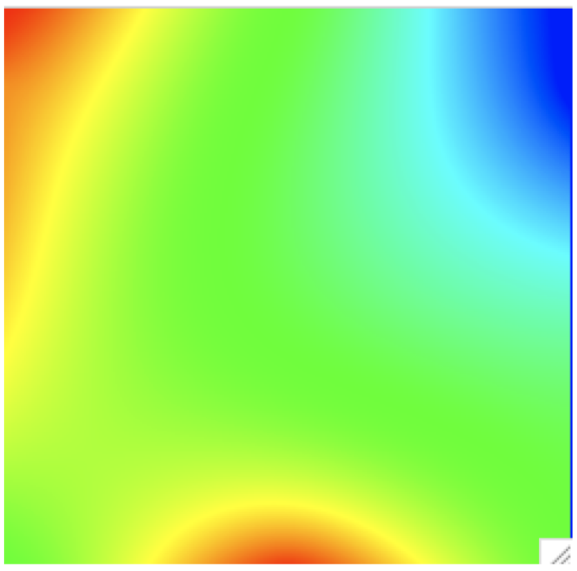
Jacobi	*Gauss-Seidel*
	

Figure 1: Image representing the temperature in each point of the 2D solid body with both algorithms Jacobi and Gauss-Seidel.

As we can see, there is a slight difference when using one solver or another: in the *Jacobi* version, the cold region is bigger.

2. Analysis of task granularities and dependencies

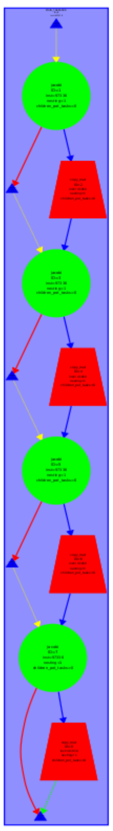
We'll do the analysis of the task graphs generated (and therefore the analysis of the dependencies) with *Tareador* when using both *Jacobi* and *Gauss-Seidel*.

In the initial version of the *solver.c* code, we already have a coarse-grain task definition that we will later refine.

```
iter = 0;
while(1) {
switch( param.algorithm ) {
    case 0: // JACOBI
        tareador_start_task("jacobi");
        residual = solve(param.u, param.uhelp, np, np);
        tareador_end_task("jacobi");
        // Copy uhelp into u
        tareador_start_task("copy_mat");
        copy_mat(param.uhelp, param.u, np, np);
        tareador_end_task("copy_mat");
        break;
    case 1: // GAUSS-SEIDEL
        tareador_start_task("gausseidel");
        residual = solve(param.u, param.u, np, np);
        tareador_end_task("gausseidel");
        break;
    default: // WRONG OPTION
        fprintf(stdout, "Error: solver not implemented, exiting execution \n");
        return EXIT_FAILURE;
    }

    iter++;
```

Figure 2: part of the code of the heat-tareador.c

We can see that the tasks are both called with the function solve, but Jacobi will also use the copy mat function to store a copy of a matrix that will help improve its algorithm.



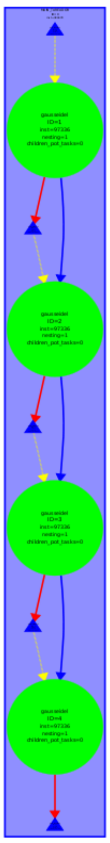
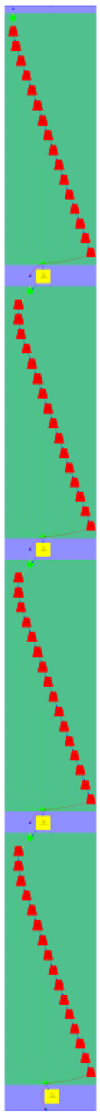
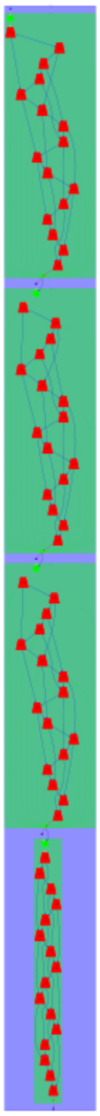


Figure 3 (left): heat-tareador.c decomposition for Jacobi algorithm.

Figure 4 (right): heat-tareador.c decomposition for Gauss-Seidel algorithm.

We compile the code with ***make heat*** and execute ***./run_tareador.c heat 0***, obtaining the task graph for the *Jacobi* solver (on the left), where the green tasks correspond to the *relax_jacobi* tasks and the red ones to *copy_mat*; we do the same but changing the 0 to **1** to get the graph for the *Gauss-Seidel* solver (on the right).



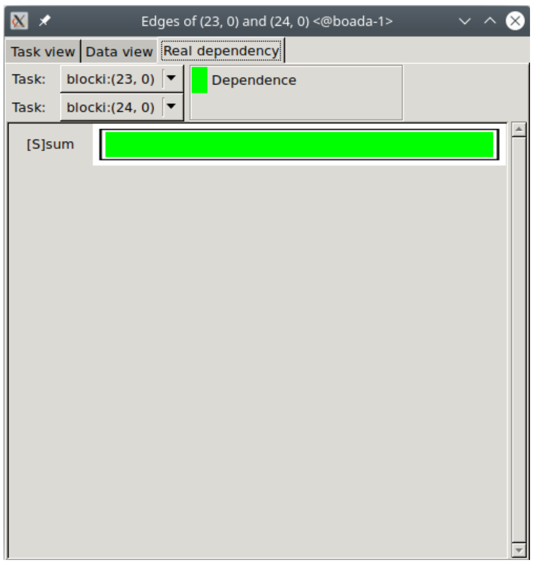


Because the tasks form a chain in both cases, we can conclude that at this granularity level, there isn't any parallelism that can be exploited. This is why we will go on to explore a finer granularity.

Figure 5 (left): heat-tareador.c decomposition for each block with Jacobi algorithm.

Figure 6 (right): heat-tareador.c decomposition for each block with Gauss-Seidel algorithm.

Next, we will analyse the code and see if they have some variables that can cause dependencies.

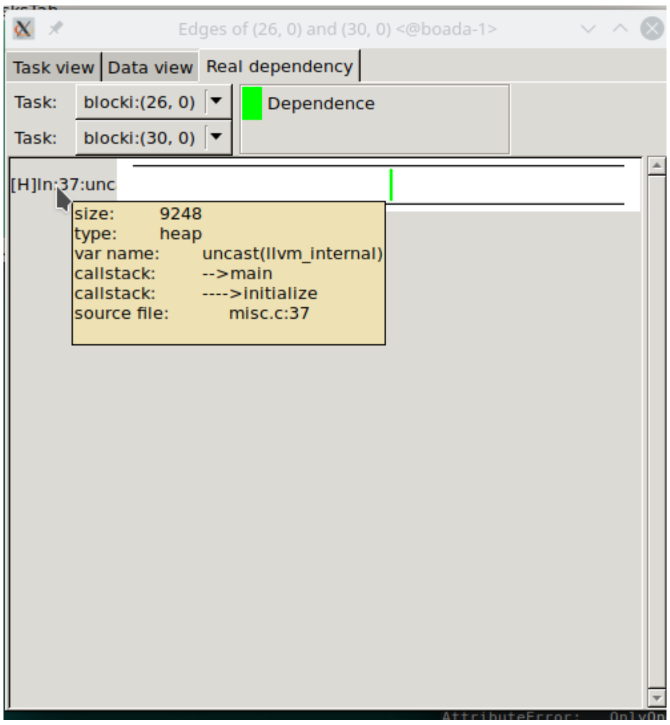


Task: blocki:(23, 0)
Task: blocki:(24, 0)

Figure 7: Real dependency graph of Jacobi algorithm.

Each block of the Jacobi algorithm depends on the *sum* variable.

In the Gauss-Seidel case, each block depends on the two previous blocks (except the first two blocks, which have no dependencies).



Task: blocki:(26, 0)
Task: blocki:(30, 0)

```
size:       9248
type:       heap
var name:       uncast(llvm_internal)
callstack:      -->main
callstack:      ---->initialize
source file:        misc.c:37
```

Figure 8: Real dependency graph of Gauss algorithm

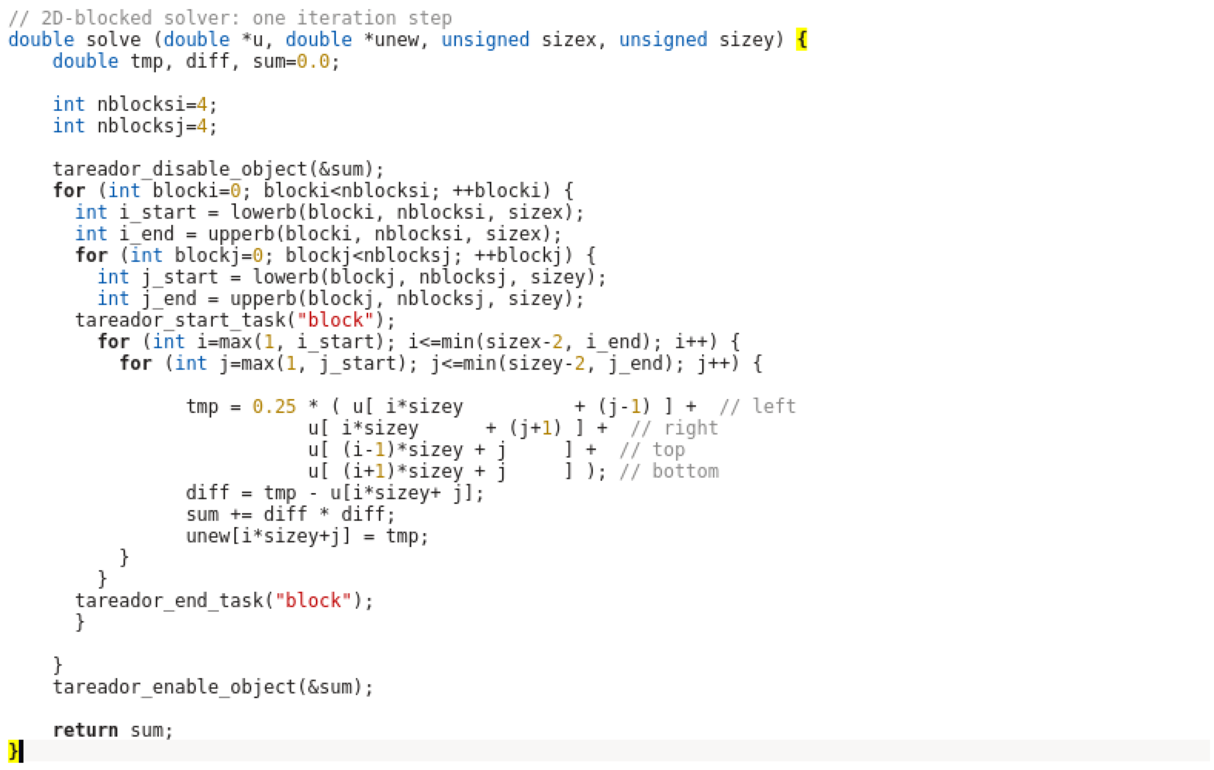
Line 37 of the code *misc.c*

```
// allocate memory
//
(param->u)      = (double*)calloc( sizeof(double),np*np );
(param->uhelp) = (double*)calloc( sizeof(double),np*np );
(param->uvis)  = (double*)calloc( sizeof(double),
                                  (param->visres+2) *
                                  (param->visres+2) );
```

We can see the dependency is caused by access to the heap by a pointer (in dynamic memory).

Next we will proceed with adding *Tareador* clauses to examine the code.

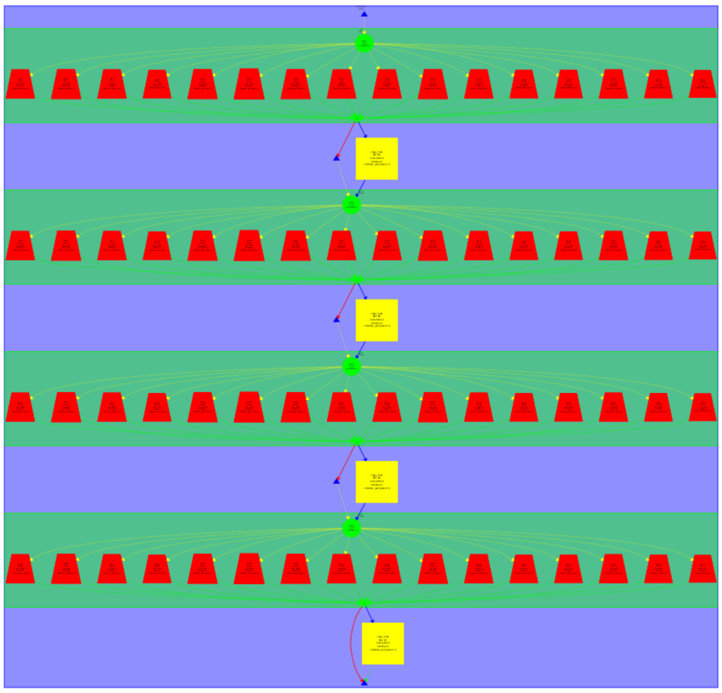
After adding our Tareador clauses and uncommenting *tareador_disable_object* and *tareador_enable_object* call, the following task dependencies graphs are obtained.



```
// 2D-blocked solver: one iteration step
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");
        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");
      }

    }
    tareador_enable_object(&sum);

    return sum;
}
```

Figure 9: code of solver-tareador.c with the Tareador clauses.

Using the Jacobi algorithm with *Tareador* we obtain the next graph:



We are obtaining more parallelism this way, as we can see in this diagram the parallelism is better with 4 processors, so the red function will be executed in parallel.

Figure 10: heat-tareador.c code with the "Tareador disable_objects" clause on, with the Jacobi algorithm.

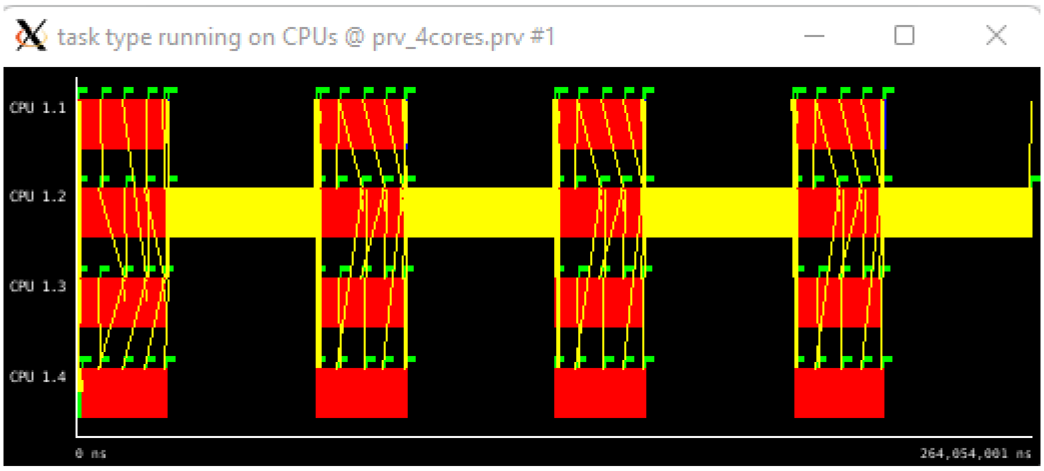


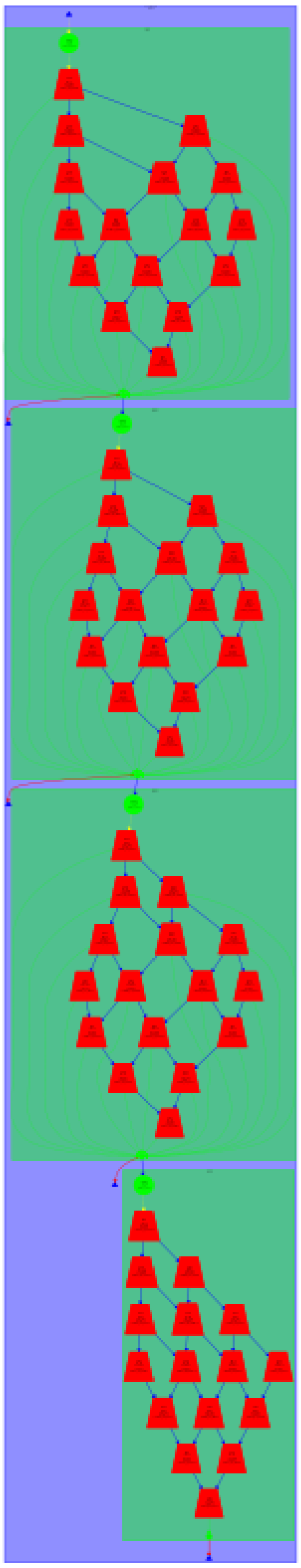
Figure 11: Paraver trace of the Jacobi algorithm.

We can also see that one thread is executing all the code, creating tasks for all threads to execute the following tasks in red with parallelism, as we can see the parallelism can be done in a better way, that's what we will explore in the next part of the deliverable.

To protect the different variables in this code we can use the *reduction* clause, these variables are for example sum, we will talk about this below on this deliverable.

To execute the Gauss-Seidel algorithm we will put in the input parameters the number 1, so we execute the second algorithm provided, which is this one.

Gauss-Seidel: We will execute the code with 4 processors.



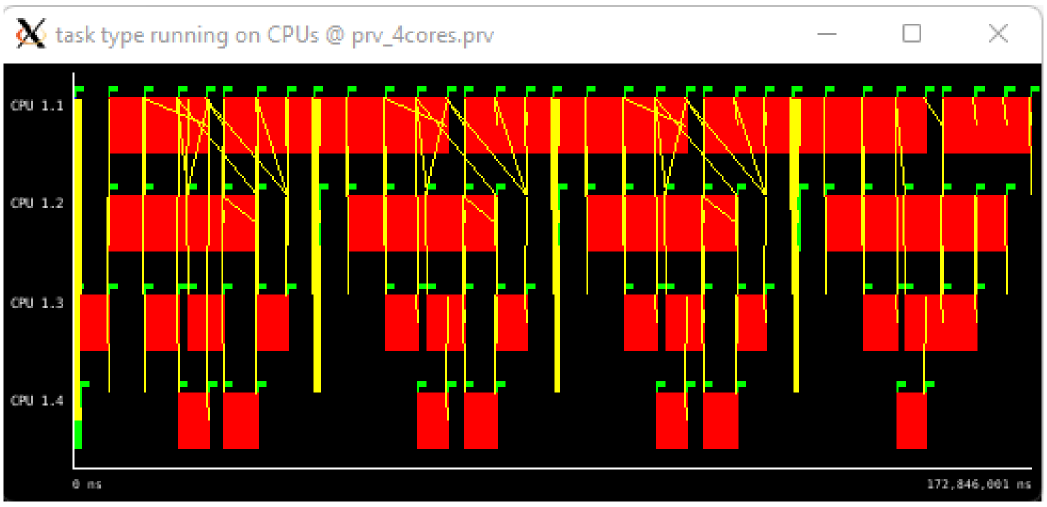


Figure 12: heat-tareador.c code with the “Tareador disable_objects” clause on, with the Gauss-Seidel algorithm.

Figure 13 (left): Paraver trace of the Jacobi algorithm.

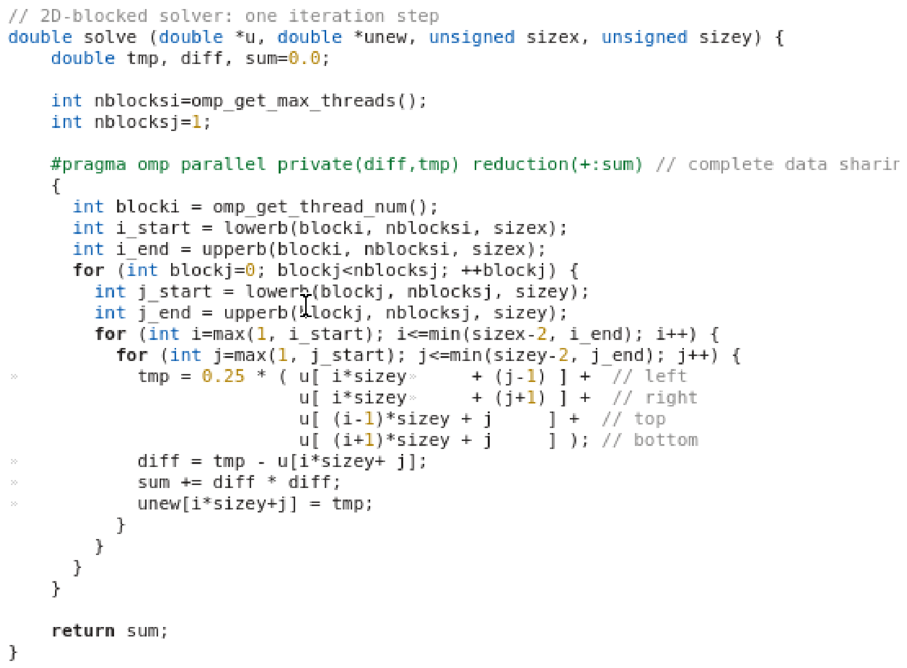
We can see that the parallelization obtained, isn’t as great as in the *Jacobi* solver case, apart from the obvious distribution of the tasks, it is also reflected by the execution times shown in the bottom right corner of the execution simulations.

3. Parallelisation of the heat equation solvers

3.1. Jacobi solver

In this section, we’ll parallelize the ***Jacobi*** sequential code found in *solver.c* (even though our updates will be made in a copy of the file, *solver-omp.c*). We will be using **implicit tasks** with the *#pragma omp parallel* construct and a **geometric block data decomposition by rows** or matrices *u* and *unew*.

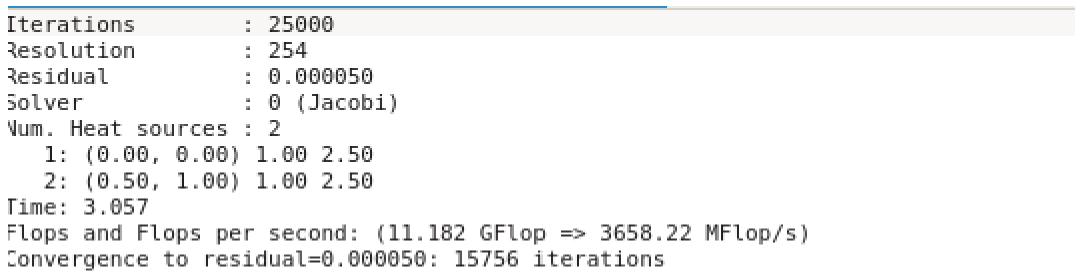
We will use the next clauses to protect the variables diff, tmp and sum.



```c
// 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(diff,tmp) reduction(+:sum) // complete data sharir
    {
      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 14: code of the function solver-omp.c using parallelisation clauses to protect the variables.

We execute the command `sbatch submit-omp.sh heat-omp 0 8`



```
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: 3.057
Flops and Flops per second: (11.182 GFlop => 3658.22 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Figure 15: result of the execution of the heat-omp.c code using Jacobi algorithm with the submit-omp.sh, also with 8 processors.

In *Jacobi*, as we said in the previous section of this document, *sum* will be protected with the *reduction* clause. Also, we have made *diff and tmp privates* due to the fact that are declared outside the parallel region, making them shared by default; if we were to leave it that way, we'd have to worry about the data race conditions.

When we execute the diff command between this version and the sequential one, we can see that both generate the same heat.

We execute the **submit-strong.sh** to generate the plot:

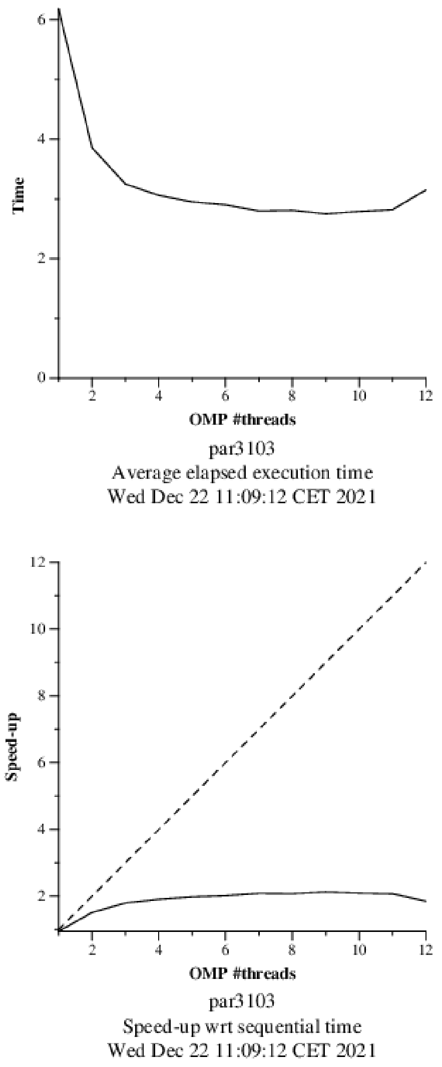


Figure 16: Scalability plot of the solver-omp.c mentioned before, generated with submit-strong.sh.

We execute the command **submit-extrae.sh** to generate the paraver trace.

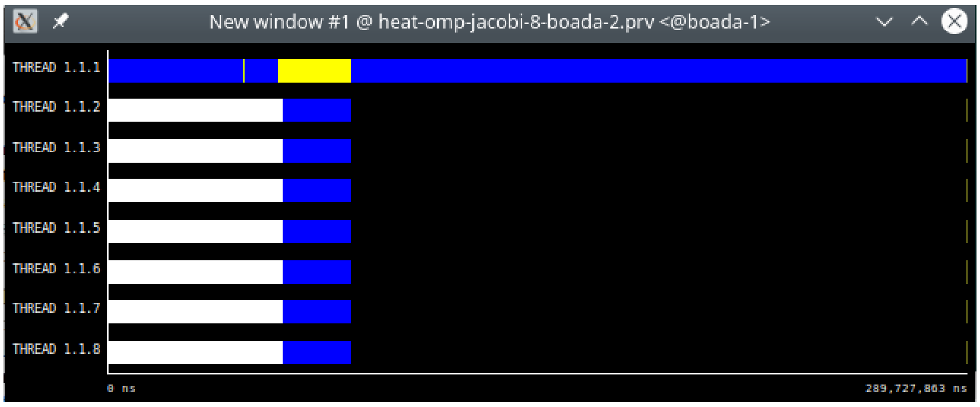
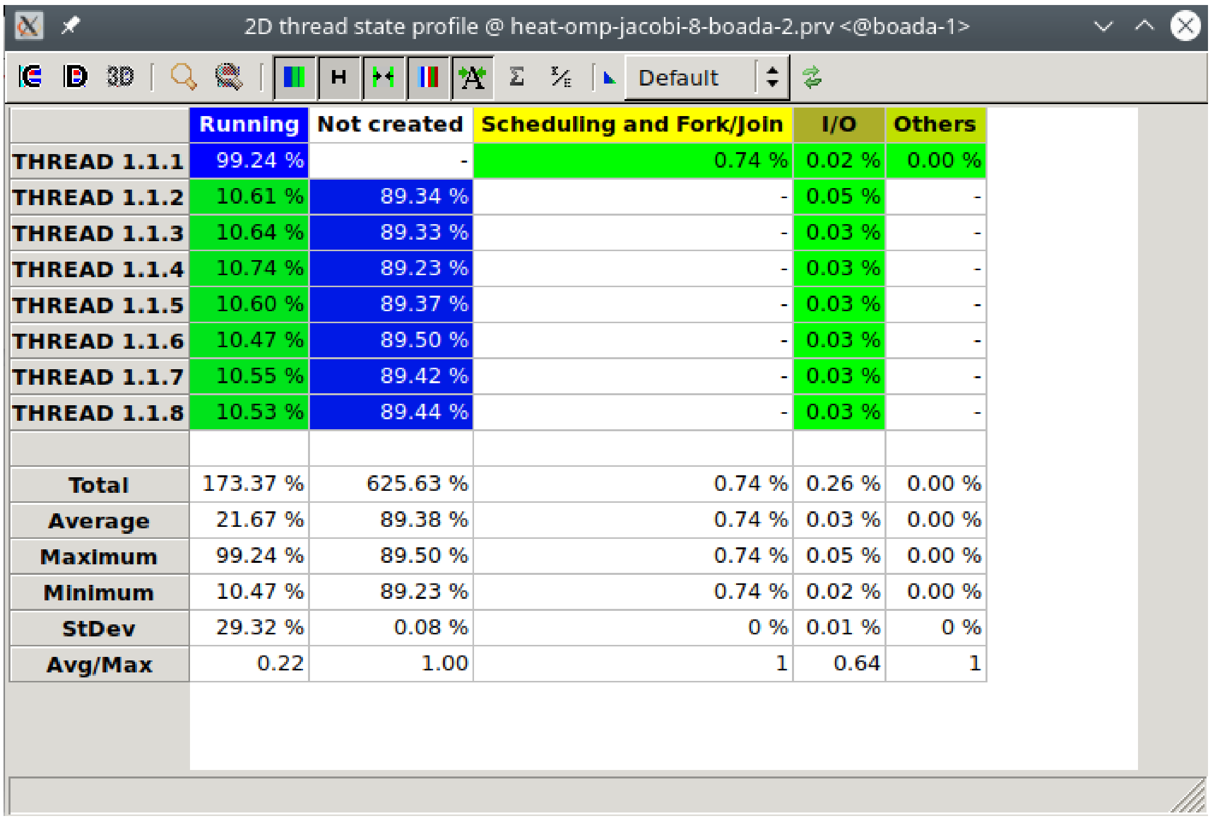


Figure 17: Paraver trace of solver-omp.c code.

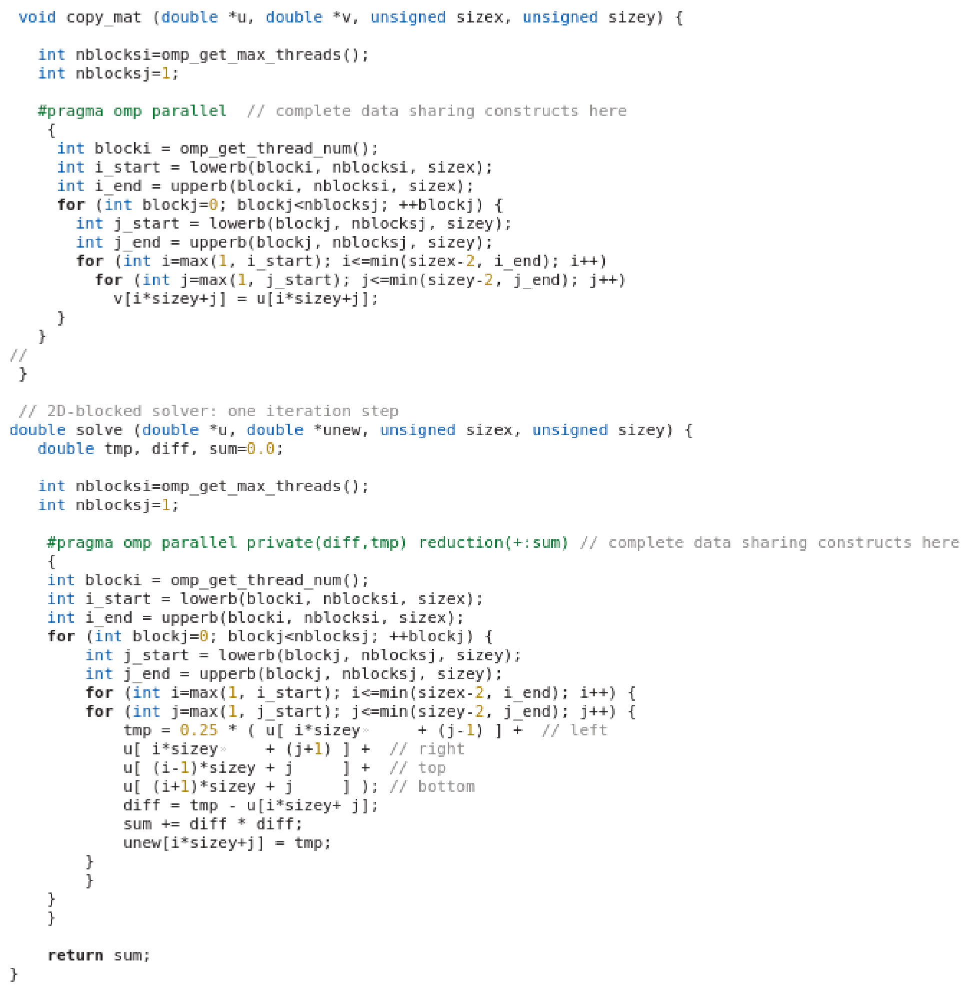
We can see that the trace has one thread executing almost all the tasks, that means parallelisation is needed from one point of the code for better speed-up.


2D thread state profile @ heat-omp-jacobi-8-boada-2.prv <@boada-1>
Default

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.24 %	-	0.74 %	0.02 %	0.00 %
THREAD 1.1.2	10.61 %	89.34 %	-	0.05 %	-
THREAD 1.1.3	10.64 %	89.33 %	-	0.03 %	-
THREAD 1.1.4	10.74 %	89.23 %	-	0.03 %	-
THREAD 1.1.5	10.60 %	89.37 %	-	0.03 %	-
THREAD 1.1.6	10.47 %	89.50 %	-	0.03 %	-
THREAD 1.1.7	10.55 %	89.42 %	-	0.03 %	-
THREAD 1.1.8	10.53 %	89.44 %	-	0.03 %	-
Total	173.37 %	625.63 %	0.74 %	0.26 %	0.00 %
Average	21.67 %	89.38 %	0.74 %	0.03 %	0.00 %
Maximum	99.24 %	89.50 %	0.74 %	0.05 %	0.00 %
Minimum	10.47 %	89.23 %	0.74 %	0.02 %	0.00 %
StDev	29.32 %	0.08 %	0 %	0.01 %	0 %
Avg/Max	0.22	1.00	1	0.64	1

Figure 18: Paraver trace of solver-omp.c code using Jacobi algorithm.

Next, we will use parallelisation to implement the Jacobi, and we will use the parallelisation clauses on the next figure:



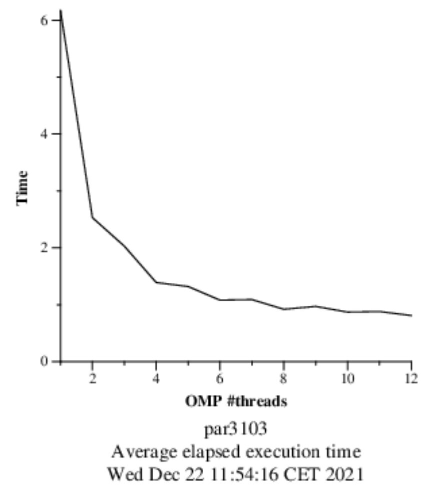
```c
 void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {

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

   #pragma omp parallel  // 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++)
           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;

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

    #pragma omp parallel private(diff,tmp) 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 19: code solver-omp.c with the Jacobi algorithm

We will substitute the first for loop from the code, and we will initialize the *i_start* and *i_end* with the *lowerb* and *upperb* functions in the copy_mat function. We will also initialize the blocki variable to omp_get_thread_num(), so now we will not go over all the blocks with the for, instead we will call the corresponding functions, so they call if it is upper block or lower block.

We use *omp_get_**max**_threads()* and **not** *omp_get_**num**_threads()*, as we've seen in the *Data decomposition strategies* section of this document, because *nblocksi* is declared outside the parallel region, not inside it; if it were inside we'd need the number of threads in the team executing the parallel region (*omp_get_num_threads*), not the number of threads that the parallel construct can create (*omp_get_max_threads*).

Finally, we will use the command diff to compare the heat between both Jacobi implementations, and we saw we have the same one, so we know that the implementation was correct.



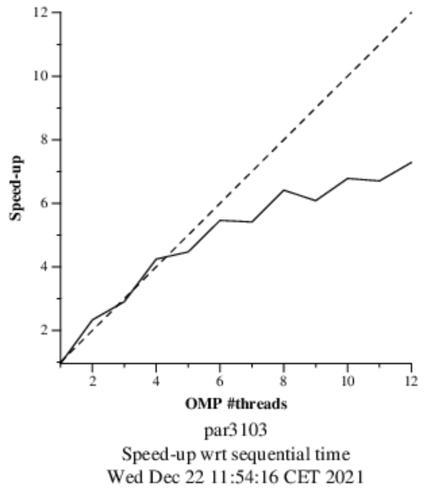


Figure 20: Scalability plot of the solver-omp.c mentioned before, generated with submit-strong.sh.

In the plot, we can see that the parallelisation used for Jacobi is better than the one implemented before.

We can see that the speed-up is far from ideal. Until 4 threads, it is quite perfect, but once we start using more than 4 threads, the speed-up line gets farther away from the diagonal each time we increment the number of processors used.

From the plots, we can see that the ideal number of threads to be used is 11, seen as there's a decay in the speed-up once 8 threads are reached (clearly seen in the second plot).

3.2. Gauss-Seidel solver

In this section, we'll parallelize the *Gauss-Seidel* solver, following the same ***geometric block by rows data decomposition*** we used with the *Jacobi* solver. We'll do it with **implicit tasks**, same as before. We'll also need to take into account the *memory consistency* problem (the fact that the compiler tries to access the memory as little as possible, placing variables in registers and only reading/writing them from/to memory at a certain point in the program, not each time as we want) that may occur and the **ordering constraints** among implicit tasks that we'll have to implement.

We parallelize the solver (code on next page). We know that the implicit tasks cannot synchronise themselves with task dependencies, which is why we create our own synchronisation object: ***sync*** ; as we can see, it is a **shared** array with an entry for each thread (*sync[nblocksi]*), initialized to 0, and **all accesses to it always access to memory** because of the ***#pragma omp atomic write***, used when updating ***sync*** and the ***#pragma omp atomic read***, when reading it.

Our implementation of the Gauss-Seidel solver is the following:

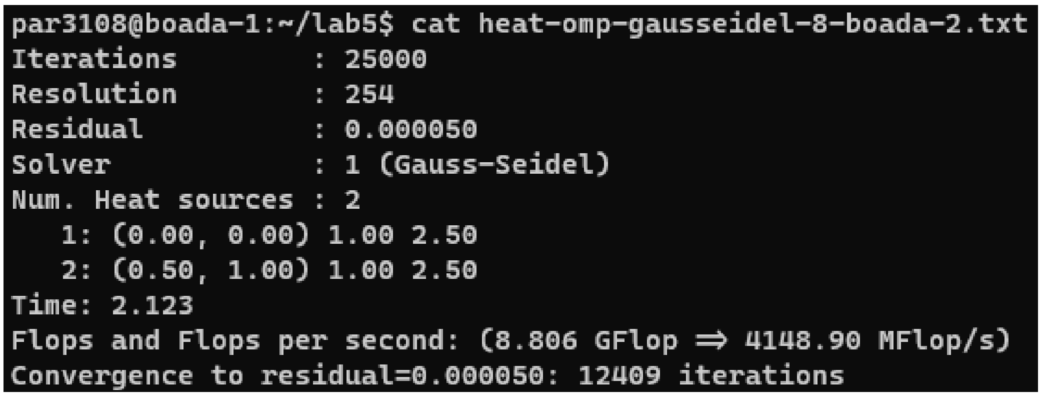
```
double solve (double *u, double *unew, unsigned sizex, unsigned sizey){
  double tmp, diff, sum = 0.0;
  int nblocksi = omp_get_max_threads ();
  int nblocksj = omp_get_max_threads ();
  int next[nblocksi][nblocksj];
  if (u == unew) {
    for (int i = 0; i < nblocksi; i++) {
      for (int j = 0; j < nblocksj; j++) {
        next[i][j] = 0;
      }
    }
  }
#pragma omp parallel private(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);
      if(u == unew && blocki != 0) {
        int aux2;
        do {
          #pragma omp atomic read
          aux2 = next[blocki-1][blockj];
        } while(aux2 == 0);
      }
      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;
        }
      }
      if(u == unew) { //gauss
        #pragma omp atomic write
        next[blocki][blockj] = 1;
      }
    }
  }
  return sum;
}
```

In order to store all the blocks, we created a matrix *next* that is accessed and written using the pragma clauses *atomic read/write*.

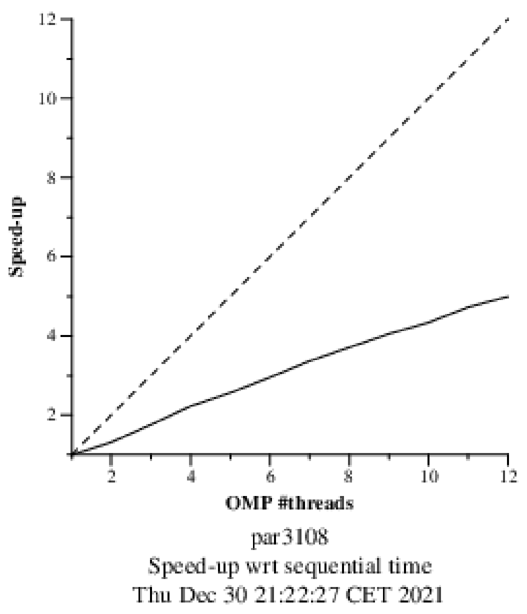
To be sure that the read is correct, we check that the Gauss-Seidel algorithm is being used (`u == unew`) and that we are not in the topmost *j block* (`blocki != 0`). Only then, we access the matrix from the main memory. If that position had not been accessed before, then we wait.

Before accessing the next *i block*, we check again if the Gauss algorithm is being used. If it is, then we store to main memory that the *i block* has been computed using the pragma *atomic write*.

Next, we compile and submit it using *make heat-omp* and *sbatch submit-omp.sh heat-omp 1 8*.


```
par3108@boada-1:~/lab5$ cat heat-omp-gausseidel-8-boada-2.txt
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: 2.123
Flops and Flops per second: (8.806 GFlop => 4148.90 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

In order to check its scalability using more threads, we submit the script using *sbatch submit-strong-omp.sh 1*.



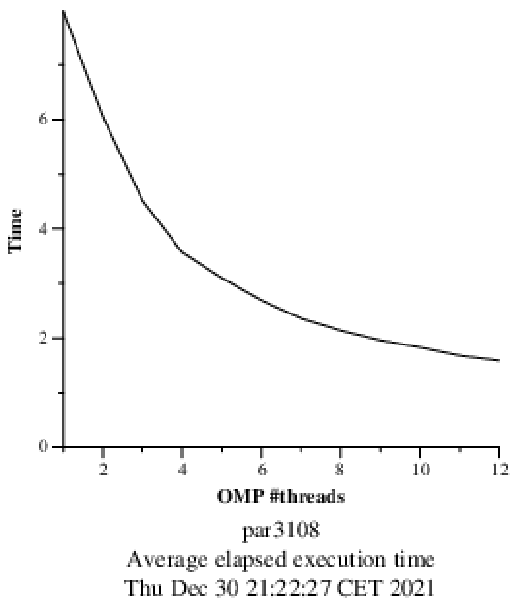


Figure 21: Scalability plot of the solver-omp.c using Gauss-Seidel algorithm.

Now we generate the trace using *sbatch ./submit-extrae.sh heat-omp 1 8* and, with paraver, we obtained the following plots.

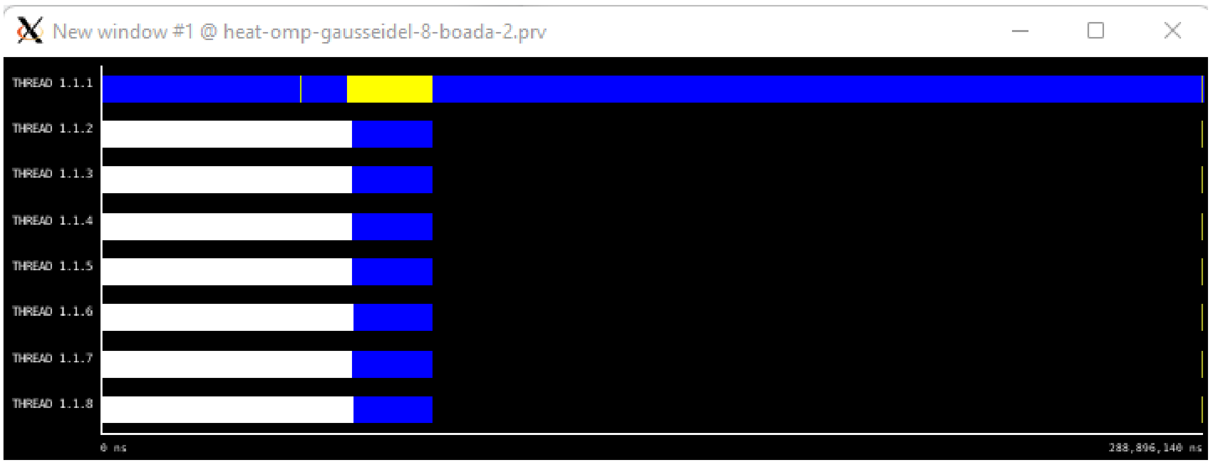
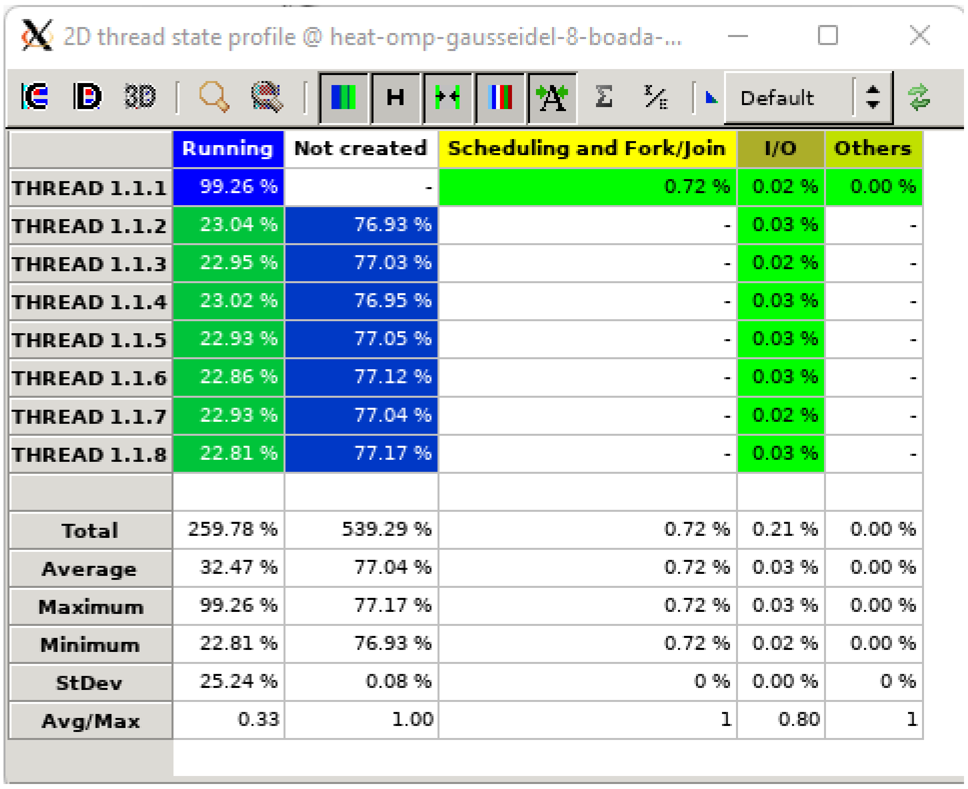


Figure 21: Parallel execution of the solver-omp.c script using Gauss-Seidel algorithm.



	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.26 %	-	0.72 %	0.02 %	0.00 %
THREAD 1.1.2	23.04 %	76.93 %	-	0.03 %	-
THREAD 1.1.3	22.95 %	77.03 %	-	0.02 %	-
THREAD 1.1.4	23.02 %	76.95 %	-	0.03 %	-
THREAD 1.1.5	22.93 %	77.05 %	-	0.03 %	-
THREAD 1.1.6	22.86 %	77.12 %	-	0.03 %	-
THREAD 1.1.7	22.93 %	77.04 %	-	0.02 %	-
THREAD 1.1.8	22.81 %	77.17 %	-	0.03 %	-
Total	259.78 %	539.29 %	0.72 %	0.21 %	0.00 %
Average	32.47 %	77.04 %	0.72 %	0.03 %	0.00 %
Maximum	99.26 %	77.17 %	0.72 %	0.03 %	0.00 %
Minimum	22.81 %	76.93 %	0.72 %	0.02 %	0.00 %
StDev	25.24 %	0.08 %	0 %	0.00 %	0 %
Avg/Max	0.33	1.00	1	0.80	1

Figure 22: Paraver trace of solver-omp.c code using Gauss-Seidel algorithm.

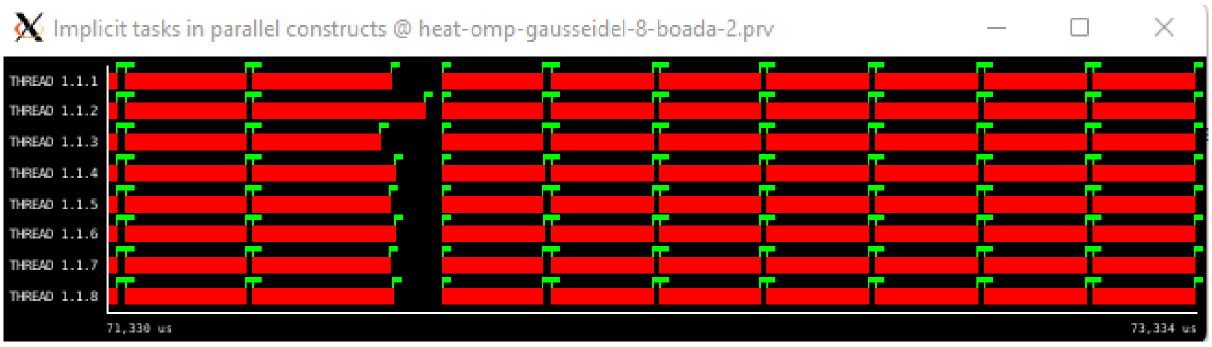
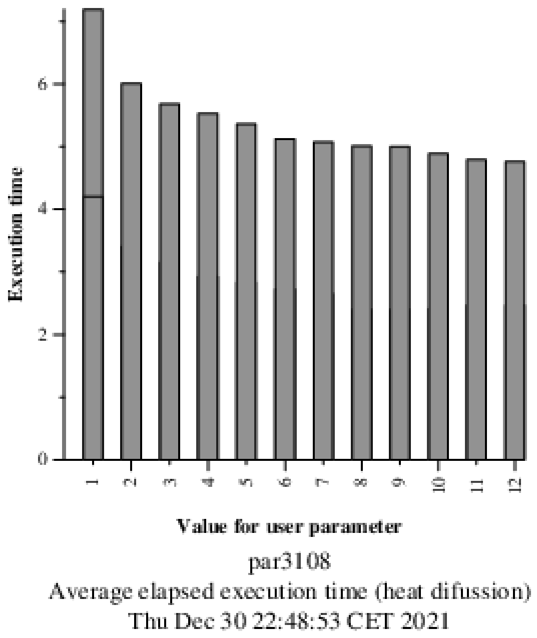


Figure 23: Implicit tasks in parallel constructs of the execution of solver-omp.c using Gauss-Seidel algorithm.

Finally, we updated the way the value of *nblocksj* is assigned. Instead of just setting it to **`omp_get_max_threads()`**, we changed it to **`omp_get_max_threads()*userparam`,** so now we ought to get even more parallelism.

We submit the modified script using sbatch *submit-userparam-omp.sh 8*, where '8' is the number of threads used.



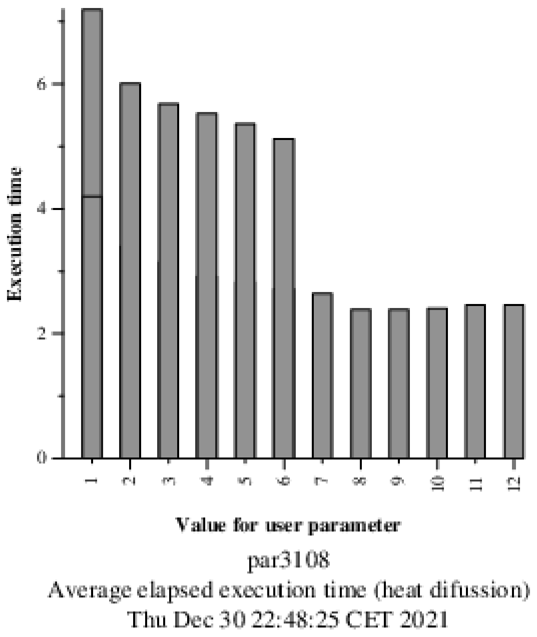
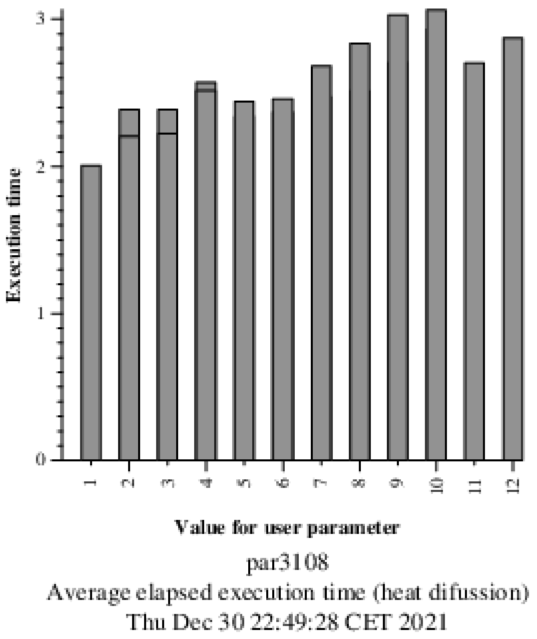


Figure 23 (left): Execution times of solver-omp.c using Gauss-Seidel algorithm and 2 threads.
Figure 24 (right): Execution times of solver-omp.c using Gauss-Seidel algorithm and 4 threads.



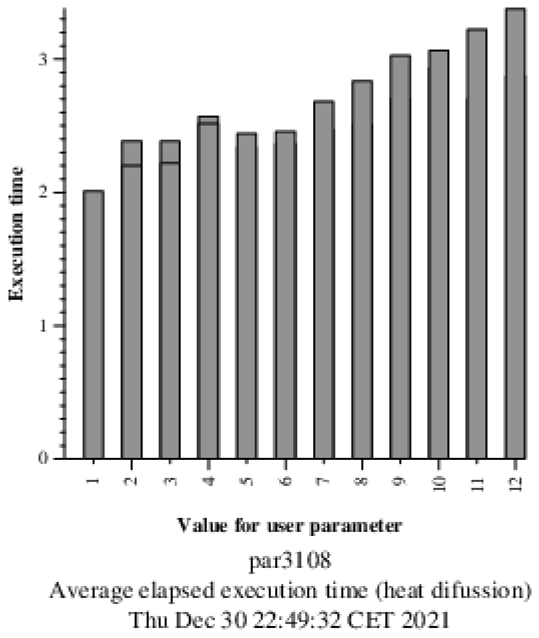


Figure 25 (left): Execution times of solver-omp.c using Gauss-Seidel algorithm and 8 threads.
Figure 26 (right): Execution times of solver-omp.c using Gauss-Seidel algorithm and 16 threads.

As the plots above show, the change from using 4 to 8 threads is very significant, especially when *userparam* is low. Yet, is we use even more threads, we see no real improvement in terms of performance.

4. Conclusions

After analysing several possibilities for the parallelization of the solver using both Jacobi and Gauss-Seidel algorithms, we can conclude that the second option is the best in terms of performance.

The Gauss-Seidel algorithm parallelized obtains better times and better speed-ups.